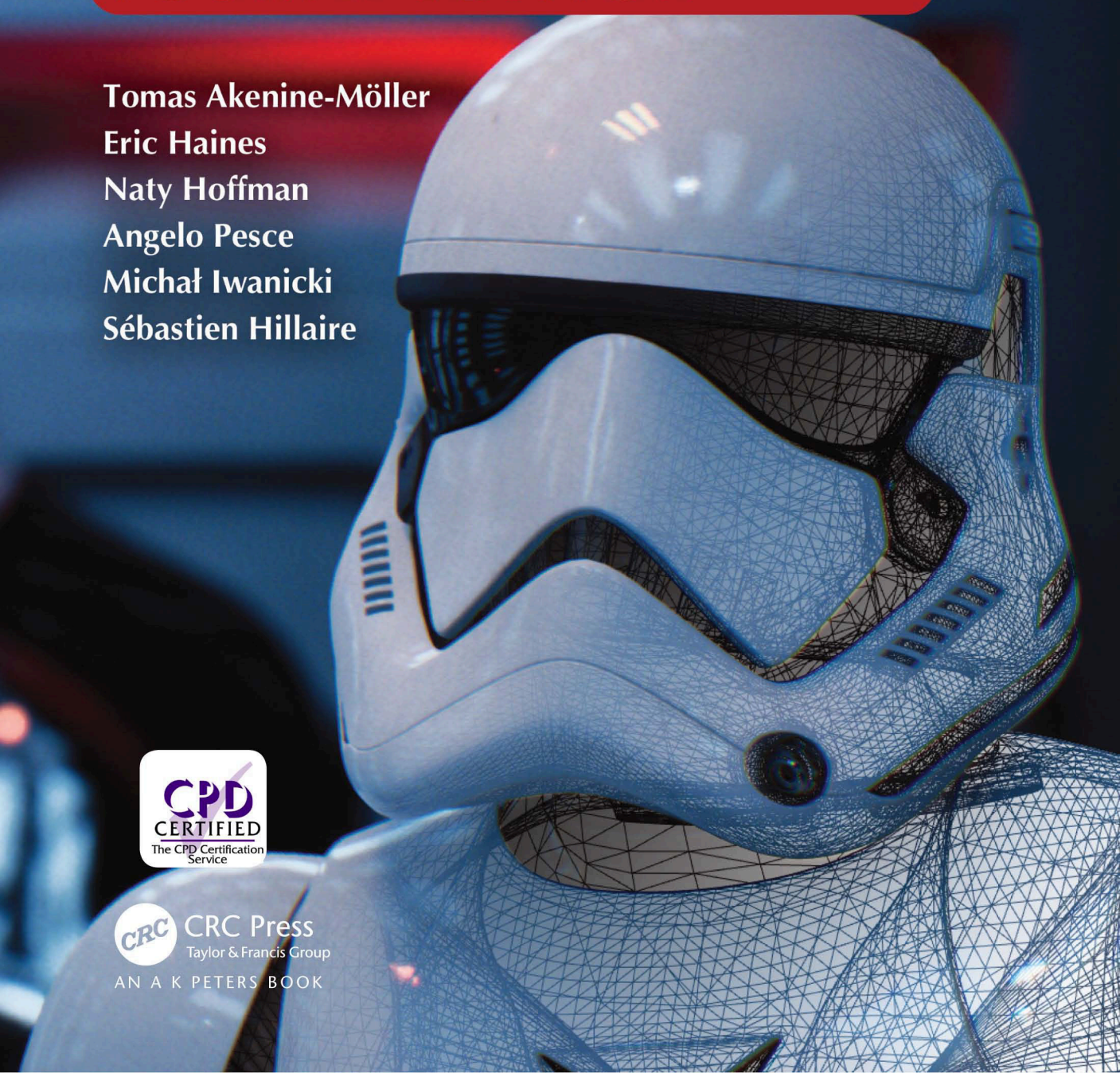
REAL-TIME RENDERING

FOURTH EDITION

Tomas Akenine-Möller
Eric Haines
Naty Hoffman
Angelo Pesce
Michał Iwanicki
Sébastien Hillaire



CPD CERTIFIED
The CPD Certification Service

CRC Press
Taylor & Francis Group
AN A K PETERS BOOK

Real-Time Rendering
Fourth Edition

Real-Time Rendering
Fourth Edition

Tomas Akenine-Möller
Eric Haines
Naty Hoffman
Angelo Pesce
Michał Iwanicki
Sébastien Hillaire

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
AN A K PETERS BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed on acid-free paper

International Standard Book Number-13: 978-1-1386-2700-0 (Hardback)

Library of Congress Cataloging-in-Publication Data

Names: Möller, Tomas, 1971- author.
Title: Real-time rendering / Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michał Iwanicki, Sébastien Hillaire
Description: Fourth edition. | Boca Raton : Taylor & Francis, CRC Press, 2018.
Identifiers: LCCN 2018009546 | ISBN 9781138627000 (hardback : alk. paper)
Subjects: LCSH: Computer graphics. | Real-time data processing. | Rendering (Computer graphics)
Classification: LCC T385 .M635 2018 | DDC 006.6/773--dc23
LC record available at https://lccn.loc.gov/2018009546

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

Dedicated to Eva, Felix, and Elina
T. A-M.

Dedicated to Cathy, Ryan, and Evan
E. H.

Dedicated to Dorit, Karen, and Daniel
N. H.

Dedicated to Fei, Clelia, and Alberto
A. P.

Dedicated to Aneta and Weronika
M. I.

Dedicated to Stéphanie and Svea
S. H.

Contents

Preface

"Things have not changed *that* much in the past eight years," was our thought entering into this fourth edition. "How hard could it be to update the book?" A year and a half later, and with three more experts recruited, our task is done. We could probably spend another year editing and elaborating, at which time there would be easily a hundred more articles and presentations to fold in. As a data point, we made a Google Doc of references that is more than 170 pages long, with about 20 references and related notes on each page. Some references we cite could and do each take up a full section in some other book. A few of our chapters, such as that on shadows, have entire books dedicated to their subjects. While creating more work for us, this wealth of information is good news for practitioners. We will often point to these primary sources, as they offer much more detail than appropriate here.

This book is about algorithms that create synthetic images fast enough that the viewer can interact with a virtual environment. We have focused on three-dimensional rendering and, to a limited extent, on the mechanics of user interaction. Modeling, animation, and many other areas are important to the process of making a real-time application, but these topics are beyond the scope of this book.

We expect you to have some basic understanding of computer graphics before reading this book, as well as knowledge of computer science and programming. We also focus on algorithms, not APIs. Many texts are available on these other subjects. If some section does lose you, skim on through or look at the references. We believe that the most valuable service we can provide you is a realization of what you yet do not know about—a basic kernel of an idea, a sense of what others have discovered about it, and ways to learn more, if you wish.

We make a point of referencing relevant material as possible, as well as providing a summary of further reading and resources at the end of most chapters. In prior editions we cited nearly everything we felt had relevant information. Here we are more a guidebook than an encyclopedia, as the field has far outgrown exhaustive (and exhausting) lists of all possible variations of a given technique. We believe you are better served by describing only a few representative schemes of many, by replacing original sources with newer, broader overviews, and by relying on you, the reader, to pursue more information from the references cited.

Most of these sources are but a mouse click away; see realtimerendering.com for the list of links to references in the bibliography. Even if you have only a passing interest in a topic, consider taking a little time to look at the related references, if for nothing else than to see some of the fantastic images presented. Our website also

contains links to resources, tutorials, demonstration programs, code samples, software libraries, book corrections, and more.

Our true goal and guiding light while writing this book was simple. We wanted to write a book that we wished we had owned when we had started out, a book that both was unified yet also included details and references not found in introductory texts. We hope that you will find this book, our view of the world, of use in your travels.

Acknowledgments for the Fourth Edition

We are not experts in everything, by any stretch of the imagination, nor perfect writers. Many, many people's responses and reviews improved this edition immeasurably, saving us from our own ignorance or inattention. As but one example, when we asked around for advice on what to cover in the area of virtual reality, Johannes Van Waveren (who did not know any of us) instantly responded with a wonderfully detailed outline of topics, which formed the basis for that chapter. These kind acts by computer graphics professionals were some of the great pleasures in writing this book. One person is of particular note: Patrick Cozzi did a yeoman's job, reviewing every chapter in the book. We are grateful to the many people who helped us along the way with this edition. We could write a sentence or three about everyone who helped us along the way, but this would push us further past our book-breaking page limit.

To all the rest, in our hearts we give our appreciation and thanks to you: Sebastian Aaltonen, Johan Andersson, Magnus Andersson, Ulf Assarsson, Dan Baker, Chad Barb, Rasmus Barringer, Michal Bastien, Louis Bavoil, Michael Beale, Adrian Bentley, Ashwin Bhat, Antoine Bouthors, Wade Brainerd, Waylon Brinck, Ryan Brucks, Eric Bruneton, Valentin de Bruyn, Ben Burbank, Brent Burley, Ignacio Castaño, Cem Cebenoyan, Mark Cerny, Matthaeus Chajdas, Danny Chan, Rob Cook, Jean-Luc Corenthin, Adrian Courrèges, Cyril Crassin, Zhihao Cui, Kuba Cupisz, Robert Cupisz, Michal Drobot, Wolfgang Engel, Eugene d'Eon, Matej Drame, Michal Drobot, Alex Evans, Cass Everitt, Kayvon Fatahalian, Adam Finkelstein, Kurt Fleischer, Tim Foley, Tom Forsyth, Guillaume François, Daniel Girardeau-Montaut, Olga Gocmen, Marcin Gollent, Ben Golus, Carlos Gonzalez-Ochoa, Judah Graham, Simon Green, Dirk Gregorius, Larry Gritz, Andrew Hamilton, Earl Hammon, Jr., Jon Harada, Jon Hasselgren, Aaron Hertzmann, Stephen Hill, Rama Hoetzlein, Nicolas Holzschuch, Liwen Hu, John "Spike" Hughes, Ben Humberston, Warren Hunt, Andrew Hurley, John Hutchinson, Milan Ikits, Jon Jansen, Jorge Jimenez, Anton Kaplanyan, Gökhan Karadayi, Brian Karis, Nicolas Kasyan, Alexander Keller, Brano Kemen, Emmett Kilgariff, Byumjin Kim, Chris King, Joe Michael Kniss, Manuel Kraemer, Anders Wang Kristensen, Christopher Kulla, Edan Kwan, Chris Landreth, David Larsson, Andrew Lauritzen, Aaron Lefohn, Eric Lengyel, David Li, Ulrik Lindahl, Edward Liu, Ignacio Llamas, Dulce Isis Segarra López, David Luebke, Patrick Lundell, Miles Macklin, Dzmitry Malyshau, Sam Martin, Morgan McGuire, Brian McIntyre, James McLaren, Mariano Merchante, Arne Meyer, Sergiy Migdalskiy, Kenny Mitchell, Gregory Mitrano, Adam Moravanszky, Jacob Munkberg, Kensaku Nakata, Srinivasa G. Narasimhan, David Neubelt, Fabrice Neyret, Jane Ng, Kasper Høy Nielsen, Matthias

Nießner, Jim Nilsson, Reza Nourai, Chris Oat, Ola Olsson, Rafael Orozco, Bryan Pardilla, Steve Parker, Ankit Patel, Jasmin Patry, Jan Pechenik, Emil Persson, Marc Petit, Matt Pettineo, Agnieszka Piechnik, Jerome Platteaux, Aras Pranckevičius, Elinor Quittner, Silvia Rasheva, Nathaniel Reed, Philip Rideout, Jon Rocatis, Robert Runesson, Marco Salvi, Nicolas Savva, Andrew Schneider, Michael Schneider, Markus Schuetz, Jeremy Selan, Tarek Sherif, Peter Shirley, Peter Sikachev, Peter-Pike Sloan, Ashley Vaughan Smith, Rys Sommefeldt, Edvard Sørgård, Tiago Sousa, Tomasz Stachowiak, Nick Stam, Lee Stemkoski, Jonathan Stone, Kier Storey, Jacob Ström, Filip Strugar, Pierre Terdiman, Aaron Thibault, Nicolas Thibieroz, Robert Toth, Thatcher Ulrich, Mauricio Vives, Alex Vlachos, Evan Wallace, Ian Webster, Nick Whiting, Brandon Whitley, Mattias Widmark, Graham Wihlidal, Michael Wimmer, Daniel Wright, Bart Wroński, Chris Wyman, Ke Xu, Cem Yuksel, and Egor Yusov. We thank you for your time and effort, selflessly offered and gratefully received.

Finally, we want to thank the people at Taylor & Francis for all their efforts, in particular Rick Adams, for getting us going and guiding us along the way, Jessica Vega and Michele Dimont, for their efficient editorial work, and Charlotte Byrnes, for her superb copyediting.

Tomas Akenine-Möller
Eric Haines
Naty Hoffman
Angelo Pesce
Michał Iwanicki
Sébastien Hillaire
February 2018

Acknowledgments for the Third Edition

Special thanks go out to a number of people who went out of their way to provide us with help. First, our graphics architecture case studies would not have been anywhere as good without the extensive and generous cooperation we received from the companies making the hardware. Many thanks to Edvard Sørgard, Borgar Ljosland, Dave Shreiner, and Jørn Nystad at ARM for providing details about their Mali 200 architecture. Thanks also to Michael Dougherty at Microsoft, who provided extremely valuable help with the Xbox 360 section. Masaaki Oka at Sony Computer Entertainment provided his own technical review of the PLAYSTATION® 3 system case study, while also serving as the liaison with the Cell Broadband Engine™ and RSX® developers for their reviews.

In answering a seemingly endless stream of questions, fact-checking numerous passages, and providing many screenshots, Natalya Tatarchuk of ATI/AMD went well beyond the call of duty in helping us out. In addition to responding to our usual requests for information and clarification, Wolfgang Engel was extremely helpful in providing us with articles from the upcoming *ShaderX*^{6} book and copies of the difficult-to-

obtain $ShaderX^2$ books [427, 428], now available online for free. Ignacio Castaño at NVIDIA provided us with valuable support and contacts, going so far as to rework a refractory demo so we could get just the right screenshot.

The chapter reviewers provided an invaluable service to us. They suggested numerous improvements and provided additional insights, helping us immeasurably. In alphabetical order they are: Michael Ashikhmin, Dan Baker, Willem de Boer, Ben Diamand, Ben Discoe, Amir Ebrahimi, Christer Ericson, Michael Gleicher, Manny Ko, Wallace Lages, Thomas Larsson, Grégory Massal, Ville Miettinen, Mike Ramsey, Scott Schaefer, Vincent Scheib, Peter Shirley, K.R. Subramanian, Mauricio Vives, and Hector Yee.

We also had a number of reviewers help us on specific sections. Our thanks go out to Matt Bronder, Christine DeNezza, Frank Fox, Jon Hasselgren, Pete Isensee, Andrew Lauritzen, Morgan McGuire, Jacob Munkberg, Manuel M. Oliveira, Aurelio Reis, Peter-Pike Sloan, Jim Tilander, and Scott Whitman.

We particularly thank Rex Crowle, Kareem Ettouney, and Francis Pang from Media Molecule for their considerable help in providing fantastic imagery and layout concepts for the cover design.

Many people helped us out in other ways, such as answering questions and providing screenshots. Many gave significant amounts of time and effort, for which we thank you. Listed alphabetically: Paulo Abreu, Timo Aila, Johan Andersson, Andreas Bærentzen, Louis Bavoil, Jim Blinn, Jaime Borasi, Per Christensen, Patrick Conran, Rob Cook, Erwin Coumans, Leo Cubbin, Richard Daniels, Mark DeLoura, Tony DeRose, Andreas Dietrich, Michael Dougherty, Bryan Dudash, Alex Evans, Cass Everitt, Randy Fernando, Jim Ferwerda, Chris Ford, Tom Forsyth, Sam Glassenberg, Robin Green, Ned Greene, Larry Gritz, Joakim Grundwall, Mark Harris, Ted Himlan, Jack Hoxley, John "Spike" Hughes, Ladislav Kavan, Alicia Kim, Gary King, Chris Lambert, Jeff Lander, Daniel Leaver, Eric Lengyel, Jennifer Liu, Brandon Lloyd, Charles Loop, David Luebke, Jonathan Maïm, Jason Mitchell, Martin Mittring, Nathan Monteleone, Gabe Newell, Hubert Nguyen, Petri Nordlund, Mike Pan, Ivan Pedersen, Matt Pharr, Fabio Policarpo, Aras Pranckevičius, Siobhan Reddy, Dirk Reiners, Christof Rezk-Salama, Eric Risser, Marcus Roth, Holly Rushmeier, Elan Ruskin, Marco Salvi, Daniel Scherzer, Kyle Shubel, Philipp Slusallek, Torbjörn Söderman, Tim Sweeney, Ben Trumbore, Michal Valient, Mark Valledor, Carsten Wenzel, Steve Westin, Chris Wyman, Cem Yuksel, Billy Zelsnack, Fan Zhang, and Renaldas Zioma.

We also thank many others who responded to our queries on public forums such as GD Algorithms. Readers who took the time to send us corrections have also been a great help. It is this supportive attitude that is one of the pleasures of working in this field.

As we have come to expect, the cheerful competence of the people at A K Peters made the publishing part of the process much easier. For this wonderful support, we thank you all.

On a personal note, Tomas would like to thank his son Felix and daughter Elina for making him understand (again) just how fun it can be to play computer games (on the Wii), instead of just looking at the graphics, and needless to say, his beautiful wife Eva...

Eric would also like to thank his sons Ryan and Evan for their tireless efforts in finding cool game demos and screenshots, and his wife Cathy for helping him survive it all.

Naty would like to thank his daughter Karen and son Daniel for their forbearance when writing took precedence over piggyback rides, and his wife Dorit for her constant encouragement and support.

Tomas Akenine-Möller
Eric Haines
Naty Hoffman
March 2008

Acknowledgments for the Second Edition

One of the most agreeable aspects of writing this second edition has been working with people and receiving their help. Despite their own pressing deadlines and concerns, many people gave us significant amounts of their time to improve this book. We would particularly like to thank the major reviewers. They are, listed alphabetically: Michael Abrash, Ian Ashdown, Ulf Assarsson, Chris Brennan, Sébastien Dominé, David Eberly, Cass Everitt, Tommy Fortes, Evan Hart, Greg James, Jan Kautz, Alexander Keller, Mark Kilgard, Adam Lake, Paul Lalonde, Thomas Larsson, Dean Macri, Carl Marshall, Jason L. Mitchell, Kasper Høy Nielsen, Jon Paul Schelter, Jacob Ström, Nick Triantos, Joe Warren, Michael Wimmer, and Peter Wonka. Of these, we wish to single out Cass Everitt at NVIDIA and Jason L. Mitchell at ATI Technologies for spending large amounts of time and effort in getting us the resources we needed. Our thanks also go out to Wolfgang Engel for freely sharing the contents of his upcoming book, *ShaderX* [426], so that we could make this edition as current as possible.

From discussing their work with us, to providing images or other resources, to writing reviews of sections of the book, many others helped in creating this edition. They all have our gratitude. These people include: Jason Ang, Haim Barad, Jules Bloomenthal, Jonathan Blow, Chas. Boyd, John Brooks, Cem Cebenoyan, Per Christensen, Hamilton Chu, Michael Cohen, Daniel Cohen-Or, Matt Craighead, Paul Debevec, Joe Demers, Walt Donovan, Howard Dortch, Mark Duchaineau, Phil Dutré, Dave Eberle, Gerald Farin, Simon Fenney, Randy Fernando, Jim Ferwerda, Nickson Fong, Tom Forsyth, Piero Foscari, Laura Fryer, Markus Giegl, Peter Glaskowsky, Andrew Glassner, Amy Gooch, Bruce Gooch, Simon Green, Ned Greene, Larry Gritz, Joakim Grundwall, Juan Guardado, Pat Hanrahan, Mark Harris, Michael Herf, Carsten Hess, Rich Hilmer, Kenneth Hoff III, Naty Hoffman, Nick Holliman, Hugues Hoppe, Heather Horne, Tom Hubina, Richard Huddy, Adam James, Kaveh Kardan, Paul Keller, David

Kirk, Alex Klimovitski, Jason Knipe, Jeff Lander, Marc Levoy, J.P. Lewis, Ming Lin, Adrian Lopez, Michael McCool, Doug McNabb, Stan Melax, Ville Miettinen, Kenny Mitchell, Steve Morein, Henry Moreton, Jerris Mungai, Jim Napier, George Ngo, Hubert Nguyen, Tito Pagán, Jörg Peters, Tom Porter, Emil Praun, Kekoa Proudfoot, Bernd Raabe, Ravi Ramamoorthi, Ashutosh Rege, Szymon Rusinkiewicz, Chris Seitz, Carlo Séquin, Jonathan Shade, Brian Smits, John Spitzer, Wolfgang Straßer, Wolfgang Stürzlinger, Philip Taylor, Pierre Terdiman, Nicolas Thibieroz, Jack Tumblin, Fredrik Ulfves, Thatcher Ulrich, Steve Upstill, Alex Vlachos, Ingo Wald, Ben Watson, Steve Westin, Dan Wexler, Matthias Wloka, Peter Woytiuk, David Wu, Garrett Young, Borut Zalik, Harold Zatz, Hansong Zhang, and Denis Zorin. We also wish to thank the journal *ACM Transactions on Graphics* for providing a mirror website for this book.

Alice and Klaus Peters, our production manager Ariel Jaffee, our editor Heather Holcombe, our copyeditor Michelle M. Richards, and the rest of the staff at A K Peters have done a wonderful job making this book the best possible. Our thanks to all of you.

Finally, and most importantly, our deepest thanks go to our families for giving us the huge amounts of quiet time we have needed to complete this edition. Honestly, we never thought it would take this long!

Tomas Akenine-Möller
Eric Haines
May 2002

Acknowledgments for the First Edition

Many people helped in making this book. Some of the greatest contributions were made by those who reviewed parts of it. The reviewers willingly gave the benefit of their expertise, helping to significantly improve both content and style. We wish to thank (in alphabetical order) Thomas Barregren, Michael Cohen, Walt Donovan, Angus Dorbie, Michael Garland, Stefan Gottschalk, Ned Greene, Ming C. Lin, Jason L. Mitchell, Liang Peng, Keith Rule, Ken Shoemake, John Stone, Phil Taylor, Ben Trumbore, Jorrit Tyberghein, and Nick Wilt. We cannot thank you enough.

Many other people contributed their time and labor to this project. Some let us use images, others provided models, still others pointed out important resources or connected us with people who could help. In addition to the people listed above, we wish to acknowledge the help of Tony Barkans, Daniel Baum, Nelson Beebe, Curtis Beeson, Tor Berg, David Blythe, Chas. Boyd, Don Brittain, Ian Bullard, Javier Castellar, Satyan Coorg, Jason Della Rocca, Paul Diefenbach, Alyssa Donovan, Dave Eberly, Kells Elmquist, Stuart Feldman, Fred Fisher, Tom Forsyth, Marty Franz, Thomas Funkhouser, Andrew Glassner, Bruce Gooch, Larry Gritz, Robert Grzeszczuk, Paul Haeberli, Evan Hart, Paul Heckbert, Chris Hecker, Joachim Helenklaken, Hugues Hoppe, John Jack, Mark Kilgard, David Kirk, James Klosowski, Subodh Kumar, André LaMothe, Jeff Lander, Jens Larsson, Jed Lengyel, Fredrik Liliegren, David Luebke, Thomas Lundqvist, Tom McReynolds, Stan Melax, Don Mitchell, André Möller,

Steve Molnar, Scott R. Nelson, Hubert Nguyen, Doug Rogers, Holly Rushmeier, Gernot Schaufler, Jonas Skeppstedt, Stephen Spencer, Per Stenström, Jacob Ström, Filippo Tampieri, Gary Tarolli, Ken Turkowski, Turner Whitted, Agata and Andrzej Wojaczek, Andrew Woo, Steve Worley, Brian Yen, Hans-Philip Zachau, Gabriel Zachmann, and Al Zimmerman. We also wish to thank the journal *ACM Transactions on Graphics* for providing a stable website for this book.

Alice and Klaus Peters and the staff at AK Peters, particularly Carolyn Artin and Sarah Gillis, have been instrumental in making this book a reality. To all of you, thanks.

Finally, our deepest thanks go to our families and friends for providing support throughout this incredible, sometimes grueling, often exhilarating process.

Tomas Möller
Eric Haines
March 1999

Chapter 1
Introduction

Real-time rendering is concerned with rapidly making images on the computer. It is the most highly interactive area of computer graphics. An image appears on the screen, the viewer acts or reacts, and this feedback affects what is generated next. This cycle of reaction and rendering happens at a rapid enough rate that the viewer does not see individual images, but rather becomes immersed in a dynamic process.

The rate at which images are displayed is measured in frames per second (FPS) or Hertz (Hz). At one frame per second, there is little sense of interactivity; the user is painfully aware of the arrival of each new image. At around 6 FPS, a sense of interactivity starts to grow. Video games aim for 30, 60, 72, or higher FPS; at these speeds the user focuses on action and reaction.

Movie projectors show frames at 24 FPS but use a shutter system to display each frame two to four times to avoid flicker. This *refresh rate* is separate from the display rate and is expressed in Hertz (Hz). A shutter that illuminates the frame three times has a 72 Hz refresh rate. LCD monitors also separate refresh rate from display rate.

Watching images appear on a screen at 24 FPS might be acceptable, but a higher rate is important for minimizing response time. As little as 15 milliseconds of temporal delay can slow and interfere with interaction [1849]. As an example, head-mounted displays for virtual reality often require 90 FPS to minimize latency.

There is more to real-time rendering than interactivity. If speed was the only criterion, any application that rapidly responded to user commands and drew anything on the screen would qualify. Rendering in real time normally means producing three-dimensional images.

Interactivity and some sense of connection to three-dimensional space are sufficient conditions for real-time rendering, but a third element has become a part of its definition: graphics acceleration hardware. Many consider the introduction of the 3Dfx Voodoo 1 card in 1996 the real beginning of consumer-level three-dimensional graphics [408]. With the rapid advances in this market, every computer, tablet, and mobile phone now comes with a graphics processor built in. Some excellent examples of the results of real-time rendering made possible by hardware acceleration are shown in Figures 1.1 and 1.2.

Figure 1.1. A shot from *Forza Motorsport 7. (Image courtesy of Turn 10 Studios, Microsoft.)*

Figure 1.2. The city of Beauclair rendered in *The Witcher 3. (CD PROJEKT®, The Witcher® are registered trademarks of CD PROJEKT Capital Group. The Witcher game© CD PROJEKT S.A. Developed by CD PROJEKT S.A. All rights reserved. The Witcher game is based on the prose of Andrzej Sapkowski. All other copyrights and trademarks are the property of their respective owners.)*

Advances in graphics hardware have fueled an explosion of research in the field of interactive computer graphics. We will focus on providing methods to increase speed and improve image quality, while also describing the features and limitations of acceleration algorithms and graphics APIs. We will not be able to cover every topic in depth, so our goal is to present key concepts and terminology, explain the most robust and practical algorithms in the field, and provide pointers to the best places to go for more information. We hope our attempts to provide you with tools for understanding this field prove to be worth the time and effort you spend with our book.

1.1 Contents Overview

What follows is a brief overview of the chapters ahead.

Chapter 2, The Graphics Rendering Pipeline. The heart of real-time rendering is the set of steps that takes a scene description and converts it into something we can see.

Chapter 3, The Graphics Processing Unit. The modern GPU implements the stages of the rendering pipeline using a combination of fixed-function and programmable units.

Chapter 4, Transforms. Transforms are the basic tools for manipulating the position, orientation, size, and shape of objects and the location and view of the camera.

Chapter 5, Shading Basics. Discussion begins on the definition of materials and lights and their use in achieving the desired surface appearance, whether realistic or stylized. Other appearance-related topics are introduced, such as providing higher image quality through the use of antialiasing, transparency, and gamma correction.

Chapter 6, Texturing. One of the most powerful tools for real-time rendering is the ability to rapidly access and display images on surfaces. This process is called texturing, and there are a wide variety of methods for applying it.

Chapter 7, Shadows. Adding shadows to a scene increases both realism and comprehension. The more popular algorithms for computing shadows rapidly are presented.

Chapter 8, Light and Color. Before we perform physically based rendering, we first need to understand how to quantify light and color. And after our physical rendering process is done, we need to transform the resulting quantities into values for the display, accounting for the properties of the screen and viewing environment. Both topics are covered in this chapter.

Chapter 9, Physically Based Shading. We build an understanding of physically based shading models from the ground up. The chapter starts with the underlying physical phenomena, covers models for a variety of rendered materials, and ends with methods for blending materials together and filtering them to avoid aliasing and preserve surface appearance.

Chapter 10, Local Illumination. Algorithms for portraying more elaborate light sources are explored. Surface shading takes into account that light is emitted by physical objects, which have characteristic shapes.

Chapter 11, Global Illumination. Algorithms that simulate multiple interactions between the light and the scene further increase the realism of an image. We discuss ambient and directional occlusion and methods for rendering global illumination effects on diffuse and specular surfaces, as well as some promising unified approaches.

Chapter 12, Image-Space Effects. Graphics hardware is adept at performing image processing at rapid speeds. Image filtering and reprojection techniques are discussed

first, then we survey several popular post-processing effects: lens flares, motion blur, and depth of field.

Chapter 13, Beyond Polygons. Triangles are not always the fastest or most realistic way to describe objects. Alternate representations based on using images, point clouds, voxels, and other sets of samples each have their advantages.

Chapter 14, Volumetric and Translucency Rendering. The focus here is the theory and practice of volumetric material representations and their interactions with light sources. The simulated phenomena range from large-scale atmospheric effects down to light scattering within thin hair fibers.

Chapter 15, Non-Photorealistic Rendering. Attempting to make a scene look realistic is only one way of rendering it. Other styles, such as cartoon shading and watercolor effects, are surveyed. Line and text generation techniques are also discussed.

Chapter 16, Polygonal Techniques. Geometric data comes from a wide range of sources, and sometimes requires modification to be rendered rapidly and well. The many facets of polygonal data representation and compression are presented.

Chapter 17, Curves and Curved Surfaces. More complex surface representations offer advantages such as being able to trade off between quality and rendering speed, more compact representation, and smooth surface generation.

Chapter 18, Pipeline Optimization. Once an application is running and uses efficient algorithms, it can be made even faster using various optimization techniques. Finding the bottleneck and deciding what to do about it is the theme here. Multiprocessing is also discussed.

Chapter 19, Acceleration Algorithms. After you make it go, make it go fast. Various forms of culling and level of detail rendering are covered.

Chapter 20, Efficient Shading. A large number of lights in a scene can slow performance considerably. Fully shading surface fragments before they are known to be visible is another source of wasted cycles. We explore a wide range of approaches to tackle these and other forms of inefficiency while shading.

Chapter 21, Virtual and Augmented Reality. These fields have particular challenges and techniques for efficiently producing realistic images at rapid and consistent rates.

Chapter 22, Intersection Test Methods. Intersection testing is important for rendering, user interaction, and collision detection. In-depth coverage is provided here for a wide range of the most efficient algorithms for common geometric intersection tests.

Chapter 23, Graphics Hardware. The focus here is on components such as color depth, framebuffers, and basic architecture types. A case study of representative GPUs is provided.

Chapter 24, The Future. Take a guess (we do).

Due to space constraints, we have made a chapter about Collision Detection free for download at realtimerendering.com, along with appendices on linear algebra and trigonometry.

1.2 Notation and Definitions

First, we shall explain the mathematical notation used in this book. For a more thorough explanation of many of the terms used in this section, and throughout this book, get our linear algebra appendix at realtimerendering.com.

1.2.1 Mathematical Notation

Table 1.1 summarizes most of the mathematical notation we will use. Some of the concepts will be described at some length here.

Note that there are some exceptions to the rules in the table, primarily shading equations using notation that is extremely well established in the literature, e.g., L for radiance, E for irradiance, and σ_s for scattering coefficient.

The angles and the scalars are taken from $\mathbb{R}$, i.e., they are real numbers. Vectors and points are denoted by bold lowercase letters, and the components are accessed as

$$\mathbf{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix},$$

that is, in column vector format, which is commonly used in the computer graphics world. At some places in the text we use (v_x, v_y, v_z) instead of the formally more correct $(v_x \quad v_y \quad v_z)^T$, since the former is easier to read.

Type	**Notation**	**Examples**
angle	lowercase Greek	$\alpha_i, \phi, \rho, \eta, \gamma_{242}, \theta$
scalar	lowercase italic	a, b, t, u_k, v, w_{ij}
vector or point	lowercase bold	$\mathbf{a}$,$\mathbf{u}$, $\mathbf{v}_s$ $\mathbf{h}(\rho)$, $\mathbf{h}_z$
matrix	capital bold	$\mathbf{T}(\mathbf{t})$, $\mathbf{X}$, $\mathbf{R}_x(\rho)$
plane	π: a vector and a scalar	$\pi : \mathbf{n} \cdot \mathbf{x} + d = 0$, $\pi_1 : \mathbf{n}_1 \cdot \mathbf{x} + d_1 = 0$
triangle	$\triangle$ 3 points	$\triangle \mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$, $\triangle \mathbf{cba}$
line segment	two points	$\mathbf{uv}$, $\mathbf{a}_i\mathbf{b}_j$
geometric entity	capital italic	A_{OBB}, T, B_{AABB}

Table 1.1. Summary of the notation used in this book.

Using homogeneous notation, a coordinate is represented by four values $\mathbf{v} = (v_x \quad v_y \quad v_z \quad v_w)^T$, where a vector is $\mathbf{v} = (v_x \quad v_y \quad v_z \quad 0)^T$ and a point is $\mathbf{v} = (v_x \quad v_y \quad v_z \quad 1)^T$. Sometimes we use only three-element vectors and points, but we try to avoid any ambiguity as to which type is being used. For matrix manipulations, it is extremely advantageous to have the same notation for vectors as for points. For more information, see Chapter 4 on transforms. In some algorithms, it will be convenient to use numeric indices instead of x, y, and z, for example $\mathbf{v} = (v_0 \quad v_1 \quad v_2)^T$. All these rules for vectors and points also hold for two-element vectors; in that case, we simply skip the last component of a three-element vector.

The matrix deserves a bit more explanation. The common sizes that will be used are 2×2, 3×3, and 4×4. We will review the manner of accessing a 3×3 matrix $\mathbf{M}$, and it is simple to extend this process to the other sizes. The (scalar) elements of $\mathbf{M}$ are denoted m_{ij}, $0 \leq (i, j) \leq 2$, where i denotes the row and j the column, as in Equation 1.1:

$$\mathbf{M} = \begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix}. \tag{1.1}$$

The following notation, shown in Equation 1.2 for a 3×3 matrix, is used to isolate vectors from the matrix $\mathbf{M}$: $\mathbf{m}_{,j}$ represents the jth column vector and $\mathbf{m}_{i,}$ represents the ith row vector (in column vector form). As with vectors and points, indexing the column vectors can also be done with x, y, z, and sometimes w, if that is more convenient:

$$\mathbf{M} = \begin{pmatrix} \mathbf{m}_{,0} & \mathbf{m}_{,1} & \mathbf{m}_{,2} \end{pmatrix} = \begin{pmatrix} \mathbf{m}_x & \mathbf{m}_y & \mathbf{m}_z \end{pmatrix} = \begin{pmatrix} \mathbf{m}_{0,}^T \\ \mathbf{m}_{1,}^T \\ \mathbf{m}_{2,}^T \end{pmatrix}. \tag{1.2}$$

A plane is denoted $\pi : \mathbf{n} \cdot \mathbf{x} + d = 0$ and contains its mathematical formula, the plane normal $\mathbf{n}$ and the scalar d. The normal is a vector describing what direction the plane faces. More generally (e.g., for curved surfaces), a normal describes this direction for a particular point on the surface. For a plane the same normal happens to apply to all its points. π is the common mathematical notation for a plane. The plane π is said to divide the space into a *positive half-space*, where $\mathbf{n} \cdot \mathbf{x} + d > 0$, and a *negative half-space*, where $\mathbf{n} \cdot \mathbf{x} + d < 0$. All other points are said to lie in the plane.

A triangle can be defined by three points $\mathbf{v}_0$, $\mathbf{v}_1$, and $\mathbf{v}_2$ and is denoted by $\triangle \mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$.

Table 1.2 presents some additional mathematical operators and their notation. The dot, cross, determinant, and length operators are explained in our downloadable linear algebra appendix at realtimerendering.com. The transpose operator turns a column vector into a row vector and vice versa. Thus a column vector can be written in compressed form in a block of text as $\mathbf{v} = (v_x \quad v_y \quad v_z)^T$. Operator 4, introduced in *Graphics Gems IV* [735], is a unary operator on a two-dimensional vector. Letting

	Operator	Description
1:	$\cdot$	dot product
2:	$\times$	cross product
3:	$\mathbf{v}^T$	transpose of the vector $\mathbf{v}$
4:	$^\perp$	the unary, perp dot product operator
5:	$\lvert \cdot \rvert$	determinant of a matrix
6:	$\lvert \cdot \rvert$	absolute value of a scalar
7:	$\lVert \cdot \rVert$	length (or norm) of argument
8:	x^+	clamping x to 0
9:	$x^{\overline{+}}$	clamping x between 0 and 1
10:	$n!$	factorial
11:	$\binom{n}{k}$	binomial coefficients

Table 1.2. Notation for some mathematical operators.

this operator work on a vector $\mathbf{v} = (v_x \;\; v_y)^T$ gives a vector that is perpendicular to $\mathbf{v}$, i.e., $\mathbf{v}^\perp = (-v_y \;\; v_x)^T$. We use $|a|$ to denote the absolute value of the scalar a, while $|\mathbf{A}|$ means the determinant of the matrix $\mathbf{A}$. Sometimes, we also use $|\mathbf{A}| = |\mathbf{a} \;\; \mathbf{b} \;\; \mathbf{c}| = \det(\mathbf{a}, \mathbf{b}, \mathbf{c})$, where $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ are column vectors of the matrix $\mathbf{A}$.

Operators 8 and 9 are clamping operators, commonly used in shading calculations. Operator 8 clamps negative values to 0:

$$x^+ = \begin{cases} x, & \text{if } x > 0, \\ 0, & \text{otherwise,} \end{cases} \tag{1.3}$$

and operator 9 clamps values between 0 and 1:

$$x^{\overline{+}} = \begin{cases} 1, & \text{if } x \geq 1, \\ x, & \text{if } 0 < x < 1, \\ 0, & \text{otherwise.} \end{cases} \tag{1.4}$$

The tenth operator, factorial, is defined as shown below, and note that $0! = 1$:

$$n! = n(n-1)(n-2)\cdots 3 \cdot 2 \cdot 1. \tag{1.5}$$

The eleventh operator, the binomial factor, is defined as shown in Equation 1.6:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}. \tag{1.6}$$

	Function	Description
1:	atan2(y, x)	two-value arctangent
2:	$\log(n)$	natural logarithm of n

Table 1.3. Notation for some specialized mathematical functions.

Further on, we call the common planes $x = 0$, $y = 0$, and $z = 0$ the *coordinate planes* or *axis-aligned planes*. The axes $\mathbf{e}_x = (1 \quad 0 \quad 0)^T$, $\mathbf{e}_y = (0 \quad 1 \quad 0)^T$, and $\mathbf{e}_z = (0 \quad 0 \quad 1)^T$ are called *main axes* or *main directions* and individually called the x-axis, y-axis, and z-axis. This set of axes is often called the *standard basis*. Unless otherwise noted, we will use orthonormal bases (consisting of mutually perpendicular unit vectors).

The notation for a range that includes both a and b, and all numbers in between, is $[a, b]$. If we want all number between a and b, but not a and b themselves, then we write (a, b). Combinations of these can also be made, e.g., $[a, b)$ means all numbers between a and b including a but not b.

The C-math function `atan2(y,x)` is often used in this text, and so deserves some attention. It is an extension of the mathematical function $\arctan(x)$. The main differences between them are that $-\frac{\pi}{2} < \arctan(x) < \frac{\pi}{2}$, that $0 \leq \texttt{atan2(y,x)} < 2\pi$, and that an extra argument has been added to the latter function. A common use for arctan is to compute $\arctan(y/x)$, but when $x = 0$, division by zero results. The extra argument for `atan2(y,x)` avoids this.

In this volume the notation $\log(n)$ always means the natural logarithm, $\log_e(n)$, not the base-10 logarithm, $\log_{10}(n)$.

We use a right-hand coordinate system since this is the standard system for three-dimensional geometry in the field of computer graphics.

Colors are represented by a three-element vector, such as (*red, green, blue*), where each element has the range $[0, 1]$.

1.2.2 Geometrical Definitions

The basic rendering primitives (also called *drawing primitives*) used by almost all graphics hardware are points, lines, and triangles.[1]

Throughout this book, we will refer to a collection of geometric entities as either a *model* or an *object*. A *scene* is a collection of models comprising everything that is included in the environment to be rendered. A scene can also include material descriptions, lighting, and viewing specifications.

Examples of objects are a car, a building, and even a line. In practice, an object often consists of a set of drawing primitives, but this may not always be the case; an object may have a higher kind of geometrical representation, such as Bézier curves or

[1] The only exceptions we know of are Pixel-Planes [502], which could draw spheres, and the NVIDIA NV1 chip, which could draw ellipsoids.

surfaces, or subdivision surfaces. Also, objects can consist of other objects, e.g., a car object includes four door objects, four wheel objects, and so on.

1.2.3 Shading

Following well-established computer graphics usage, in this book terms derived from "shading," "shader," and related words are used to refer to two distinct but related concepts: computer-generated visual appearance (e.g., "shading model," "shading equation," "toon shading") or a programmable component of a rendering system (e.g., "vertex shader," "shading language"). In both cases, the intended meaning should be clear from the context.

Further Reading and Resources

The most important resource we can refer you to is the website for this book: realtimerendering.com. It contains links to the latest information and websites relevant to each chapter. The field of real-time rendering is changing with real-time speed. In the book we have attempted to focus on concepts that are fundamental and techniques that are unlikely to go out of style. On the website we have the opportunity to present information that is relevant to today's software developer, and we have the ability to keep it up-to-date.

Chapter 2

The Graphics Rendering Pipeline

"A chain is no stronger than its weakest link."
—Anonymous

This chapter presents the core component of real-time graphics, namely the *graphics rendering pipeline*, also known simply as "the pipeline." The main function of the pipeline is to generate, or *render*, a two-dimensional image, given a virtual camera, three-dimensional objects, light sources, and more. The rendering pipeline is thus the underlying tool for real-time rendering. The process of using the pipeline is depicted in Figure 2.1. The locations and shapes of the objects in the image are determined by their geometry, the characteristics of the environment, and the placement of the camera in that environment. The appearance of the objects is affected by material properties, light sources, textures (images applied to surfaces), and shading equations.

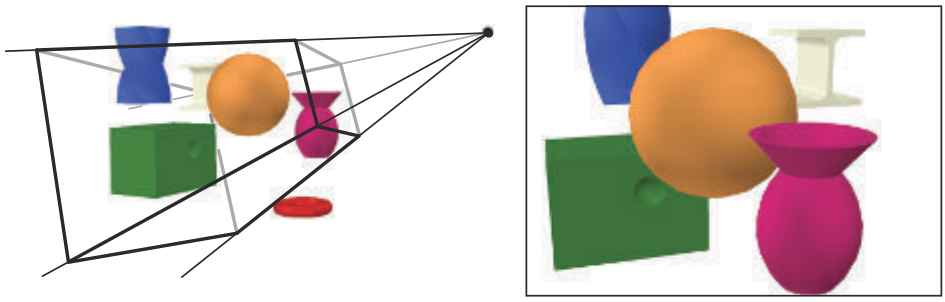


Figure 2.1. In the left image, a virtual camera is located at the tip of the pyramid (where four lines converge). Only the primitives inside the view volume are rendered. For an image that is rendered in perspective (as is the case here), the view volume is a *frustum* (plural: *frusta*), i.e., a truncated pyramid with a rectangular base. The right image shows what the camera "sees." Note that the red donut shape in the left image is not in the rendering to the right because it is located outside the view frustum. Also, the twisted blue prism in the left image is clipped against the top plane of the frustum.

We will explain the different stages of the rendering pipeline, with a focus on function rather than implementation. Relevant details for applying these stages will be covered in later chapters.

2.1 Architecture

In the physical world, the pipeline concept manifests itself in many different forms, from factory assembly lines to fast food kitchens. It also applies to graphics rendering. A pipeline consists of several stages [715], each of which performs part of a larger task.

The pipeline stages execute in parallel, with each stage dependent upon the result of the previous stage. Ideally, a nonpipelined system that is then divided into n pipelined stages could give a speedup of a factor of n. This increase in performance is the main reason to use pipelining. For example, a large number of sandwiches can be prepared quickly by a series of people—one preparing the bread, another adding meat, another adding toppings. Each passes the result to the next person in line and immediately starts work on the next sandwich. If each person takes twenty seconds to perform their task, a maximum rate of one sandwich every twenty seconds, three a minute, is possible. The pipeline stages execute in parallel, but they are stalled until the slowest stage has finished its task. For example, say the meat addition stage becomes more involved, taking thirty seconds. Now the best rate that can be achieved is two sandwiches a minute. For this particular pipeline, the meat stage is the *bottleneck*, since it determines the speed of the entire production. The toppings stage is said to be *starved* (and the customer, too) during the time it waits for the meat stage to be done.

This kind of pipeline construction is also found in the context of real-time computer graphics. A coarse division of the real-time rendering pipeline into four main stages—*application*, *geometry processing*, *rasterization*, and *pixel processing*—is shown in Figure 2.2. This structure is the core—the engine of the rendering pipeline—which is used in real-time computer graphics applications and is thus an essential base for

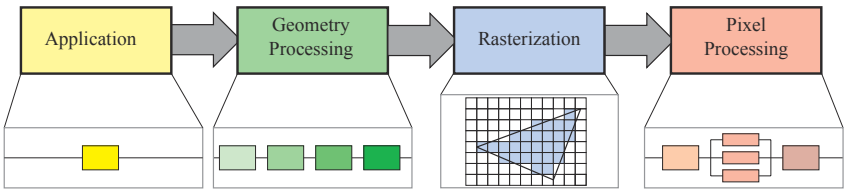


Figure 2.2. The basic construction of the rendering pipeline, consisting of four stages: application, geometry processing, rasterization, and pixel processing. Each of these stages may be a pipeline in itself, as illustrated below the geometry processing stage, or a stage may be (partly) parallelized, as shown below the pixel processing stage. In this illustration, the application stage is a single process, but this stage could also be pipelined or parallelized. Note that rasterization finds the pixels inside a primitive, e.g., a triangle.

discussion in subsequent chapters. Each of these stages is usually a pipeline in itself, which means that it consists of several substages. We differentiate between the functional stages shown here and the structure of their implementation. A functional stage has a certain task to perform but does not specify the way that task is executed in the pipeline. A given implementation may combine two functional stages into one unit or execute using programmable cores, while it divides another, more time-consuming, functional stage into several hardware units.

The rendering speed may be expressed in *frames per second* (FPS), that is, the number of images rendered per second. It can also be represented using *Hertz* (Hz), which is simply the notation for $1/seconds$, i.e., the frequency of update. It is also common to just state the time, in milliseconds (ms), that it takes to render an image. The time to generate an image usually varies, depending on the complexity of the computations performed during each frame. Frames per second is used to express either the rate for a particular frame, or the average performance over some duration of use. Hertz is used for hardware, such as a display, which is set to a fixed rate.

As the name implies, the *application* stage is driven by the application and is therefore typically implemented in software running on general-purpose CPUs. These CPUs commonly include multiple cores that are capable of processing multiple *threads of execution* in parallel. This enables the CPUs to efficiently run the large variety of tasks that are the responsibility of the application stage. Some of the tasks traditionally performed on the CPU include collision detection, global acceleration algorithms, animation, physics simulation, and many others, depending on the type of application. The next main stage is *geometry processing*, which deals with transforms, projections, and all other types of geometry handling. This stage computes what is to be drawn, how it should be drawn, and where it should be drawn. The geometry stage is typically performed on a graphics processing unit (GPU) that contains many programmable cores as well as fixed-operation hardware. The *rasterization* stage typically takes as input three vertices, forming a triangle, and finds all pixels that are considered inside that triangle, then forwards these to the next stage. Finally, the *pixel processing* stage executes a program per pixel to determine its color and may perform depth testing to see whether it is visible or not. It may also perform per-pixel operations such as blending the newly computed color with a previous color. The rasterization and pixel processing stages are also processed entirely on the GPU. All these stages and their internal pipelines will be discussed in the next four sections. More details on how the GPU processes these stages are given in Chapter 3.

2.2 The Application Stage

The developer has full control over what happens in the application stage, since it usually executes on the CPU. Therefore, the developer can entirely determine the implementation and can later modify it in order to improve performance. Changes here can also affect the performance of subsequent stages. For example, an application

stage algorithm or setting could decrease the number of triangles to be rendered.

All this said, some application work can be performed by the GPU, using a separate mode called a *compute shader*. This mode treats the GPU as a highly parallel general processor, ignoring its special functionality meant specifically for rendering graphics.

At the end of the application stage, the geometry to be rendered is fed to the geometry processing stage. These are the *rendering primitives*, i.e., points, lines, and triangles, that might eventually end up on the screen (or whatever output device is being used). This is the most important task of the application stage.

A consequence of the software-based implementation of this stage is that it is not divided into substages, as are the geometry processing, rasterization, and pixel processing stages.[1] However, to increase performance, this stage is often executed in parallel on several processor cores. In CPU design, this is called a *superscalar* construction, since it is able to execute several processes at the same time in the same stage. Section 18.5 presents various methods for using multiple processor cores.

One process commonly implemented in this stage is *collision detection*. After a collision is detected between two objects, a response may be generated and sent back to the colliding objects, as well as to a force feedback device. The application stage is also the place to take care of input from other sources, such as the keyboard, the mouse, or a head-mounted display. Depending on this input, several different kinds of actions may be taken. Acceleration algorithms, such as particular culling algorithms (Chapter 19), are also implemented here, along with whatever else the rest of the pipeline cannot handle.

2.3 Geometry Processing

The geometry processing stage on the GPU is responsible for most of the per-triangle and per-vertex operations. This stage is further divided into the following functional stages: vertex shading, projection, clipping, and screen mapping (Figure 2.3).

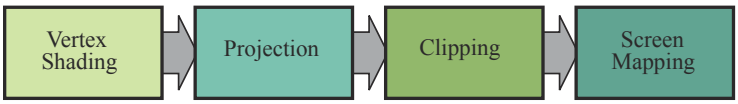


Figure 2.3. The geometry processing stage divided into a pipeline of functional stages.

[1] Since a CPU itself is pipelined on a much smaller scale, you could say that the application stage is further subdivided into several pipeline stages, but this is not relevant here.

2.3.1 Vertex Shading

There are two main tasks of vertex shading, namely, to compute the position for a vertex and to evaluate whatever the programmer may like to have as vertex output data, such as a normal and texture coordinates. Traditionally much of the shade of an object was computed by applying lights to each vertex's location and normal and storing only the resulting color at the vertex. These colors were then interpolated across the triangle. For this reason, this programmable vertex processing unit was named the vertex shader [1049]. With the advent of the modern GPU, along with some or all of the shading taking place per pixel, this vertex shading stage is more general and may not evaluate any shading equations at all, depending on the programmer's intent. The vertex shader is now a more general unit dedicated to setting up the data associated with each vertex. As an example, the vertex shader can animate an object using the methods in Sections 4.4 and 4.5.

We start by describing how the vertex position is computed, a set of coordinates that is always required. On its way to the screen, a model is transformed into several different *spaces* or *coordinate systems*. Originally, a model resides in its own *model space*, which simply means that it has not been transformed at all. Each model can be associated with a *model transform* so that it can be positioned and oriented. It is possible to have several model transforms associated with a single model. This allows several copies (called *instances*) of the same model to have different locations, orientations, and sizes in the same scene, without requiring replication of the basic geometry.

It is the vertices and the normals of the model that are transformed by the model transform. The coordinates of an object are called *model coordinates*, and after the model transform has been applied to these coordinates, the model is said to be located in *world coordinates* or in *world space*. The world space is unique, and after the models have been transformed with their respective model transforms, all models exist in this same space.

As mentioned previously, only the models that the camera (or observer) sees are rendered. The camera has a location in world space and a direction, which are used to place and aim the camera. To facilitate projection and clipping, the camera and all the models are transformed with the *view transform*. The purpose of the view transform is to place the camera at the origin and aim it, to make it look in the direction of the negative z-axis, with the y-axis pointing upward and the x-axis pointing to the right. We use the $-z$-axis convention; some texts prefer looking down the $+z$-axis. The difference is mostly semantic, as transform between one and the other is simple. The actual position and direction after the view transform has been applied are dependent on the underlying application programming interface (API). The space thus delineated is called *camera space*, or more commonly, *view space* or *eye space*. An example of the way in which the view transform affects the camera and the models is shown in Figure 2.4. Both the model transform and the view transform may be implemented as 4×4 matrices, which is the topic of Chapter 4. However, it is important to realize that

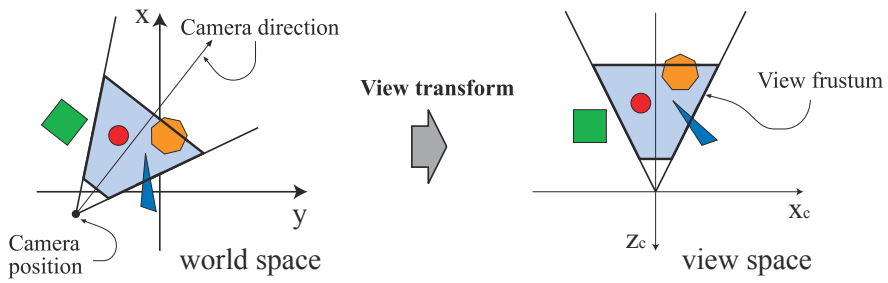


Figure 2.4. In the left illustration, a top-down view shows the camera located and oriented as the user wants it to be, in a world where the $+z$-axis is up. The view transform reorients the world so that the camera is at the origin, looking along its negative z-axis, with the camera's $+y$-axis up, as shown on the right. This is done to make the clipping and projection operations simpler and faster. The light blue area is the view volume. Here, perspective viewing is assumed, since the view volume is a frustum. Similar techniques apply to any kind of projection.

the position and normal of a vertex can be computed in whatever way the programmer prefers.

Next, we describe the second type of output from vertex shading. To produce a realistic scene, it is not sufficient to render the shape and position of objects, but their appearance must be modeled as well. This description includes each object's material, as well as the effect of any light sources shining on the object. Materials and lights can be modeled in any number of ways, from simple colors to elaborate representations of physical descriptions.

This operation of determining the effect of a light on a material is known as *shading*. It involves computing a *shading equation* at various points on the object. Typically, some of these computations are performed during geometry processing on a model's vertices, and others may be performed during per-pixel processing. A variety of material data can be stored at each vertex, such as the point's location, a normal, a color, or any other numerical information that is needed to evaluate the shading equation. Vertex shading results (which can be colors, vectors, texture coordinates, along with any other kind of shading data) are then sent to the rasterization and pixel processing stages to be interpolated and used to compute the shading of the surface.

Vertex shading in the form of the GPU vertex shader is discussed in more depth throughout this book and most specifically in Chapters 3 and 5.

As part of vertex shading, rendering systems perform *projection* and then clipping, which transforms the view volume into a unit cube with its extreme points at $(-1,-1,-1)$ and $(1,1,1)$. Different ranges defining the same volume can and are used, for example, $0 \leq z \leq 1$. The unit cube is called the *canonical view volume*. Projection is done first, and on the GPU it is done by the vertex shader. There are two commonly used projection methods, namely *orthographic* (also called *parallel*)

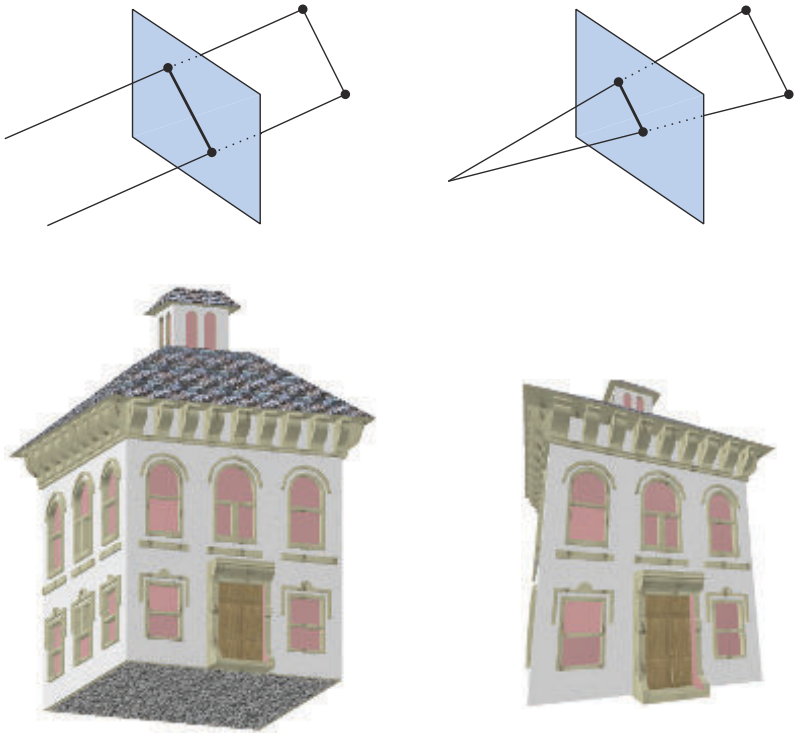


Figure 2.5. On the left is an orthographic, or parallel, projection; on the right is a perspective projection.

and *perspective* projection. See Figure 2.5. In truth, orthographic is just one type of parallel projection. Several others find use, particularly in the field of architecture, such as oblique and axonometric projections. The old arcade game *Zaxxon* is named from the latter.

Note that projection is expressed as a matrix (Section 4.7) and so it may sometimes be concatenated with the rest of the geometry transform.

The view volume of orthographic viewing is normally a rectangular box, and the orthographic projection transforms this view volume into the unit cube. The main characteristic of orthographic projection is that parallel lines remain parallel after the transform. This transformation is a combination of a translation and a scaling.

The perspective projection is a bit more complex. In this type of projection, the farther away an object lies from the camera, the smaller it appears after projection. In addition, parallel lines may converge at the horizon. The perspective transform thus mimics the way we perceive objects' size. Geometrically, the view volume, called a *frustum*, is a truncated pyramid with rectangular base. The frustum is transformed

into the unit cube as well. Both orthographic and perspective transforms can be constructed with 4×4 matrices (Chapter 4), and after either transform, the models are said to be in *clip coordinates*. These are in fact homogeneous coordinates, discussed in Chapter 4, and so this occurs before division by w. The GPU's vertex shader must always output coordinates of this type in order for the next functional stage, clipping, to work correctly.

Although these matrices transform one volume into another, they are called projections because after display, the z-coordinate is not stored in the image generated but is stored in a z-buffer, described in Section 2.5. In this way, the models are projected from three to two dimensions.

2.3.2 Optional Vertex Processing

Every pipeline has the vertex processing just described. Once this processing is done, there are a few optional stages that can take place on the GPU, in this order: tessellation, geometry shading, and stream output. Their use depends both on the capabilities of the hardware—not all GPUs have them—and the desires of the programmer. They are independent of each other, and in general they are not commonly used. More will be said about each in Chapter 3.

The first optional stage is *tessellation*. Imagine you have a bouncing ball object. If you represent it with a single set of triangles, you can run into problems with quality or performance. Your ball may look good from 5 meters away, but up close the individual triangles, especially along the silhouette, become visible. If you make the ball with more triangles to improve quality, you may waste considerable processing time and memory when the ball is far away and covers only a few pixels on the screen. With tessellation, a curved surface can be generated with an appropriate number of triangles.

We have talked a bit about triangles, but up to this point in the pipeline we have just processed vertices. These could be used to represent points, lines, triangles, or other objects. Vertices can be used to describe a curved surface, such as a ball. Such surfaces can be specified by a set of patches, and each patch is made of a set of vertices. The tessellation stage consists of a series of stages itself—hull shader, tessellator, and domain shader—that converts these sets of patch vertices into (normally) larger sets of vertices that are then used to make new sets of triangles. The camera for the scene can be used to determine how many triangles are generated: many when the patch is close, few when it is far away.

The next optional stage is the *geometry shader*. This shader predates the tessellation shader and so is more commonly found on GPUs. It is like the tessellation shader in that it takes in primitives of various sorts and can produce new vertices. It is a much simpler stage in that this creation is limited in scope and the types of output primitives are much more limited. Geometry shaders have several uses, with one of the most popular being particle generation. Imagine simulating a fireworks explosion.

Each fireball could be represented by a point, a single vertex. The geometry shader can take each point and turn it into a square (made of two triangles) that faces the viewer and covers several pixels, so providing a more convincing primitive for us to shade.

The last optional stage is called *stream output*. This stage lets us use the GPU as a geometry engine. Instead of sending our processed vertices down the rest of the pipeline to be rendered to the screen, at this point we can optionally output these to an array for further processing. These data can be used by the CPU, or the GPU itself, in a later pass. This stage is typically used for particle simulations, such as our fireworks example.

These three stages are performed in this order—tessellation, geometry shading, and stream output—and each is optional. Regardless of which (if any) options are used, if we continue down the pipeline we have a set of vertices with homogeneous coordinates that will be checked for whether the camera views them.

2.3.3 Clipping

Only the primitives wholly or partially inside the view volume need to be passed on to the rasterization stage (and the subsequent pixel processing stage), which then draws them on the screen. A primitive that lies fully inside the view volume will be passed on to the next stage as is. Primitives entirely outside the view volume are not passed on further, since they are not rendered. It is the primitives that are partially inside the view volume that require clipping. For example, a line that has one vertex outside and one inside the view volume should be clipped against the view volume, so that the vertex that is outside is replaced by a new vertex that is located at the intersection between the line and the view volume. The use of a projection matrix means that the transformed primitives are clipped against the unit cube. The advantage of performing the view transformation and projection before clipping is that it makes the clipping problem consistent; primitives are always clipped against the unit cube.

The clipping process is depicted in Figure 2.6. In addition to the six clipping planes of the view volume, the user can define additional clipping planes to visibly chop objects. An image showing this type of visualization, called *sectioning*, is shown in Figure 19.1 on page 818.

The clipping step uses the 4-value homogeneous coordinates produced by projection to perform clipping. Values do not normally interpolate linearly across a triangle in perspective space. The fourth coordinate is needed so that data are properly interpolated and clipped when a perspective projection is used. Finally, *perspective division* is performed, which places the resulting triangles' positions into three-dimensional *normalized device coordinates*. As mentioned earlier, this view volume ranges from $(-1, -1, -1)$ to $(1, 1, 1)$. The last step in the geometry stage is to convert from this space to window coordinates.

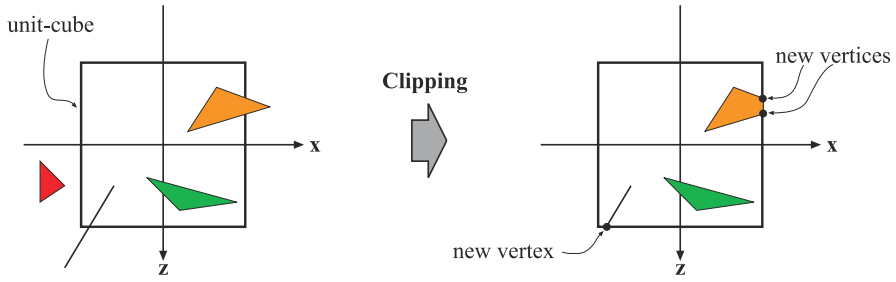


Figure 2.6. After the projection transform, only the primitives inside the unit cube (which correspond to primitives inside the view frustum) are needed for continued processing. Therefore, the primitives outside the unit cube are discarded, and primitives fully inside are kept. Primitives intersecting with the unit cube are clipped against the unit cube, and thus new vertices are generated and old ones are discarded.

2.3.4 Screen Mapping

Only the (clipped) primitives inside the view volume are passed on to the screen mapping stage, and the coordinates are still three-dimensional when entering this stage. The x- and y-coordinates of each primitive are transformed to form *screen coordinates*. Screen coordinates together with the z-coordinates are also called *window coordinates*. Assume that the scene should be rendered into a window with the minimum corner at (x_1, y_1) and the maximum corner at (x_2, y_2), where $x_1 < x_2$ and $y_1 < y_2$. Then the screen mapping is a translation followed by a scaling operation. The new x- and y-coordinates are said to be screen coordinates. The z-coordinate ($[-1, +1]$ for OpenGL and $[0, 1]$ for DirectX) is also mapped to $[z_1, z_2]$, with $z_1 = 0$ and $z_2 = 1$ as the default values. These can be changed with the API, however. The window coordinates along with this remapped z-value are passed on to the rasterizer stage. The screen mapping process is depicted in Figure 2.7.

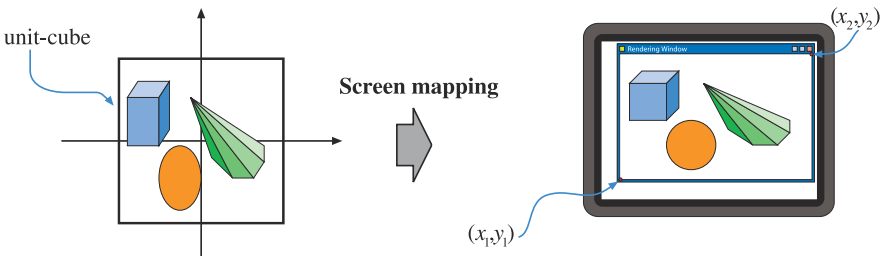


Figure 2.7. The primitives lie in the unit cube after the projection transform, and the screen mapping procedure takes care of finding the coordinates on the screen.

Next, we describe how integer and floating point values relate to pixels (and texture coordinates). Given a horizontal array of pixels and using Cartesian coordinates, the left edge of the leftmost pixel is 0.0 in floating point coordinates. OpenGL has always used this scheme, and DirectX 10 and its successors use it. The center of this pixel is at 0.5. So, a range of pixels $[0, 9]$ cover a span from $[0.0, 10.0)$. The conversions are simply

$$d = \texttt{floor}(\texttt{c}), \tag{2.1}$$

$$c = d + 0.5, \tag{2.2}$$

where d is the discrete (integer) index of the pixel and c is the continuous (floating point) value within the pixel.

While all APIs have pixel location values that increase going from left to right, the location of zero for the top and bottom edges is inconsistent in some cases between OpenGL and DirectX.[2] OpenGL favors the Cartesian system throughout, treating the lower left corner as the lowest-valued element, while DirectX sometimes defines the upper left corner as this element, depending on the context. There is a logic to each, and no right answer exists where they differ. As an example, $(0, 0)$ is located at the lower left corner of an image in OpenGL, while it is upper left for DirectX. This difference is important to take into account when moving from one API to the other.

2.4 Rasterization

Given the transformed and projected vertices with their associated shading data (all from geometry processing), the goal of the next stage is to find all pixels—short for *picture elements*—that are inside the primitive, e.g., a triangle, being rendered. We call this process *rasterization*, and it is split up into two functional substages: triangle setup (also called primitive assembly) and triangle traversal. These are shown to the left in Figure 2.8. Note that these can handle points and lines as well, but since triangles are most common, the substages have "triangle" in their names. Rasterization, also called *scan conversion*, is thus the conversion from two-dimensional vertices in screen space—each with a z-value (depth value) and various shading information associated with each vertex—into pixels on the screen. Rasterization can also be thought of as a synchronization point between geometry processing and pixel processing, since it is here that triangles are formed from three vertices and eventually sent down to pixel processing.

Whether the triangle is considered to overlap the pixel depends on how you have set up the GPU's pipeline. For example, you may use point sampling to determine

[2] "Direct3D" is the three-dimensional graphics API component of DirectX. DirectX includes other API elements, such an input and audio control. Rather than differentiate between writing "DirectX" when specifying a particular release and "Direct3D" when discussing this particular API, we follow common usage by writing "DirectX" throughout.

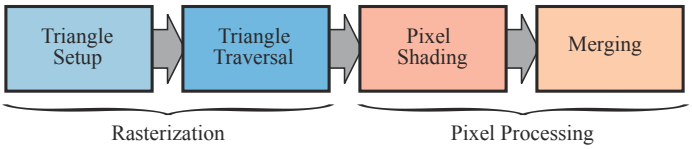


Figure 2.8. Left: rasterization split into two functional stages, called triangle setup and triangle traversal. Right: pixel processing split into two functional stages, namely, pixel processing and merging.

"insideness." The simplest case uses a single point sample in the center of each pixel, and so if that center point is inside the triangle then the corresponding pixel is considered inside the triangle as well. You may also use more than one sample per pixel using supersampling or multisampling antialiasing techniques (Section 5.4.2). Yet another way is to use conservative rasterization, where the definition is that a pixel is "inside" the triangle if at least part of the pixel overlaps with the triangle (Section 23.1.2).

2.4.1 Triangle Setup

In this stage the differentials, edge equations, and other data for the triangle are computed. These data may be used for triangle traversal (Section 2.4.2), as well as for interpolation of the various shading data produced by the geometry stage. Fixed-function hardware is used for this task.

2.4.2 Triangle Traversal

Here is where each pixel that has its center (or a sample) covered by the triangle is checked and a *fragment* generated for the part of the pixel that overlaps the triangle. More elaborate sampling methods can be found in Section 5.4. Finding which samples or pixels are inside a triangle is often called *triangle traversal*. Each triangle fragment's properties are generated using data interpolated among the three triangle vertices (Chapter 5). These properties include the fragment's depth, as well as any shading data from the geometry stage. McCormack et al. [1162] offer more information on triangle traversal. It is also here that perspective-correct interpolation over the triangles is performed [694] (Section 23.1.1). All pixels or samples that are inside a primitive are then sent to the pixel processing stage, described next.

2.5 Pixel Processing

At this point, all the pixels that are considered inside a triangle or other primitive have been found as a consequence of the combination of all the previous stages. The

pixel processing stage is divided into *pixel shading* and *merging*, shown to the right in Figure 2.8. Pixel processing is the stage where per-pixel or per-sample computations and operations are performed on pixels or samples that are inside a primitive.

2.5.1 Pixel Shading

Any per-pixel shading computations are performed here, using the interpolated shading data as input. The end result is one or more colors to be passed on to the next stage. Unlike the triangle setup and traversal stages, which are usually performed by dedicated, hardwired silicon, the pixel shading stage is executed by programmable GPU cores. To that end, the programmer supplies a program for the pixel shader (or fragment shader, as it is known in OpenGL), which can contain any desired computations. A large variety of techniques can be employed here, one of the most important of which is *texturing*. Texturing is treated in more detail in Chapter 6. Simply put, texturing an object means "gluing" one or more images onto that object, for a variety of purposes. A simple example of this process is depicted in Figure 2.9. The image may be one-, two-, or three-dimensional, with two-dimensional images being the most common. At its simplest, the end product is a color value for each fragment, and these are passed on to the next substage.

Figure 2.9. A dragon model without textures is shown in the upper left. The pieces in the image texture are "glued" onto the dragon, and the result is shown in the lower left.

2.5.2 Merging

The information for each pixel is stored in the *color buffer*, which is a rectangular array of colors (a red, a green, and a blue component for each color). It is the responsibility of the merging stage to combine the fragment color produced by the pixel shading stage with the color currently stored in the buffer. This stage is also called ROP, standing for "raster operations (pipeline)" or "render output unit," depending on who you ask. Unlike the shading stage, the GPU subunit that performs this stage is typically not fully programmable. However, it is highly configurable, enabling various effects.

This stage is also responsible for resolving visibility. This means that when the whole scene has been rendered, the color buffer should contain the colors of the primitives in the scene that are visible from the point of view of the camera. For most or even all graphics hardware, this is done with the z-buffer (also called *depth buffer*) algorithm [238]. A z-buffer is the same size and shape as the color buffer, and for each pixel it stores the z-value to the currently closest primitive. This means that when a primitive is being rendered to a certain pixel, the z-value on that primitive at that pixel is being computed and compared to the contents of the z-buffer at the same pixel. If the new z-value is smaller than the z-value in the z-buffer, then the primitive that is being rendered is closer to the camera than the primitive that was previously closest to the camera at that pixel. Therefore, the z-value and the color of that pixel are updated with the z-value and color from the primitive that is being drawn. If the computed z-value is greater than the z-value in the z-buffer, then the color buffer and the z-buffer are left untouched. The z-buffer algorithm is simple, has $O(n)$ convergence (where n is the number of primitives being rendered), and works for any drawing primitive for which a z-value can be computed for each (relevant) pixel. Also note that this algorithm allows most primitives to be rendered in any order, which is another reason for its popularity. However, the z-buffer stores only a single depth at each point on the screen, so it cannot be used for partially transparent primitives. These must be rendered after all opaque primitives, and in back-to-front order, or using a separate order-independent algorithm (Section 5.5). Transparency is one of the major weaknesses of the basic z-buffer.

We have mentioned that the color buffer is used to store colors and that the z-buffer stores z-values for each pixel. However, there are other channels and buffers that can be used to filter and capture fragment information. The *alpha channel* is associated with the color buffer and stores a related opacity value for each pixel (Section 5.5). In older APIs, the alpha channel was also used to discard pixels selectively via the alpha test feature. Nowadays a discard operation can be inserted into the pixel shader program and any type of computation can be used to trigger a discard. This type of test can be used to ensure that fully transparent fragments do not affect the z-buffer (Section 6.6).

The *stencil buffer* is an offscreen buffer used to record the locations of the rendered primitive. It typically contains 8 bits per pixel. Primitives can be rendered into the stencil buffer using various functions, and the buffer's contents can then be used to

control rendering into the color buffer and z-buffer. As an example, assume that a filled circle has been drawn into the stencil buffer. This can be combined with an operator that allows rendering of subsequent primitives into the color buffer only where the circle is present. The stencil buffer can be a powerful tool for generating some special effects. All these functions at the end of the pipeline are called *raster operations* (ROP) or *blend operations*. It is possible to mix the color currently in the color buffer with the color of the pixel being processed inside a triangle. This can enable effects such as transparency or the accumulation of color samples. As mentioned, blending is typically configurable using the API and not fully programmable. However, some APIs have support for raster order views, also called pixel shader ordering, which enable programmable blending capabilities.

The *framebuffer* generally consists of all the buffers on a system.

When the primitives have reached and passed the rasterizer stage, those that are visible from the point of view of the camera are displayed on screen. The screen displays the contents of the color buffer. To avoid allowing the human viewer to see the primitives as they are being rasterized and sent to the screen, *double buffering* is used. This means that the rendering of a scene takes place off screen, in a *back buffer*. Once the scene has been rendered in the back buffer, the contents of the back buffer are swapped with the contents of the *front buffer* that was previously displayed on the screen. The swapping often occurs during *vertical retrace*, a time when it is safe to do so.

For more information on different buffers and buffering methods, see Sections 5.4.2, 23.6, and 23.7.

2.6 Through the Pipeline

Points, lines, and triangles are the rendering primitives from which a model or an object is built. Imagine that the application is an interactive *computer aided design* (CAD) application, and that the user is examining a design for a waffle maker. Here we will follow this model through the entire graphics rendering pipeline, consisting of the four major stages: application, geometry, rasterization, and pixel processing. The scene is rendered with perspective into a window on the screen. In this simple example, the waffle maker model includes both lines (to show the edges of parts) and triangles (to show the surfaces). The waffle maker has a lid that can be opened. Some of the triangles are textured by a two-dimensional image with the manufacturer's logo. For this example, surface shading is computed completely in the geometry stage, except for application of the texture, which occurs in the rasterization stage.

Application

CAD applications allow the user to select and move parts of the model. For example, the user might select the lid and then move the mouse to open it. The application stage must translate the mouse move to a corresponding rotation matrix, then see to

it that this matrix is properly applied to the lid when it is rendered. Another example: An animation is played that moves the camera along a predefined path to show the waffle maker from different views. The camera parameters, such as position and view direction, must then be updated by the application, dependent upon time. For each frame to be rendered, the application stage feeds the camera position, lighting, and primitives of the model to the next major stage in the pipeline—the geometry stage.

Geometry Processing

For perspective viewing, we assume here that the application has supplied a projection matrix. Also, for each object, the application has computed a matrix that describes both the view transform and the location and orientation of the object in itself. In our example, the waffle maker's base would have one matrix, the lid another. In the geometry stage the vertices and normals of the object are transformed with this matrix, putting the object into view space. Then shading or other calculations at the vertices may be computed, using material and light source properties. Projection is then performed using a separate user-supplied projection matrix, transforming the object into a unit cube's space that represents what the eye sees. All primitives outside the cube are discarded. All primitives intersecting this unit cube are clipped against the cube in order to obtain a set of primitives that lies entirely inside the unit cube. The vertices then are mapped into the window on the screen. After all these per-triangle and per-vertex operations have been performed, the resulting data are passed on to the rasterization stage.

Rasterization

All the primitives that survive clipping in the previous stage are then rasterized, which means that all pixels that are inside a primitive are found and sent further down the pipeline to pixel processing.

Pixel Processing

The goal here is to compute the color of each pixel of each visible primitive. Those triangles that have been associated with any textures (images) are rendered with these images applied to them as desired. Visibility is resolved via the z-buffer algorithm, along with optional discard and stencil tests. Each object is processed in turn, and the final image is then displayed on the screen.

Conclusion

This pipeline resulted from decades of API and graphics hardware evolution targeted to real-time rendering applications. It is important to note that this is not the only possible rendering pipeline; offline rendering pipelines have undergone different evolutionary paths. Rendering for film production was often done with *micropolygon* pipelines [289, 1734], but ray tracing and path tracing have taken over lately. These

techniques, covered in Section 11.2.2, may also be used in architectural and design previsualization.

For many years, the only way for application developers to use the process described here was through a *fixed-function pipeline* defined by the graphics API in use. The fixed-function pipeline is so named because the graphics hardware that implements it consists of elements that cannot be programmed in a flexible way. The last example of a major fixed-function machine is Nintendo's Wii, introduced in 2006. Programmable GPUs, on the other hand, make it possible to determine exactly what operations are applied in various sub-stages throughout the pipeline. For the fourth edition of the book, we assume that all development is done using programmable GPUs.

Further Reading and Resources

Blinn's book *A Trip Down the Graphics Pipeline* [165] is an older book about writing a software renderer from scratch. It is a good resource for learning about some of the subtleties of implementing a rendering pipeline, explaining key algorithms such as clipping and perspective interpolation. The venerable (yet frequently updated) *OpenGL Programming Guide* (a.k.a. the "Red Book") [885] provides a thorough description of the graphics pipeline and algorithms related to its use. Our book's website, realtimerendering.com, gives links to a variety of pipeline diagrams, rendering engine implementations, and more.

Chapter 3
The Graphics Processing Unit

"The display is the computer."
—Jen-Hsun Huang

Historically, graphics acceleration started with interpolating colors on each pixel scanline overlapping a triangle and then displaying these values. Including the ability to access image data allowed textures to be applied to surfaces. Adding hardware for interpolating and testing z-depths provided built-in visibility checking. Because of their frequent use, such processes were committed to dedicated hardware to increase performance. More parts of the rendering pipeline, and much more functionality for each, were added in successive generations. Dedicated graphics hardware's only computational advantage over the CPU is speed, but speed is critical.

Over the past two decades, graphics hardware has undergone an incredible transformation. The first consumer graphics chip to include hardware vertex processing (NVIDIA's GeForce256) shipped in 1999. NVIDIA coined the term *graphics processing unit* (GPU) to differentiate the GeForce 256 from the previously available rasterization-only chips, and it stuck. During the next few years, the GPU evolved from configurable implementations of a complex fixed-function pipeline to highly programmable blank slates where developers could implement their own algorithms. Programmable *shaders* of various kinds are the primary means by which the GPU is controlled. For efficiency, some parts of the pipeline remain configurable, not programmable, but the trend is toward programmability and flexibility [175].

GPUs gain their great speed from a focus on a narrow set of highly parallelizable tasks. They have custom silicon dedicated to implementing the z-buffer, to rapidly accessing texture images and other buffers, and to finding which pixels are covered by a triangle, for example. How these elements perform their functions is covered in Chapter 23. More important to know early on is how the GPU achieves parallelism for its programmable shaders.