Matt Pharr, Wenzel Jakob, Greg Humphreys

PHYSICALLY BASED RENDERING

From Theory to Implementation

Third Edition

MK
MORGAN KAUFMANN

Physically Based Rendering is a terrific book. It covers all the marvelous math, fascinating physics, practical software engineering, and clever tricks that are necessary to write a state-of-the-art photorealistic renderer. All of these topics are dealt with in a clear and pedagogical manner without omitting the all-important practical details.

pbrt is not just a "toy" implementation of a ray tracer but a general and robust full-scale global illumination renderer. It contains many important optimizations to reduce execution time and memory consumption for complex scenes. Furthermore, pbrt is easy to extend to experiment with other rendering algorithm variations.

This book is not only a textbook for students but also a useful reference book for practitioners in the field. The third edition has been extended with new sections on bidirectional path tracing, realistic camera models, and a state-of-the-art explanation of subsurface scattering.

Per Christensen
Senior Software Developer, RenderMan Products, Pixar Animation Studios

Looking for a job in research or high end rendering? Get your kick-start education and create your own project with this book that comes along with both theory and real examples, meaning real code and real content for your renderer.

With their third edition, Matt Pharr, Greg Humphreys, and Wenzel Jakob provide easy access to even the most advanced rendering techniques like multiplexed Metropolis light transport and quasi-Monte Carlo methods. Most importantly, the framework lets you skip the bootstrap pain of getting data into and out of your renderer.

The holistic approach of literate programming results in a clear logic of an easy-to-study text. If you are serious about graphics, there is no way around this unique and extremely valuable book that is closest to the state of the art.

Alexander Keller
Director of Research, NVIDIA

Physically Based Rendering

FROM THEORY TO IMPLEMENTATION

THIRD EDITION

MATT PHARR

WENZEL JAKOB

GREG HUMPHREYS

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

Morgan Kaufmann is an imprint of Elsevier
50 Hampshire Street, 5th Floor, Cambridge, MA 02139, USA

Notices
Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods, professional practices, or medical treatment may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the Library of Congress

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

ISBN: 978-0-12-800645-0

Publisher: Todd Green
Editorial Project Manager: Jennifer Pierce
Production Project Manager: Mohana Natarajan
Cover Designer: Victoria Pearson

Typeset by: Windfall Software and SPi global

To Deirdre, who even let me bring the manuscript on our honeymoon.

M. P.

To Olesya, who thought it was cute that my favorite book is a computer program.

W. J.

To Isabel and Leila, the two most extraordinary people I've ever met. May your pixels never be little squares.

G. H.

ABOUT THE AUTHORS

Matt Pharr is a Software Engineer at Google. He previously co-founded Neoptica, which was acquired by Intel, and co-founded Exluna, which was acquired by NVIDIA. He has a B.S. degree from Yale and a Ph.D. from the Stanford Graphics Lab, where he worked under the supervision of Pat Hanrahan.

Wenzel Jakob is an assistant professor in the School of Computer and Communication Sciences at École Polytechnique Fédérale de Lausanne (EPFL). His research interests revolve around material appearance modeling, rendering algorithms, and the high-dimensional geometry of light paths. Wenzel obtained his Ph.D. at Cornell University under the supervision of Steve Marschner, after which he joined ETH Zürich for post-doctoral studies under the supervision of Olga Sorkine Hornung. Wenzel is also the lead developer of the Mitsuba renderer, a research-oriented rendering system.

Greg Humphreys is Director of Engineering at FanDuel, having previously worked on the Chrome graphics team at Google and the OptiX GPU ray-tracing engine at NVIDIA. Before that, he was a professor of Computer Science at the University of Virginia, where he conducted research in both high-performance and physically based computer graphics, as well as computer architecture and visualization. Greg has a B.S.E. degree from Princeton and a Ph.D. in Computer Science from Stanford under the supervision of Pat Hanrahan. When he's not tracing rays, Greg can usually be found playing tournament bridge.

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. Nearly all of the images in this book, including the one on the front cover, 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.

AUDIENCE

There are three main audiences that this book is intended for. The first 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.

The second audience is advanced graduate students and researchers in computer graphics. For those doing research in rendering, the book provides a broad introduction to the area, and the `pbrt` source code provides a foundation that can be useful to build upon (or at least to use bits of source code from). For those working in other areas, we believe that having a thorough understanding of rendering can be helpful context to carry along.

Our final audience is software developers in industry. Although many of the ideas in this book will likely be familiar to this audience, seeing explanations of the algorithms presented in the literate style may provide new perspectives. `pbrt` includes implementations of a number of advanced and/or difficult-to-implement algorithms and techniques, such as subdivision surfaces, Monte Carlo sampling algorithms, bidirectional path tracing, Metropolis sampling, and subsurface scattering; 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 Friedrich 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, numerical precision, 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 an extensible architecture, with the core of the system implemented in terms of a set of carefully designed abstract base classes, and as much of the specific functionality as possible in implementations of these base classes. The result is that one doesn't need to understand all of the specific implementations 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 unacceptably 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.

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. Numerical 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 in the pbrt software distribution 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*. The latest version of the pbrt source code is available from this site, and we will also post errata and bug fixes, additional scenes to render, and supplemental utilities. 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!

CHANGES BETWEEN THE FIRST AND SECOND EDITIONS

Six years passed between the publication of the first edition of this book in 2004 and the second edition in 2010. In that time, thousands of copies of the book were sold, and the pbrt software was downloaded thousands of times from the book's Web site. The pbrt user base gave us a significant amount of feedback and encouragement, and our experience with the system guided many of the decisions we made in making changes between the version of pbrt presented in the first edition and the version in the second edition. In addition to a number of bug fixes, we also made several significant design changes and enhancements:

- Removal of the plugin architecture. The first version of pbrt used a run-time plugin architecture to dynamically load code for implementations of objects like shapes, lights, integrators, cameras, and other objects that were used in the scene currently being rendered. This approach allowed users to extend pbrt with new object types (e.g., new shape primitives) without recompiling the entire rendering system. This approach initially seemed elegant, but it complicated the task of supporting pbrt on multiple platforms and it made debugging more difficult. The only new usage scenario that it truly enabled (binary-only distributions of pbrt or binary plugins) was actually contrary to our pedagogical and open-source goals. Therefore, the plugin architecture was dropped in this edition.
- Removal of the image-processing pipeline. The first version of pbrt provided a tone-mapping interface that converted high-dynamic-range (HDR) floating-point output images directly into low-dynamic-range TIFFs for display. This functionality made sense in 2004, as support for HDR images was still sparse. In 2010, however, advances in digital photography had made HDR images commonplace. Although the theory and practice of tone mapping are elegant and worth learning, we decided to focus the new book exclusively on the process of image formation and skip the topic of image display. Interested readers should read the book written by Reinhard et al. (2010) for a thorough and modern treatment of the HDR image display process.
- Task parallelism. Multicore architectures became ubiquitous, and we felt that pbrt would not remain relevant without the ability to scale to the number of locally available cores. We also hoped that the parallel programming implementation details documented in this book would help graphics programmers understand some of the subtleties and complexities in writing scalable parallel code (e.g., choosing appropriate task granularities), which is still a difficult and too infrequently taught topic.
- Appropriateness for "production" rendering. The first version of pbrt was intended exclusively as a pedagogical tool and a stepping-stone for rendering research. Indeed, we made a number of decisions in preparing the first edition that were contrary to use in a production environment, such as limited support for image-based lighting, no support for motion blur, and a photon mapping implementation that wasn't robust in the presence of complex lighting. With much improved support for these features as well as support for subsurface scattering and Metropolis light transport, we feel that with the second edition, pbrt became much more suitable for rendering very high-quality images of complex environments.

CHANGES BETWEEN THE SECOND AND THIRD EDITIONS

With the passage of another six years, it was time to update and extend the book and the pbrt system. We continued to learn from readers' and users' experiences to better understand which topics were most useful to cover. Further, rendering research continued apace; many parts of the book were due for an update to reflect current best practices. We made significant improvements on a number of fronts:

- Bidirectional light transport. The third version of pbrt now includes a full-featured bidirectional path tracer, including full support for volumetric light transport and multiple importance sampling to weight paths. An all-new Metropolis light transport integrator uses components of the bidirectional path tracer, allowing for a particularly succinct implementation of that algorithm. The foundations of these algorithms were established approximately fifteen years ago; it's overdue to have solid support for them in pbrt.
- Subsurface scattering. The appearance of many objects—notably, skin and translucent objects—is a result of subsurface light transport. Our implementation of subsurface scattering in the second edition reflected the state of the art in the early 2000s; we have thoroughly updated both our BSSRDF models and our subsurface light transport algorithms to reflect the progress made in ten subsequent years of research. We now use a considerably more accurate diffusion solution together with a ray-tracing-based sampling technique, removing the need for the costly preprocessing step used in the second edition.
- Numerically robust intersections. The effects of floating-point round-off error in geometric ray intersection calculations have been a long-standing challenge in ray tracing: they can cause small errors to be present throughout the image. We have focused on this issue and derived conservative (but tight) bounds of this error, which makes our implementation more robust to this issue than previous rendering systems.
- Participating media representation. We have significantly improved the way that scattering media are described and represented in the system; this allows for more accurate results with nested scattering media. A new sampling technique enables unbiased rendering of heterogeneous media in a way that cleanly integrates with all of the other parts of the system.
- Measured materials. This edition includes a new technique to represent and evaluate measured materials using a sparse frequency-space basis. This approach is convenient because it allows for exact importance sampling, which was not possible with the representation used in the previous edition.
- Photon mapping. A significant step forward for photon mapping algorithms has been the development of variants that don't require storing all of the photons in memory. We have replaced pbrt's photon mapping algorithm with an implementation based on stochastic progressive photon mapping, which efficiently renders many difficult light transport effects.
- Sample generation algorithms. The distribution of sample values used for numerical integration in rendering algorithms can have a surprisingly large effect on the quality of the final results. We have thoroughly updated our treatment of this topic, covering new approaches and efficient implementation techniques in more depth than before.

Many other parts of the system have been improved and updated to reflect progress in the field: microfacet reflection models are treated in more depth, with much better sampling techniques; a new "curve" shape has been added for modeling hair and other fine geometry; and a new camera model that simulates realistic lens systems is now available. Throughout the book, we have made numerous smaller changes to more clearly explain and illustrate the key concepts in physically based rendering systems like pbrt.

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. Matt and Greg both feel extremely privileged to have been there.

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 acknowledge colleagues and co-workers 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. 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 Eric Veach, who has also been generous with his time and expertise. Thanks also to Doug Shult and Stan Eisenstat for formative lessons in mathematics and computer science during high school and college, respectively, and most important 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 the preparation of the first edition of the book.

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.

Wenzel Jakob was excited when the first edition of pbrt arrived in his mail during his undergraduate studies in 2004. Needless to say, this had a lasting effect on his career—thus Wenzel would like to begin by thanking his co-authors for inviting him to become a part of third edition of this book. Wenzel is extremely indebted to Steve Marschner, who was his PhD advisor during a fulfilling five years at Cornell University. Steve brought him into the world of research and remains a continuous source of inspiration. Wenzel is also thankful for the guidance and stimulating research environment created by the other members of the graphics group, including Kavita Bala, Doug James, and Bruce Walter. Wenzel spent a wonderful postdoc with Olga Sorkine Hornung who introduced him to geometry processing. Olga's support for Wenzel's involvement in this book is deeply appreciated.

For the first edition, 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 Christer Ericson, who had a number of suggestions for improving our kd-tree implementation. For the second edition, we're thankful to Christophe Hery and Eugene d'Eon for helping us with the nuances of subsurface scattering.

For the third edition, we'd especially like to thank Leo Grünschloß for reviewing our sampling chapter; Alexander Keller for suggestions about topics for that chapter; Eric Heitz for extensive help with microfacets (and reviewing our text on that topic); Thiago Ize for thoroughly reviewing the text on floating-point error; Tom van Bussel for reporting a number of errors in our BSSRDF code; Ralf Habel for reviewing our BSSRDF text; and Toshiya Hachisuka and Anton Kaplanyan for extensive review and comments about our light transport chapters. Discussions with Eric Veach about floating-point round-off error and ray tracing were extremely helpful to our development of our approach to that topic. We'd also like to thank Per Christensen, Doug Epps, Luca Fascione, Marcos Fajardo, Christiphe Hery, John "Spike" Hughes, Andrew Kensler, Alan King, Chris Kulla, Morgan McGuire, Andy Selle, and Ingo Wald for helpful discussions, suggestions, and pointers to research.

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. We'd particularly like to thank the reviewers who provided feedback on both the first and second editions of the book: 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. For the second edition, Janne Kontkanen, Nelson Max, Bill Mark, and Eric Tabellion also contributed numerous helpful suggestions.

Many people have contributed to not only pbrt but to our own better understanding of rendering through bug reports, patches, and suggestions about better implementation approaches. A few have made particularly substantial contributions over the years—we would especially like to thank Solomon Boulos, Stephen Chenney, John Danks, Kevin Egan, Volodymyr Kachurovskyi, and Ke Xu.

In addition, we would like to thank Rachit Agrawal, Frederick Akalin, Mark Bolstad, Thomas de Bodt, Brian Budge, Mark Colbert, Yunjian Ding, Tao Du, Shaohua Fan, Etienne Ferrier, Nigel Fisher, Jeppe Revall Frisvad, Robert G. Graf, Asbjørn Heid, Keith Jeffery, Greg Johnson, Aaron Karp, Donald Knuth, Martin Kraus, Murat Kurt, Larry Lai, Craig McNaughton, Swaminathan Narayanan, Anders Nilsson, Jens Olsson, Vincent Pegoraro, Srinath Ravichandiran, Sébastien Speierer, Nils Thuerey, Xiong Wei, Wei-Wei Xu, Arek Zimny, and Matthias Zwicker for their suggestions and bug reports. Finally, we would like to thank the *LuxRender* developers and the *LuxRender* community, particularly Terrence Vergauwen, Jean-Philippe Grimaldi, and Asbjørn Heid; it has been a delight to see the rendering system they have built from `pbrt`'s foundation, and we have learned from reading their source code and implementations of new rendering algorithms.

Special thanks to Martin Preston and Steph Bruning from Framestore for their help with our being able to use a frame from *Gravity* (image courtesy of Warner Bros. and Framestore), and to Joe Letteri, Dave Gouge, and Luca Fascione from Weta Digital for their help with the frame from *The Hobbit: The Battle of the Five Armies* (© 2014 Warner Bros. Entertainment Inc. and Metro-Goldwyn-Mayer Pictures Inc. (US, Canada & New Line Foreign Territories), © 2014 Metro-Goldwyn-Mayer Pictures Inc. and Warner Bros. Entertainment Inc. (all other territories). All Rights Reserved.

PRODUCTION

For the production of the first edition, 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).

For the second edition, we'd like to thank Greg Chalson who talked us into expanding and updating the book; Greg also ensured that Paul Anagnostopoulos at Windfall Software would again do the book's composition. We'd like to thank Paul again for his efforts in working with this book's production complexity. Finally, we'd also like to thank Todd Green, Paul Gottehrer, and Heather Scherer at Elsevier.

For the third edition, we'd like to thank Todd Green from Elsevier, who oversaw this go-round, and Amy Invernizzi, who kept the train on the rails throughout the process. We were delighted to have Paul Anagnostopoulos at Windfall Software part of this process for a third time; his efforts have been critical to the book's high production value, which is so important to us.

SCENES AND MODELS

Many people and organizations have generously supplied us with scenes and models for use in this book and the pbrt distribution. 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. 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 dragon model scan used in Chapters 8 and 9 is courtesy of Christian Schüller, and thanks to Yasutoshi Mori for the sports car used in Chapters 7 and 12. The glass used to illustrate caustics in Figures 16.9 and 16.11 is thanks to Simon Wendsche, and the physically accurate smoke data sets were created by Duc Nguyen and Ron Fedkiw.

The head model used to illustrate subsurface scattering was made available by Infinite Realities, Inc. under a Creative Commons Attribution 3.0 license. Thanks to "Wig42" for the breakfast table scene used in Figure 16.8 and "guismo" for the coffee splash scene used in Figure 15.5; both were was posted to *blendswap.com* also under a Creative Commons Attribution 3.0 license.

Nolan Goodnight created environment maps with a realistic skylight model, and Paul Debevec provided numerous high dynamic-range environment maps. Thanks also to Bernhard Vogl (*dativ.at/lightprobes/*) for environment maps that we used in numerous figures. Marc Ellens provided spectral data for a variety of light sources, and the spectral RGB measurement data for a variety of displays is courtesy of Tom Lianza at X-Rite.

We are most particularly grateful to Guillermo M. Leal Llaguno of Evolución Visual, *www.evvisual.com*, who modeled and rendered the San Miguel scene that was featured on the cover of the second edition and is still used in numerous figures in the book. We would also especially like to thank Marko Dabrovic (*www.3lhd.com*) and Mihovil Odak at RNA Studios (*www.rna.hr*), who supplied a bounty of excellent models and scenes, including the Sponza atrium, the Sibenik cathedral, and the Audi TT car model. Many thanks are also due to Florent Boyer (*www.florentboyer.com*), who provided the contemporary house scene used in some of the images in Chapter 16.

ABOUT THE COVER

The "Countryside" scene on the cover of the book was created by Jan-Walter Schliep, Burak Kahraman, and Timm Dapper of Laubwerk (*www.laubwerk.com*). The scene features 23,241 individual plants, with a total of 3.1 billion triangles. (Thanks to object instancing, only 24 million triangles need to be stored in memory.) The pbrt files that describe the scene geometry require 1.1 GB of on-disk storage. There are a total of 192 texture maps, representing 528 MB of texture data. The scene is one of the example scenes that are available from the pbrt Web site.

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 programs we know of that have been published as books are 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), and Martin Ruckert's book on the *mp3* audio format, *Understanding MP3* (Ruckert 2005).

01 INTRODUCTION

Rendering is the process of producing an image from the description of a 3D scene. Obviously, this is a very broad task, and there are many ways to approach it. *Physically based* techniques attempt to simulate reality; that is, they use principles of physics to model the interaction of light and matter. While a physically based approach may seem to be the most obvious way to approach rendering, it has only been widely adopted in practice over the past 10 or so years. Section 1.7 at the end of this chapter gives a brief history of physically based rendering and its recent adoption for offline rendering for movies and for interactive rendering for games.

This book describes `pbrt`, a physically based rendering system based on the ray-tracing algorithm. Most computer graphics books present algorithms and theory, sometimes combined with snippets of code. In contrast, this book couples the theory with a complete implementation of a fully functional rendering system. The source code to the system (as well as example scenes and a collection of data for rendering) can be found on the `pbrt` Web site, *pbrt.org*.

1.1 LITERATE PROGRAMMING

While writing the TEX typesetting system, Donald Knuth developed a new programming methodology based on the simple but revolutionary idea that *programs should be written more for people's consumption than for computers' consumption*. He named this methodology *literate programming*. This book (including the chapter you're reading now) is a long literate program. This means that in the course of reading this book, you will read the *full* implementation of the `pbrt` rendering system, not just a high-level description of it.

Literate programs are written in a metalanguage that mixes a document formatting language (e.g., TEX or HTML) and a programming language (e.g., C++). Two separate systems process the program: a "weaver" that transforms the literate program into a

Physically Based Rendering: From Theory To Implementation.
http://dx.doi.org/10.1016/B978-0-12-800645-0.50001-4

document suitable for typesetting and a "tangler" that produces source code suitable for compilation. Our literate programming system is homegrown, but it was heavily influenced by Norman Ramsey's `noweb` system.

The literate programming metalanguage provides two important features. The first is the ability to mix prose with source code. This feature makes the description of the program just as important as its actual source code, encouraging careful design and documentation. Second, the language provides mechanisms for presenting the program code to the reader in an order that is entirely different from the compiler input. Thus, the program can be described in a logical manner. Each named block of code is called a *fragment*, and each fragment can refer to other fragments by name.

As a simple example, consider a function `InitGlobals()` that is responsible for initializing all of a program's global variables:[1]

```
void InitGlobals() {
    nMarbles = 25.7;
    shoeSize = 13;
    dielectric = true;
}
```

Despite its brevity, this function is hard to understand without any context. Why, for example, can the variable `nMarbles` take on floating-point values? Just looking at the code, one would need to search through the entire program to see where each variable is declared and how it is used in order to understand its purpose and the meanings of its legal values. Although this structuring of the system is fine for a compiler, a human reader would much rather see the initialization code for each variable presented separately, near the code that actually declares and uses the variable.

In a literate program, one can instead write `InitGlobals()` like this:

⟨*Function Definitions*⟩ ≡
```
void InitGlobals() {
    ⟨Initialize Global Variables 2⟩
}
```

This defines a fragment, called ⟨*Function Definitions*⟩, that contains the definition of the `InitGlobals()` function. The `InitGlobals()` function itself refers to another fragment, ⟨*Initialize Global Variables*⟩. Because the initialization fragment has not yet been defined, we don't know anything about this function except that it will probably contain assignments to global variables. This is just the right level of abstraction for now, since no variables have been declared yet. When we introduce the global variable `shoeSize` somewhere later in the program, we can then write

⟨*Initialize Global Variables*⟩ ≡ 2
```
shoeSize = 13;
```

1 The example code in this section is merely illustrative and is not part of `pbrt` itself.

Here we have started to define the contents of ⟨*Initialize Global Variables*⟩. When the literate program is tangled into source code for compilation, the literate programming system will substitute the code `shoeSize = 13;` inside the definition of the `InitGlobals()` function. Later in the text, we may define another global variable, `dielectric`, and we can append its initialization to the fragment:

⟨*Initialize Global Variables*⟩ +≡ 2

```
dielectric = true;
```

The +≡ symbol after the fragment name shows that we have added to a previously defined fragment. When tangled, the result of these three fragments is the code

```
void InitGlobals() {
    shoeSize = 13;
    dielectric = true;
}
```

In this way, we can decompose complex functions into logically distinct parts, making them much easier to understand. For example, we can write a complicated function as a series of fragments:

⟨*Function Definitions*⟩ +≡

```
void complexFunc(int x, int y, double *values) {
    ⟨Check validity of arguments⟩
    if (x < y) {
        ⟨Swap parameter values⟩
    }
    ⟨Do precomputation before loop⟩
    ⟨Loop through and update values array⟩
}
```

Again, the contents of each fragment are expanded inline in `complexFunc()` for compilation. In the document, we can introduce each fragment and its implementation in turn. This decomposition lets us present code a few lines at a time, making it easier to understand. Another advantage of this style of programming is that by separating the function into logical fragments, each with a single and well-delineated purpose, each one can then be written, verified, or read independently. In general, we will try to make each fragment less than 10 lines long.

In some sense, the literate programming system is just an enhanced macro substitution package tuned to the task of rearranging program source code. This may seem like a trivial change, but in fact literate programming is quite different from other ways of structuring software systems.

1.1.1 INDEXING AND CROSS-REFERENCING

The following features are designed to make the text easier to navigate. Indices in the page margins give page numbers where the functions, variables, and methods used on that page are defined. Indices at the end of the book collect all of these identifiers so that it's possible to find definitions by name. Appendix C, "Index of Fragments," lists the pages where each fragment is defined and the pages where it is used. Within the text, a defined

fragment name is followed by a list of page numbers on which that fragment is used. For example, a hypothetical fragment definition such as

⟨*A fascinating fragment*⟩ ≡ **184, 690**

```
nMarbles += .001;
```

indicates that this fragment is used on pages 184 and 690. Occasionally we elide fragments from the printed book that are either boilerplate code or substantially the same as other fragments; when these fragments are used, no page numbers will be listed.

When a fragment is used inside another fragment, the page number on which it is first defined appears after the fragment name. For example,

⟨*Do something interesting*⟩+≡ **500**

```
InitializeSomethingInteresting();
⟨Do something else interesting 486⟩
CleanUp();
```

indicates that the ⟨*Do something else interesting*⟩ fragment is defined on page 486. If the definition of the fragment is not included in the book, no page number will be listed.

1.2 PHOTOREALISTIC RENDERING AND THE RAY-TRACING ALGORITHM

The goal of photorealistic rendering is to create an image of a 3D scene that is indistinguishable from a photograph of the same scene. Before we describe the rendering process, it is important to understand that in this context the word *indistinguishable* is imprecise because it involves a human observer, and different observers may perceive the same image differently. Although we will cover a few perceptual issues in this book, accounting for the precise characteristics of a given observer is a very difficult and largely unsolved problem. For the most part, we will be satisfied with an accurate simulation of the physics of light and its interaction with matter, relying on our understanding of display technology to present the best possible image to the viewer.

Almost all photorealistic rendering systems are based on the ray-tracing algorithm. Ray tracing is actually a very simple algorithm; it is based on following the path of a ray of light through a scene as it interacts with and bounces off objects in an environment. Although there are many ways to write a ray tracer, all such systems simulate at least the following objects and phenomena:

- *Cameras:* A camera model determines how and from where the scene is being viewed, including how an image of the scene is recorded on a sensor. Many rendering systems generate viewing rays starting at the camera that are then traced into the scene.
- *Ray–object intersections:* We must be able to tell precisely where a given ray intersects a given geometric object. In addition, we need to determine certain properties of the object at the intersection point, such as a surface normal or its material. Most ray tracers also have some facility for testing the intersection of a ray with multiple objects, typically returning the closest intersection along the ray.

- *Light sources:* Without lighting, there would be little point in rendering a scene. A ray tracer must model the distribution of light throughout the scene, including not only the locations of the lights themselves but also the way in which they distribute their energy throughout space.
- *Visibility:* In order to know whether a given light deposits energy at a point on a surface, we must know whether there is an uninterrupted path from the point to the light source. Fortunately, this question is easy to answer in a ray tracer, since we can just construct the ray from the surface to the light, find the closest ray–object intersection, and compare the intersection distance to the light distance.
- *Surface scattering:* Each object must provide a description of its appearance, including information about how light interacts with the object's surface, as well as the nature of the reradiated (or *scattered*) light. Models for surface scattering are typically parameterized so that they can simulate a variety of appearances.
- *Indirect light transport:* Because light can arrive at a surface after bouncing off or passing through other surfaces, it is usually necessary to trace additional rays originating at the surface to fully capture this effect.
- *Ray propagation:* We need to know what happens to the light traveling along a ray as it passes through space. If we are rendering a scene in a vacuum, light energy remains constant along a ray. Although true vacuums are unusual on Earth, they are a reasonable approximation for many environments. More sophisticated models are available for tracing rays through fog, smoke, the Earth's atmosphere, and so on.

We will briefly discuss each of these simulation tasks in this section. In the next section, we will show `pbrt`'s high-level interface to the underlying simulation components and follow the progress of a single ray through the main rendering loop. We will also present the implementation of a surface scattering model based on Turner Whitted's original ray-tracing algorithm.

1.2.1 CAMERAS

Nearly everyone has used a camera and is familiar with its basic functionality: you indicate your desire to record an image of the world (usually by pressing a button or tapping a screen), and the image is recorded onto a piece of film or an electronic sensor. One of the simplest devices for taking photographs is called the *pinhole camera*. Pinhole cameras consist of a light-tight box with a tiny hole at one end (Figure 1.1). When the hole is uncovered, light enters this hole and falls on a piece of photographic paper that is affixed to the other end of the box. Despite its simplicity, this kind of camera is still used today, frequently for artistic purposes. Very long exposure times are necessary to get enough light on the film to form an image.

Although most cameras are substantially more complex than the pinhole camera, it is a convenient starting point for simulation. The most important function of the camera is to define the portion of the scene that will be recorded onto the film. In Figure 1.1, we can see how connecting the pinhole to the edges of the film creates a double pyramid that extends into the scene. Objects that are not inside this pyramid cannot be imaged onto the film. Because actual cameras image a more complex shape than a pyramid, we will refer to the region of space that can potentially be imaged onto the film as the *viewing volume*.

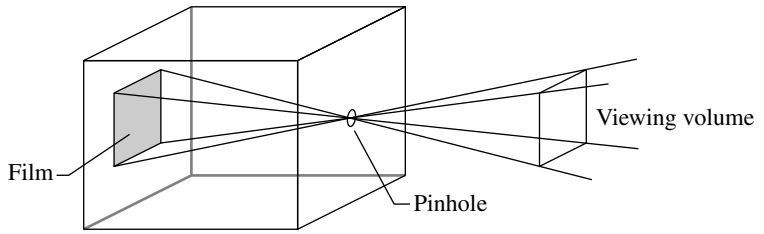

Figure 1.1: A Pinhole Camera.

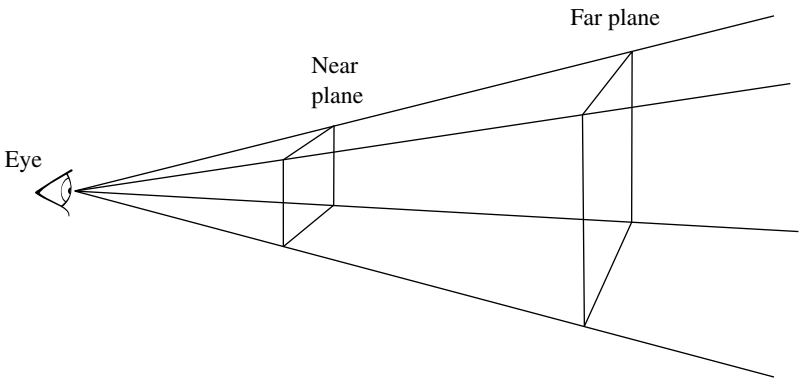

Figure 1.2: When we simulate a pinhole camera, we place the film in front of the hole at the near plane, and the hole is renamed the *eye*.

Another way to think about the pinhole camera is to place the film plane in *front* of the pinhole but at the same distance (Figure 1.2). Note that connecting the hole to the film defines exactly the same viewing volume as before. Of course, this is not a practical way to build a real camera, but for simulation purposes it is a convenient abstraction. When the film (or image) plane is in front of the pinhole, the pinhole is frequently referred to as the *eye*.

Now we come to the crucial issue in rendering: at each point in the image, what color value does the camera record? If we recall the original pinhole camera, it is clear that only light rays that travel along the vector between the pinhole and a point on the film can contribute to that film location. In our simulated camera with the film plane in front of the eye, we are interested in the amount of light traveling from the image point to the eye.

Therefore, an important task of the camera simulator is to take a point on the image and generate *rays* along which incident light will contribute to that image location. Because

a ray consists of an origin point and a direction vector, this task is particularly simple for the pinhole camera model of Figure 1.2: it uses the pinhole for the origin and the vector from the pinhole to the near plane as the ray's direction. For more complex camera models involving multiple lenses, the calculation of the ray that corresponds to a given point on the image may be more involved. (Section 6.4 describes the implementation of such a model.)

With the process of converting image locations to rays completely encapsulated in the camera module, the rest of the rendering system can focus on evaluating the lighting along those rays, and a variety of camera models can be supported. `pbrt`'s camera abstraction is described in detail in Chapter 6.

1.2.2 RAY–OBJECT INTERSECTIONS

Each time the camera generates a ray, the first task of the renderer is to determine which object, if any, that ray intersects first and where the intersection occurs. This intersection point is the visible point along the ray, and we will want to simulate the interaction of light with the object at this point. To find the intersection, we must test the ray for intersection against all objects in the scene and select the one that the ray intersects first. Given a ray r, we first start by writing it in *parametric form*:

$$\mathrm{r}(t) = \mathrm{o} + t\mathbf{d},$$

where o is the ray's origin, $\mathbf{d}$ is its direction vector, and t is a parameter whose legal range is $(0, \infty)$. We can obtain a point along the ray by specifying its parametric t value and evaluating the above equation.

It is often easy to find the intersection between the ray r and a surface defined by an implicit function $F(x, y, z) = 0$. We first substitute the ray equation into the implicit equation, producing a new function whose only parameter is t. We then solve this function for t and substitute the smallest positive root into the ray equation to find the desired point. For example, the implicit equation of a sphere centered at the origin with radius r is

$$x^2 + y^2 + z^2 - r^2 = 0.$$

Substituting the ray equation, we have

$$\left(\mathrm{o}_x + t\mathbf{d}_x\right)^2 + \left(\mathrm{o}_y + t\mathbf{d}_y\right)^2 + \left(\mathrm{o}_z + t\mathbf{d}_z\right)^2 - r^2 = 0.$$

All of the values besides t are known, giving us an easily solved quadratic equation in t. If there are no real roots, the ray misses the sphere; if there are roots, the smallest positive one gives the intersection point.

The intersection point alone is not enough information for the rest of the ray tracer; it needs to know certain properties of the surface at the point. First, a representation of the material at the point must be determined and passed along to later stages of the ray-tracing algorithm. Second, additional geometric information about the intersection point will also be required in order to shade the point. For example, the surface normal $\mathbf{n}$ is always required. Although many ray tracers operate with only $\mathbf{n}$, more sophisticated rendering systems like `pbrt` require even more information, such as various partial

derivatives of position and surface normal with respect to the local parameterization of the surface.

Of course, most scenes are made up of multiple objects. The brute-force approach would be to test the ray against each object in turn, choosing the minimum positive t value of all intersections to find the closest intersection. This approach, while correct, is very slow, even for scenes of modest complexity. A better approach is to incorporate an *acceleration structure* that quickly rejects whole groups of objects during the ray intersection process. This ability to quickly cull irrelevant geometry means that ray tracing frequently runs in $O(I \log N)$ time, where I is the number of pixels in the image and N is the number of objects in the scene.[2] (Building the acceleration structure itself is necessarily at least $O(N)$ time, however.)

`pbrt`'s geometric interface and implementations of it for a variety of shapes is described in Chapter 3, and the acceleration interface and implementations are shown in Chapter 4.

1.2.3 LIGHT DISTRIBUTION

The ray–object intersection stage gives us a point to be shaded and some information about the local geometry at that point. Recall that our eventual goal is to find the amount of light leaving this point in the direction of the camera. In order to do this, we need to know how much light is *arriving* at this point. This involves both the *geometric* and *radiometric* distribution of light in the scene. For very simple light sources (e.g., point lights), the geometric distribution of lighting is a simple matter of knowing the position of the lights. However, point lights do not exist in the real world, and so physically based lighting is often based on *area* light sources. This means that the light source is associated with a geometric object that emits illumination from its surface. However, we will use point lights in this section to illustrate the components of light distribution; rigorous discussion of light measurement and distribution is the topic of Chapters 5 and 12.

We frequently would like to know the amount of light power being deposited on the differential area surrounding the intersection point (Figure 1.3). We will assume that the point light source has some power Φ associated with it and that it radiates light equally in all directions. This means that the power per area on a unit sphere surrounding the light is $\Phi/(4\pi)$. (These measurements will be explained and formalized in Section 5.4.)

If we consider two such spheres (Figure 1.4), it is clear that the power per area at a point on the larger sphere must be less than the power at a point on the smaller sphere because the same total power is distributed over a larger area. Specifically, the power per area arriving at a point on a sphere of radius r is proportional to $1/r^2$. Furthermore, it can be shown that if the tiny surface patch $\mathrm{d}A$ is tilted by an angle θ away from the vector from the surface point to the light, the amount of power deposited on $\mathrm{d}A$ is proportional

2 Although ray tracing's logarithmic complexity is often heralded as one of its key strengths, this complexity is typically only true on average. A number of ray-tracing algorithms that have guaranteed logarithmic running time have been published in the computational geometry literature, but these algorithms only work for certain types of scenes and have very expensive preprocessing and storage requirements. Szirmay-Kalos and Márton provide pointers to the relevant literature (Szirmay-Kalos and Márton 1998). One consolation is that scenes representing realistic environments generally don't exhibit this worst-case behavior. In practice, the ray intersection algorithms presented in this book are sublinear, but without expensive preprocessing and huge memory usage it is always possible to construct worst-case scenes where ray tracing runs in $O(IN)$ time.

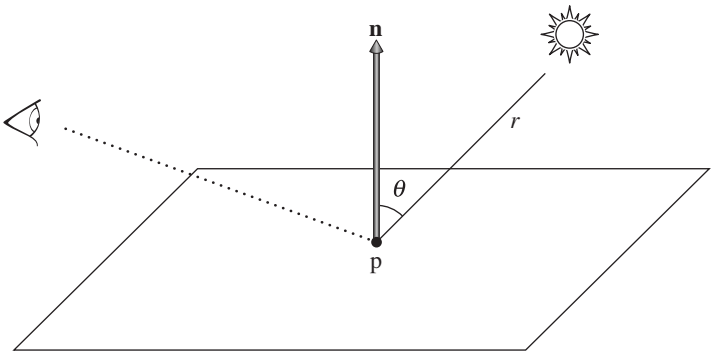

Figure 1.3: Geometric construction for determining the power per area arriving at a point due to a point light source. The distance from the point to the light source is denoted by r.

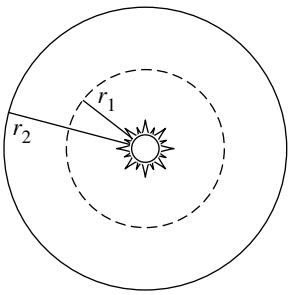

Figure 1.4: Since the point light radiates light equally in all directions, the same total power is deposited on all spheres centered at the light.

to $\cos\theta$. Putting this all together, the differential power per area $\mathrm{d}E$ (the *differential irradiance*) is

$$\mathrm{d}E = \frac{\Phi \cos\theta}{4\pi r^2}.$$

Readers already familiar with basic lighting in computer graphics will notice two familiar laws encoded in this equation: the cosine falloff of light for tilted surfaces mentioned above, and the one-over-r-squared falloff of light with distance.

Scenes with multiple lights are easily handled because illumination is *linear:* the contribution of each light can be computed separately and summed to obtain the overall contribution.

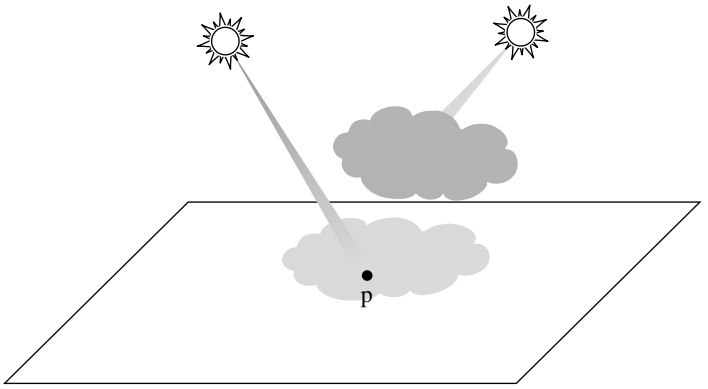

Figure 1.5: A light source only deposits energy on a surface if the source is not obscured as seen from the receiving point. The light source on the left illuminates the point p, but the light source on the right does not.

1.2.4 VISIBILITY

The lighting distribution described in the previous section ignores one very important component: *shadows*. Each light contributes illumination to the point being shaded only if the path from the point to the light's position is unobstructed (Figure 1.5).

Fortunately, in a ray tracer it is easy to determine if the light is visible from the point being shaded. We simply construct a new ray whose origin is at the surface point and whose direction points toward the light. These special rays are called *shadow rays*. If we trace this ray through the environment, we can check to see whether any intersections are found between the ray's origin and the light source by comparing the parametric t value of any intersections found to the parametric t value along the ray of the light source position. If there is no blocking object between the light and the surface, the light's contribution is included.

1.2.5 SURFACE SCATTERING

We now are able to compute two pieces of information that are vital for proper shading of a point: its location and the incident lighting.[3] Now we need to determine how the incident lighting is *scattered* at the surface. Specifically, we are interested in the amount of light energy scattered back along the ray that we originally traced to find the intersection point, since that ray leads to the camera (Figure 1.6).

Each object in the scene provides a *material*, which is a description of its appearance properties at each point on the surface. This description is given by the *bidirectional reflectance distribution function* (BRDF). This function tells us how much energy is reflected

3 Readers already familiar with rendering might object that the discussion in this section considers only direct lighting. Rest assured that `pbrt` does support global illumination.

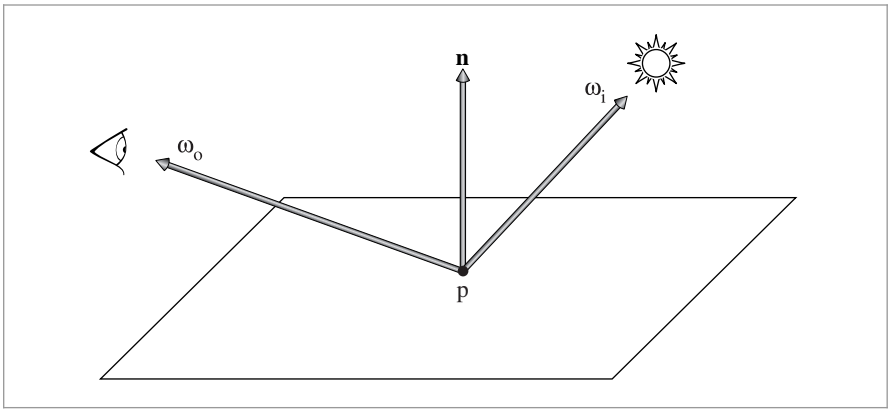

Figure 1.6: The Geometry of Surface Scattering. Incident light arriving along direction ω_i interacts with the surface at point p and is scattered back toward the camera along direction ω_o. The amount of light scattered toward the camera is given by the product of the incident light energy and the BRDF.

from an incoming direction ω_i to an outgoing direction ω_o. We will write the BRDF at p as $f_r(p, \omega_o, \omega_i)$. Now, computing the amount of light L scattered back toward the camera is straightforward:

```
for each light:
    if light is not blocked:
        incident_light = light.L(point)
        amount_reflected =
            surface.BRDF(hit_point, camera_vector, light_vector)
        L += amount_reflected * incident_light
```

Here we are using the symbol L to represent the light; this represents a slightly different unit for light measurement than dE, which was used before. L represents *radiance*, a unit for measuring light that we will see much of in the following.

It is easy to generalize the notion of a BRDF to transmitted light (obtaining a BTDF) or to general scattering of light arriving from either side of the surface. A function that describes general scattering is called a *bidirectional scattering distribution function* (BSDF). `pbrt` supports a variety of BSDF models; they are described in Chapter 8. More complex yet is the *bidirectional subsurface scattering reflectance distribution function* (BSSRDF), which models light that exits a surface at a different point than it enters. The BSSRDF is described in Sections 5.6.2, 11.4, and 15.5.

1.2.6 INDIRECT LIGHT TRANSPORT

Turner Whitted's original paper on ray tracing (1980) emphasized its *recursive* nature, which was the key that made it possible to include indirect specular reflection and transmission in rendered images. For example, if a ray from the camera hits a shiny object like a mirror, we can reflect the ray about the surface normal at the intersection point and recursively invoke the ray-tracing routine to find the light arriving at the point on the

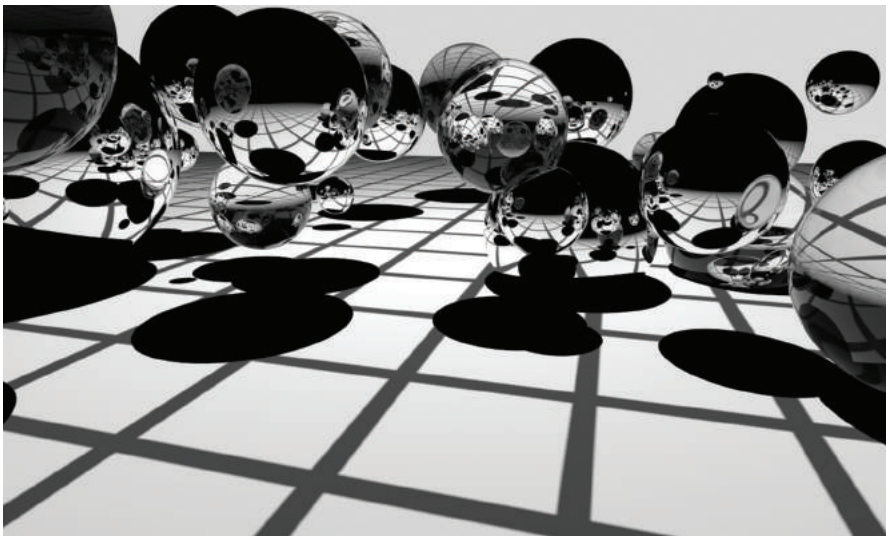

Figure 1.7: A Prototypical Example of Early Ray Tracing. Note the use of mirrored and glass objects, which emphasizes the algorithm's ability to handle these kinds of surfaces.

mirror, adding its contribution to the original camera ray. This same technique can be used to trace transmitted rays that intersect transparent objects. For a long time, most early ray-tracing examples showcased mirrors and glass balls (Figure 1.7) because these types of effects were difficult to capture with other rendering techniques.

In general, the amount of light that reaches the camera from a point on an object is given by the sum of light emitted by the object (if it is itself a light source) and the amount of reflected light. This idea is formalized by the *light transport equation* (also often known as the *rendering equation*), which says that the outgoing radiance $L_o(p, \omega_o)$ from a point p in direction ω_o is the emitted radiance at that point in that direction, $L_e(p, \omega_o)$, plus the incident radiance from all directions on the sphere $\mathbb{S}^2$ around p scaled by the BSDF $f(p, \omega_o, \omega_i)$ and a cosine term:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{\mathbb{S}^2} f(p, \omega_o, \omega_i)\, L_i(p, \omega_i)\, |\cos \theta_i|\, d\omega_i. \quad [1.1]$$

We will show a more complete derivation of this equation in Sections 5.6.1 and 14.4. Solving this integral analytically is not possible except for the simplest of scenes, so we must either make simplifying assumptions or use numerical integration techniques.

Whitted's algorithm simplifies this integral by ignoring incoming light from most directions and only evaluating $L_i(p, \omega_i)$ for directions to light sources and for the directions of perfect reflection and refraction. In other words, it turns the integral into a sum over a small number of directions.

Whitted's method can be extended to capture more effects than just perfect mirrors and glass. For example, by tracing many recursive rays near the mirror-reflection direction and averaging their contributions, we obtain an approximation of glossy reflection. In

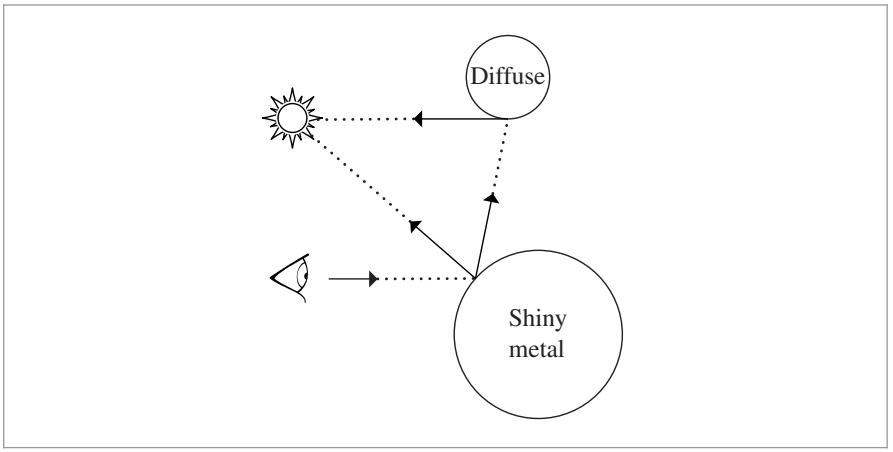

Figure 1.8: Recursive ray tracing associates an entire tree of rays with each image location.

fact, we can *always* recursively trace a ray whenever we hit an object. For example, we can randomly choose a reflection direction ω_i and weight the contribution of this newly spawned ray by evaluating the BRDF $f_r(p, \omega_o, \omega_i)$. This simple but powerful idea can lead to very realistic images because it captures all of the interreflection of light between objects. Of course, we need to know when to terminate the recursion, and choosing directions completely at random may make the rendering algorithm slow to converge to a reasonable result. These problems can be addressed, however; these issues are the topics of Chapters 13 through 16.

When we trace rays recursively in this manner, we are really associating a *tree* of rays with each image location (Figure 1.8), with the ray from the camera at the root of this tree. Each ray in this tree can have a *weight* associated with it; this allows us to model, for example, shiny surfaces that do not reflect 100% of the incoming light.

1.2.7 RAY PROPAGATION

The discussion so far has assumed that rays are traveling through a vacuum. For example, when describing the distribution of light from a point source, we assumed that the light's power was distributed equally on the surface of a sphere centered at the light without decreasing along the way. The presence of *participating media* such as smoke, fog, or dust can invalidate this assumption. These effects are important to simulate: even if we are not making a rendering of a smoke-filled room, almost all outdoor scenes are affected substantially by participating media. For example, Earth's atmosphere causes objects that are farther away to appear less saturated (Figure 1.9).

There are two ways in which a participating medium can affect the light propagating along a ray. First, the medium can *extinguish* (or *attenuate*) light, either by absorbing it or by scattering it in a different direction. We can capture this effect by computing the *transmittance* T between the ray origin and the intersection point. The transmittance