

REAL-TIME RENDERING

FOURTH EDITION

Tomas Akenine-Möller

Eric Haines

Naty Hoffman

Angelo Pesce

Michał Iwanicki

Sébastien Hillaire



 **CRC Press**
Taylor & Francis Group
AN A K PETERS BOOK

Real-Time Rendering

Fourth Edition



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

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

© 2018 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works

Printed on acid-free paper

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

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (<http://www.copyright.com/>) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

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://lcn.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>



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Contents

Preface	xiii
1 Introduction	1
1.1 Contents Overview	3
1.2 Notation and Definitions	5
2 The Graphics Rendering Pipeline	11
2.1 Architecture	12
2.2 The Application Stage	13
2.3 Geometry Processing	14
2.4 Rasterization	21
2.5 Pixel Processing	22
2.6 Through the Pipeline	25
3 The Graphics Processing Unit	29
3.1 Data-Parallel Architectures	30
3.2 GPU Pipeline Overview	34
3.3 The Programmable Shader Stage	35
3.4 The Evolution of Programmable Shading and APIs	37
3.5 The Vertex Shader	42
3.6 The Tessellation Stage	44
3.7 The Geometry Shader	47
3.8 The Pixel Shader	49
3.9 The Merging Stage	53
3.10 The Compute Shader	54
4 Transforms	57
4.1 Basic Transforms	58
4.2 Special Matrix Transforms and Operations	70
4.3 Quaternions	76
4.4 Vertex Blending	84
4.5 Morphing	87
4.6 Geometry Cache Playback	92
4.7 Projections	92

5	Shading Basics	103
5.1	Shading Models	103
5.2	Light Sources	106
5.3	Implementing Shading Models	117
5.4	Aliasing and Antialiasing	130
5.5	Transparency, Alpha, and Compositing	148
5.6	Display Encoding	160
6	Texturing	167
6.1	The Texturing Pipeline	169
6.2	Image Texturing	176
6.3	Procedural Texturing	198
6.4	Texture Animation	200
6.5	Material Mapping	201
6.6	Alpha Mapping	202
6.7	Bump Mapping	208
6.8	Parallax Mapping	214
6.9	Textured Lights	221
7	Shadows	223
7.1	Planar Shadows	225
7.2	Shadows on Curved Surfaces	229
7.3	Shadow Volumes	230
7.4	Shadow Maps	234
7.5	Percentage-Closer Filtering	247
7.6	Percentage-Closer Soft Shadows	250
7.7	Filtered Shadow Maps	252
7.8	Volumetric Shadow Techniques	257
7.9	Irregular Z-Buffer Shadows	259
7.10	Other Applications	262
8	Light and Color	267
8.1	Light Quantities	267
8.2	Scene to Screen	281
9	Physically Based Shading	293
9.1	Physics of Light	293
9.2	The Camera	307
9.3	The BRDF	308
9.4	Illumination	315
9.5	Fresnel Reflectance	316
9.6	Microgeometry	327
9.7	Microfacet Theory	331

9.8	BRDF Models for Surface Reflection	336
9.9	BRDF Models for Subsurface Scattering	347
9.10	BRDF Models for Cloth	356
9.11	Wave Optics BRDF Models	359
9.12	Layered Materials	363
9.13	Blending and Filtering Materials	365
10	Local Illumination	375
10.1	Area Light Sources	377
10.2	Environment Lighting	391
10.3	Spherical and Hemispherical Functions	392
10.4	Environment Mapping	404
10.5	Specular Image-Based Lighting	414
10.6	Irradiance Environment Mapping	424
10.7	Sources of Error	433
11	Global Illumination	437
11.1	The Rendering Equation	437
11.2	General Global Illumination	441
11.3	Ambient Occlusion	446
11.4	Directional Occlusion	465
11.5	Diffuse Global Illumination	472
11.6	Specular Global Illumination	497
11.7	Unified Approaches	509
12	Image-Space Effects	513
12.1	Image Processing	513
12.2	Reprojection Techniques	522
12.3	Lens Flare and Bloom	524
12.4	Depth of Field	527
12.5	Motion Blur	536
13	Beyond Polygons	545
13.1	The Rendering Spectrum	545
13.2	Fixed-View Effects	546
13.3	Skyboxes	547
13.4	Light Field Rendering	549
13.5	Sprites and Layers	550
13.6	Billboarding	551
13.7	Displacement Techniques	564
13.8	Particle Systems	567
13.9	Point Rendering	572
13.10	Voxels	578

14	Volumetric and Translucency Rendering	589
14.1	Light Scattering Theory	589
14.2	Specialized Volumetric Rendering	600
14.3	General Volumetric Rendering	605
14.4	Sky Rendering	613
14.5	Translucent Surfaces	623
14.6	Subsurface Scattering	632
14.7	Hair and Fur	640
14.8	Unified Approaches	648
15	Non-Photorealistic Rendering	651
15.1	Toon Shading	652
15.2	Outline Rendering	654
15.3	Stroke Surface Stylization	669
15.4	Lines	673
15.5	Text Rendering	675
16	Polygonal Techniques	681
16.1	Sources of Three-Dimensional Data	682
16.2	Tessellation and Triangulation	683
16.3	Consolidation	690
16.4	Triangle Fans, Strips, and Meshes	696
16.5	Simplification	706
16.6	Compression and Precision	712
17	Curves and Curved Surfaces	717
17.1	Parametric Curves	718
17.2	Parametric Curved Surfaces	734
17.3	Implicit Surfaces	749
17.4	Subdivision Curves	753
17.5	Subdivision Surfaces	756
17.6	Efficient Tessellation	767
18	Pipeline Optimization	783
18.1	Profiling and Debugging Tools	784
18.2	Locating the Bottleneck	786
18.3	Performance Measurements	788
18.4	Optimization	790
18.5	Multiprocessing	805

19 Acceleration Algorithms	817
19.1 Spatial Data Structures	818
19.2 Culling Techniques	830
19.3 Backface Culling	831
19.4 View Frustum Culling	835
19.5 Portal Culling	837
19.6 Detail and Small Triangle Culling	839
19.7 Occlusion Culling	840
19.8 Culling Systems	850
19.9 Level of Detail	852
19.10 Rendering Large Scenes	866
20 Efficient Shading	881
20.1 Deferred Shading	883
20.2 Decal Rendering	888
20.3 Tiled Shading	892
20.4 Clustered Shading	898
20.5 Deferred Texturing	905
20.6 Object- and Texture-Space Shading	908
21 Virtual and Augmented Reality	915
21.1 Equipment and Systems Overview	916
21.2 Physical Elements	919
21.3 APIs and Hardware	924
21.4 Rendering Techniques	932
22 Intersection Test Methods	941
22.1 GPU-Accelerated Picking	942
22.2 Definitions and Tools	943
22.3 Bounding Volume Creation	948
22.4 Geometric Probability	953
22.5 Rules of Thumb	954
22.6 Ray/Sphere Intersection	955
22.7 Ray/Box Intersection	959
22.8 Ray/Triangle Intersection	962
22.9 Ray/Polygon Intersection	966
22.10 Plane/Box Intersection	970
22.11 Triangle/Triangle Intersection	972
22.12 Triangle/Box Intersection	974
22.13 Bounding-Volume/Bounding-Volume Intersection	976
22.14 View Frustum Intersection	981
22.15 Line/Line Intersection	987
22.16 Intersection between Three Planes	990

23 Graphics Hardware	993
23.1 Rasterization	993
23.2 Massive Compute and Scheduling	1002
23.3 Latency and Occupancy	1004
23.4 Memory Architecture and Buses	1006
23.5 Caching and Compression	1007
23.6 Color Buffering	1009
23.7 Depth Culling, Testing, and Buffering	1014
23.8 Texturing	1017
23.9 Architecture	1019
23.10 Case Studies	1024
23.11 Ray Tracing Architectures	1039
24 The Future	1041
24.1 Everything Else	1042
24.2 You	1046
Bibliography	1051
Index	1155

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 Sjørgård, 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 Sjørgård, 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 Prankevičius, 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 Villedor, Carsten Wenzel, Steve Westin, Chris Wyman, Cem Yuksel, Billy Zelnack, 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



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

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 $(v_x \ v_y \ v_z)^T$, 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 $\mathbf{v} = (v_x \ v_y \ v_z \ v_w)^T$, where a vector is $\mathbf{v} = (v_x \ v_y \ v_z \ 0)^T$ and a point is $\mathbf{v} = (v_x \ v_y \ v_z \ 1)^T$. 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 $\mathbf{v} = (v_0 \ v_1 \ v_2)^T$. 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](#):

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

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 j th column vector and $\mathbf{m}_i$ represents the i th 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:

$$\mathbf{M} = \begin{pmatrix} \mathbf{m}_{,0} & \mathbf{m}_{,1} & \mathbf{m}_{,2} \end{pmatrix} = \begin{pmatrix} \mathbf{m}_x & \mathbf{m}_y & \mathbf{m}_z \end{pmatrix} = \begin{pmatrix} \mathbf{m}_0^T \\ \mathbf{m}_1^T \\ \mathbf{m}_2^T \end{pmatrix}. \quad (1.2)$$

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 $\mathbf{v} = (v_x \ v_y \ v_z)^T$. 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:	$ \cdot $	determinant of a matrix
6:	$ \cdot $	absolute value of a scalar
7:	$\ \cdot\ $	length (or norm) of argument
8:	x^+	clamping x to 0
9:	$x^\mp$	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:

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

and operator 9 clamps values between 0 and 1:

$$x^\mp = \begin{cases} 1, & \text{if } x \geq 1, \\ x, & \text{if } 0 < x < 1, \\ 0, & \text{otherwise.} \end{cases} \quad (1.4)$$

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

$$n! = n(n-1)(n-2) \cdots 3 \cdot 2 \cdot 1. \quad (1.5)$$

The eleventh operator, the binomial factor, is defined as shown in [Equation 1.6](#):

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

	Function	Description
1:	<code>atan2(y, x)</code>	two-value arctangent
2:	<code>log(n)</code>	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 $\mathbf{e}_x = (1 \ 0 \ 0)^T$, $\mathbf{e}_y = (0 \ 1 \ 0)^T$, and $\mathbf{e}_z = (0 \ 0 \ 1)^T$ 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 $0 \leq \text{atan2}(y, x) < 2\pi$, 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

¹The 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.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

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.

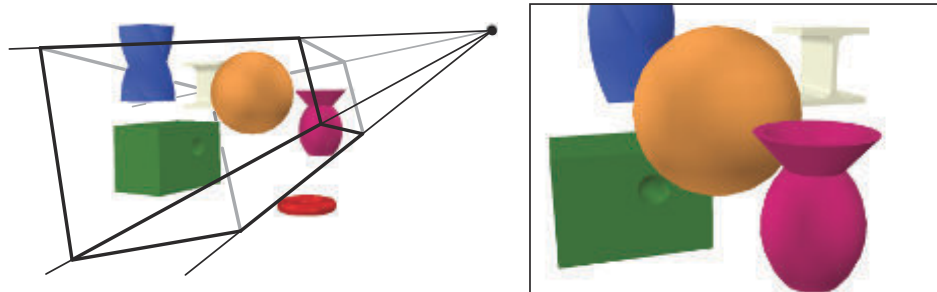


Figure 2.1. In the left image, a virtual camera is located at the tip of the pyramid (where four lines converge). Only the primitives inside the view volume are rendered. For an image that is rendered in perspective (as is the case here), the view volume is a *frustum* (plural: *frusta*), i.e., a truncated 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

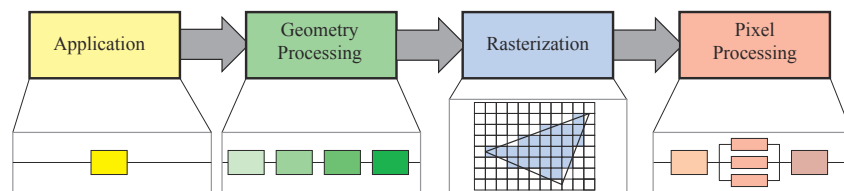


Figure 2.2. The basic construction of the rendering pipeline, consisting of four stages: application, geometry processing, rasterization, and pixel processing. Each of these stages may be a pipeline in itself, as illustrated below the geometry processing stage, or a stage may be (partly) parallelized, 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 $1/\text{seconds}$, 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).

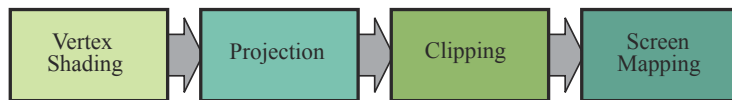


Figure 2.3. The geometry processing stage divided into a pipeline of functional stages.

¹Since 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

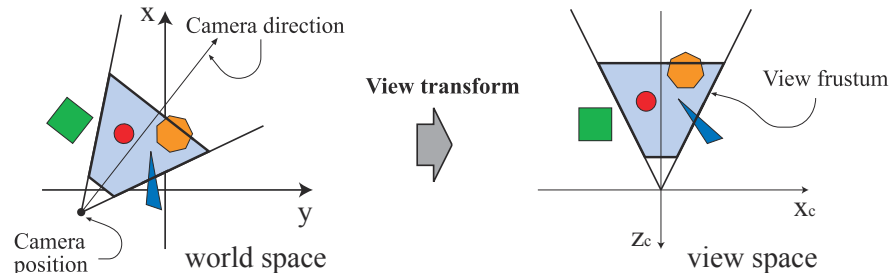


Figure 2.4. In the left illustration, a top-down view shows the camera located and oriented as the user wants it to be, in a world where the $+z$ -axis is up. The view transform reorients the world so that the camera is at the origin, looking along its negative z -axis, with the camera's $+y$ -axis up, 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*)

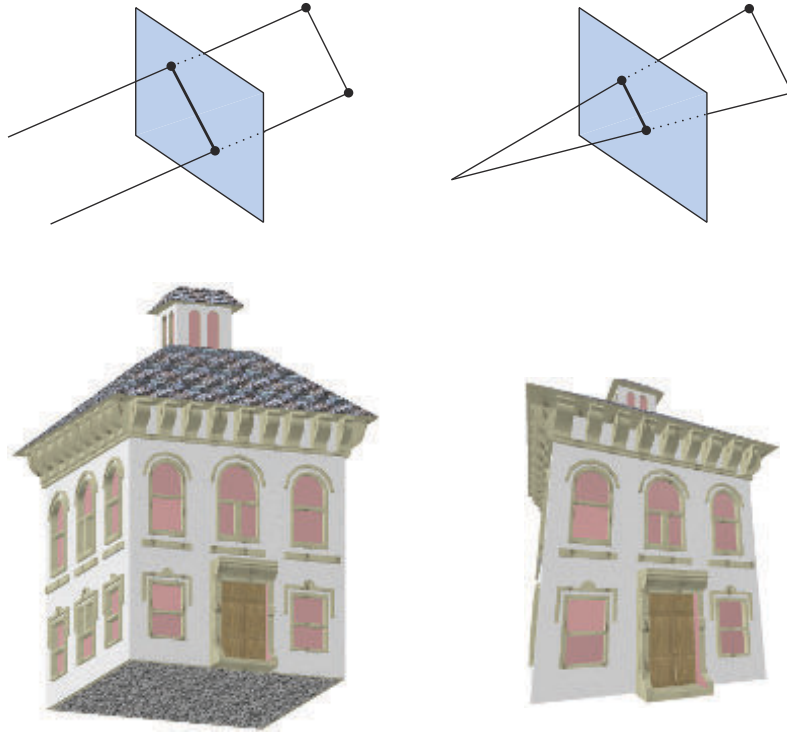


Figure 2.5. On the left is an orthographic, or parallel, projection; on the right is a perspective projection.

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.

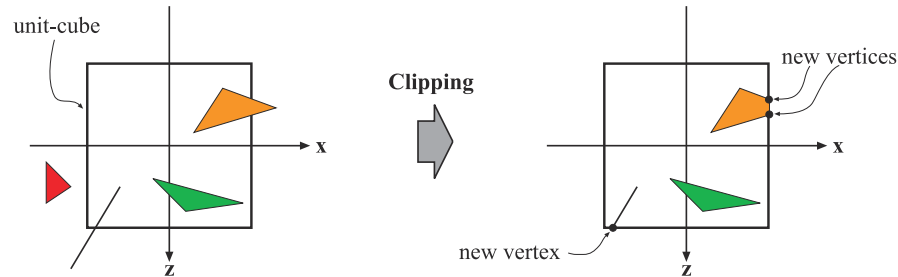


Figure 2.6. After the projection transform, only the primitives inside the unit cube (which correspond to primitives inside the view frustum) are needed for continued processing. Therefore, the primitives outside the unit cube are discarded, and primitives fully inside are kept. Primitives intersecting 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.

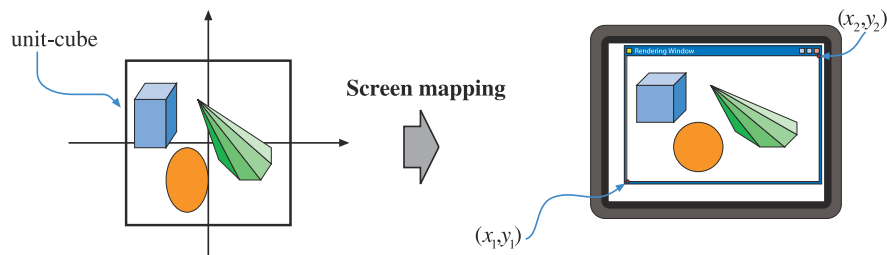


Figure 2.7. The primitives lie in the unit cube after the projection transform, and the screen mapping procedure takes care of finding the coordinates on the screen.

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

$$d = \text{floor}(c), \quad (2.1)$$

$$c = d + 0.5, \quad (2.2)$$

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 as 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.

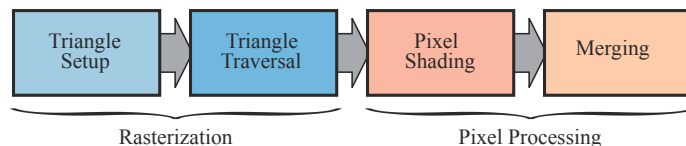


Figure 2.8. Left: rasterization split into two functional stages, called triangle setup and triangle traversal. Right: pixel processing split into two functional stages, namely, pixel processing and merging.

“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.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Chapter 3

The Graphics Processing Unit

“The display is the computer.”
—Jen-Hsun Huang

Historically, graphics acceleration started with interpolating colors on each pixel scan-line 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.

Section 3.3 explains how shaders function. For now, what you need to know is that a shader core is a small processor that does some relatively isolated task, such as transforming a vertex from its location in the world to a screen coordinate, or computing the color of a pixel covered by a triangle. With thousands or millions of triangles being sent to the screen each frame, every second there can be billions of *shader invocations*, that is, separate instances where shader programs are run.

To begin with, *latency* is a concern that all processors face. Accessing data takes some amount of time. A basic way to think about latency is that the farther away the information is from the processor, the longer the wait. Section 23.3 covers latency in more detail. Information stored in memory chips will take longer to access than that in local registers. Section 18.4.1 discusses memory access in more depth. The key point is that waiting for data to be retrieved means the processor is stalled, which reduces performance.

3.1 Data-Parallel Architectures

Various strategies are used by different processor architectures to avoid stalls. A CPU is optimized to handle a wide variety of data structures and large code bases. CPUs can have multiple processors, but each runs code in a mostly serial fashion, limited SIMD vector processing being the minor exception. To minimize the effect of latency, much of a CPU's chip consists of fast local caches, memory that is filled with data likely to be needed next. CPUs also avoid stalls by using clever techniques such as branch prediction, instruction reordering, register renaming, and cache prefetching [715]

GPUs take a different approach. Much of a GPU's chip area is dedicated to a large set of processors, called *shader cores*, often numbering in the thousands. The GPU is a stream processor, in which ordered sets of similar data are processed in turn. Because of this similarity—a set of vertices or pixels, for example—the GPU can process these data in a massively parallel fashion. One other important element is that these invocations are as independent as possible, such that they have no need for information from neighboring invocations and do not share writable memory locations. This rule is sometimes broken to allow new and useful functionality, but such exceptions come at a price of potential delays, as one processor may wait on another processor to finish its work.

The GPU is optimized for *throughput*, defined as the maximum rate at which data can be processed. However, this rapid processing has a cost. With less chip area dedicated to cache memory and control logic, latency for each shader core is generally considerably higher than what a CPU processor encounters [462].

Say a mesh is rasterized and two thousand pixels have fragments to be processed; a pixel shader program is to be invoked two thousand times. Imagine there is only a single shader processor, the world's weakest GPU. It starts to execute the shader program for the first fragment of the two thousand. The shader processor performs a few arithmetic operations on values in registers. Registers are local and quick to

access, so no stall occurs. The shader processor then comes to an instruction such as a texture access; e.g., for a given surface location the program needs to know the pixel color of the image applied to the mesh. A texture is an entirely separate resource, not a part of the pixel program's local memory, and texture access can be somewhat involved. A memory fetch can take hundreds to thousands of clock cycles, during which time the GPU processor is doing nothing. At this point the shader processor would stall, waiting for the texture's color value to be returned.

To make this terrible GPU into something considerably better, give each fragment a little storage space for its local registers. Now, instead of stalling on a texture fetch, the shader processor is allowed to switch and execute another fragment, number two of two thousand. This switch is extremely fast, nothing in the first or second fragment is affected other than noting which instruction was executing on the first. Now the second fragment is executed. Same as with the first, a few arithmetic functions are performed, then a texture fetch is again encountered. The shader core now switches to another fragment, number three. Eventually all two thousand fragments are processed in this way. At this point the shader processor returns to fragment number one. By this time the texture color has been fetched and is available for use, so the shader program can then continue executing. The processor proceeds in the same fashion until another instruction that is known to stall execution is encountered, or the program completes. A single fragment will take longer to execute than if the shader processor stayed focused on it, but overall execution time for the fragments as a whole is dramatically reduced.

In this architecture, latency is hidden by having the GPU stay busy by switching to another fragment. GPUs take this design a step further by separating the instruction execution logic from the data. Called *single instruction, multiple data* (SIMD), this arrangement executes the same command in lock-step on a fixed number of shader programs. The advantage of SIMD is that considerably less silicon (and power) needs to be dedicated to processing data and switching, compared to using an individual logic and dispatch unit to run each program. Translating our two-thousand-fragment example into modern GPU terms, each pixel shader invocation for a fragment is called a *thread*. This type of thread is unlike a CPU thread. It consists of a bit of memory for the input values to the shader, along with any register space needed for the shader's execution. Threads that use the same shader program are bundled into groups, called *warps* by NVIDIA and *wavefronts* by AMD. A warp/wavefront is scheduled for execution by some number GPU shader cores, anywhere from 8 to 64, using SIMD-processing. Each thread is mapped to a *SIMD lane*.

Say we have two thousand threads to be executed. Warps on NVIDIA GPUs contain 32 threads. This yields $2000/32 = 62.5$ warps, which means that 63 warps are allocated, one warp being half empty. A warp's execution is similar to our single GPU processor example. The shader program is executed in lock-step on all 32 processors. When a memory fetch is encountered, all threads encounter it at the same time, because the same instruction is executed for all. The fetch signals that this warp of threads will stall, all waiting for their (different) results. Instead of stalling, the

warp is swapped out for a different warp of 32 threads, which is then executed by the 32 cores. This swapping is just as fast as with our single processor system, as no data within each thread is touched when a warp is swapped in or out. Each thread has its own registers, and each warp keeps track of which instruction it is executing. Swapping in a new warp is just a matter of pointing the set of cores at a different set of threads to execute; there is no other overhead. Warps execute or swap out until all are completed. See [Figure 3.1](#).

In our simple example the latency of a memory fetch for a texture can cause a warp to swap out. In reality warps could be swapped out for shorter delays, since the cost of swapping is so low. There are several other techniques used to optimize execution [945], but warp-swapping is the major latency-hiding mechanism used by all GPUs. Several factors are involved in how efficiently this process works. For example, if there are few threads, then few warps can be created, making latency hiding problematic.

The shader program's structure is an important characteristic that influences efficiency. A major factor is the amount of register use for each thread. In our example we assume that two thousand threads can all be resident on the GPU at one time. The more registers needed by the shader program associated with each thread, the fewer threads, and thus the fewer warps, can be resident in the GPU. A shortage of warps can mean that a stall cannot be mitigated by swapping. Warps that are resident are said to be "in flight," and this number is called the *occupancy*. High occupancy means that there are many warps available for processing, so that idle processors are less likely. Low occupancy will often lead to poor performance. The frequency of memory fetches also affects how much latency hiding is needed. Lauritzen [993] outlines how occupancy is affected by the number of registers and the shared memory that a shader uses. Wronski [1911, 1914] discusses how the ideal occupancy rate can vary depending on the type of operations a shader performs.

Another factor affecting overall efficiency is dynamic branching, caused by "if" statements and loops. Say an "if" statement is encountered in a shader program. If all the threads evaluate and take the same branch, the warp can continue without any concern about the other branch. However, if some threads, or even one thread, take the alternate path, then the warp must execute both branches, throwing away the results not needed by each particular thread [530, 945]. This problem is called *thread divergence*, where a few threads may need to execute a loop iteration or perform an "if" path that the other threads in the warp do not, leaving them idle during this time.

All GPUs implement these architectural ideas, resulting in systems with strict limitations but massive amounts of compute power per watt. Understanding how this system operates will help you as a programmer make more efficient use of the power it provides. In the sections that follow we discuss how the GPU implements the rendering pipeline, how programmable shaders operate, and the evolution and function of each GPU stage.

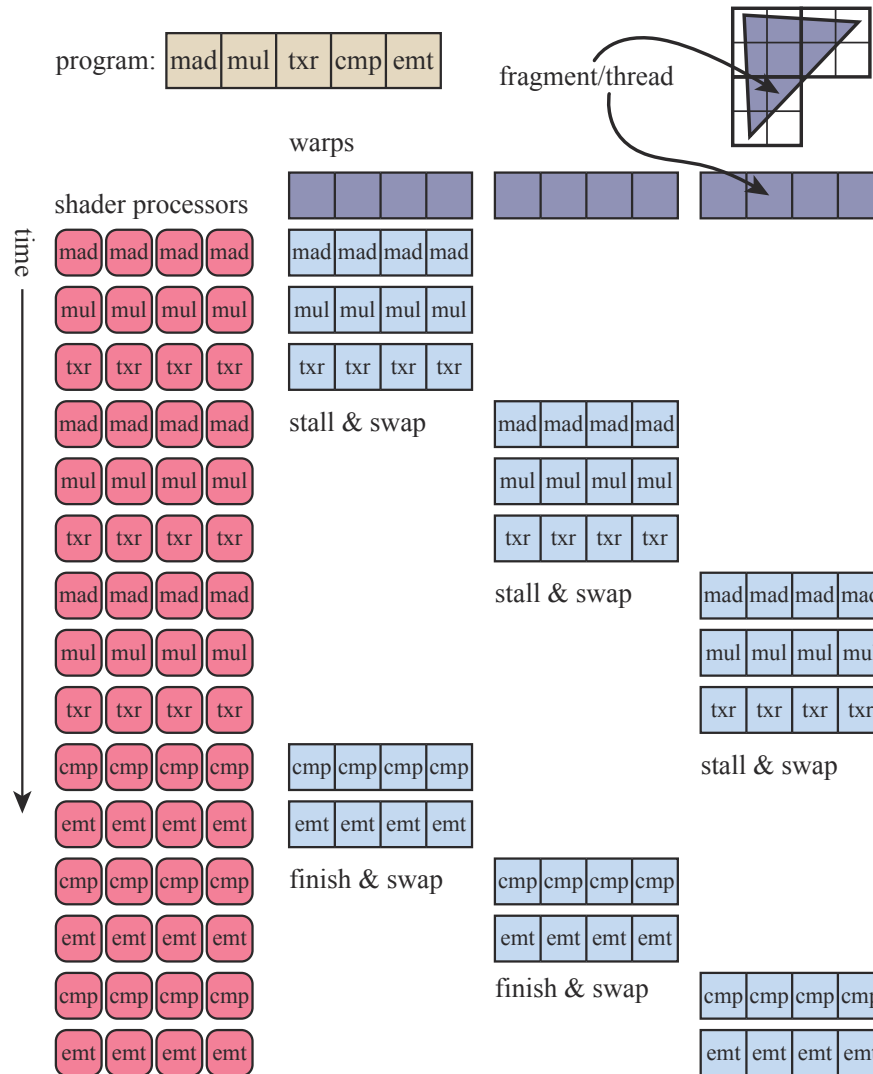


Figure 3.1. Simplified shader execution example. A triangle's fragments, called threads, are gathered into warps. Each warp is shown as four threads but have 32 threads in reality. The shader program to be executed is five instructions long. The set of four GPU shader processors executes these instructions for the first warp until a stall condition is detected on the "txr" command, which needs time to fetch its data. The second warp is swapped in and the shader program's first three instructions are applied to it, until a stall is again detected. After the third warp is swapped in and stalls, execution continues by swapping in the first warp and continuing execution. If its "txr" command's data are not yet returned at this point, execution truly stalls until these data are available. Each warp finishes in turn.

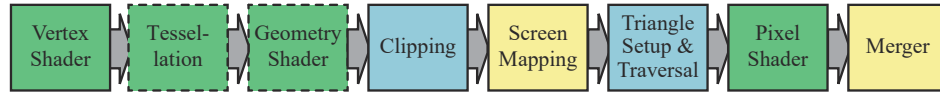


Figure 3.2. GPU implementation of the rendering pipeline. The stages are color coded according to the degree of user control over their operation. Green stages are fully programmable. Dashed lines show optional stages. Yellow stages are configurable but not programmable, e.g., various blend modes can be set for the merge stage. Blue stages are completely fixed in their function.

3.2 GPU Pipeline Overview

The GPU implements the conceptual geometry processing, rasterization, and pixel processing pipeline stages described in [Chapter 2](#). These are divided into several hardware stages with varying degrees of configurability or programmability. [Figure 3.2](#) shows the various stages color coded according to how programmable or configurable they are. Note that these physical stages are split up somewhat differently than the functional stages presented in [Chapter 2](#).

We describe here the *logical model* of the GPU, the one that is exposed to you as a programmer by an API. As [Chapters 18](#) and [23](#) discuss, the implementation of this logical pipeline, the *physical model*, is up to the hardware vendor. A stage that is fixed-function in the logical model may be executed on the GPU by adding commands to an adjacent programmable stage. A single program in the pipeline may be split into elements executed by separate sub-units, or be executed by a separate pass entirely. The logical model can help you reason about what affects performance, but it should not be mistaken for the way the GPU actually implements the pipeline.

The vertex shader is a fully programmable stage that is used to implement the geometry processing stage. The geometry shader is a fully programmable stage that operates on the vertices of a primitive (point, line, or triangle). It can be used to perform per-primitive shading operations, to destroy primitives, or to create new ones. The tessellation stage and geometry shader are both optional, and not all GPUs support them, especially on mobile devices.

The clipping, triangle setup, and triangle traversal stages are implemented by fixed-function hardware. Screen mapping is affected by window and viewport settings, internally forming a simple scale and repositioning. The pixel shader stage is fully programmable. Although the merger stage is not programmable, it is highly configurable and can be set to perform a wide variety of operations. It implements the “merging” functional stage, in charge of modifying the color, *z*-buffer, blend, stencil, and any other output-related buffers. The pixel shader execution together with the merger stage form the conceptual pixel processing stage presented in [Chapter 2](#).

Over time, the GPU pipeline has evolved away from hard-coded operation and toward increasing flexibility and control. The introduction of programmable shader stages was the most important step in this evolution. The next section describes the features common to the various programmable stages.

3.3 The Programmable Shader Stage

Modern shader programs use a unified shader design. This means that the vertex, pixel, geometry, and tessellation-related shaders share a common programming model. Internally they have the same *instruction set architecture* (ISA). A processor that implements this model is called a *common-shader core* in DirectX, and a GPU with such cores is said to have a unified shader architecture. The idea behind this type of architecture is that shader processors are usable in a variety of roles, and the GPU can allocate these as it sees fit. For example, a set of meshes with tiny triangles will need more vertex shader processing than large squares each made of two triangles. A GPU with separate pools of vertex and pixel shader cores means that the ideal work distribution to keep all the cores busy is rigidly predetermined. With unified shader cores, the GPU can decide how to balance this load.

Describing the entire shader programming model is well beyond the scope of this book, and there are many documents, books, and websites that already do so. Shaders are programmed using C-like *shading languages* such as DirectX's *High-Level Shading Language* (HLSL) and the *OpenGL Shading Language* (GLSL). DirectX's HLSL can be compiled to virtual machine bytecode, also called the *intermediate language* (IL or DXIL), to provide hardware independence. An intermediate representation can also allow shader programs to be compiled and stored offline. This intermediate language is converted to the ISA of the specific GPU by the driver. Console programming usually avoids the intermediate language step, since there is then only one ISA for the system.

The basic data types are 32-bit single-precision floating point scalars and vectors, though vectors are only part of the shader code and are not supported in hardware as outlined above. On modern GPUs 32-bit integers and 64-bit floats are also supported natively. Floating point vectors typically contain data such as positions (*xyzw*), normals, matrix rows, colors (*rgba*), or texture coordinates (*uvwq*). Integers are most often used to represent counters, indices, or bitmasks. Aggregate data types such as structures, arrays, and matrices are also supported.

A *draw call* invokes the graphics API to draw a group of primitives, so causing the graphics pipeline to execute and run its shaders. Each programmable shader stage has two types of inputs: *uniform* inputs, with values that remain constant throughout a draw call (but can be changed between draw calls), and *varying* inputs, data that come from the triangle's vertices or from rasterization. For example, a pixel shader may provide the color of a light source as a uniform value, and the triangle surface's location changes per pixel and so is varying. A texture is a special kind of uniform input that once was always a color image applied to a surface, but that now can be thought of as any large array of data.

The underlying virtual machine provides special registers for the different types of inputs and outputs. The number of available *constant registers* for uniforms is much larger than those registers available for varying inputs or outputs. This happens because the varying inputs and outputs need to be stored separately for each vertex

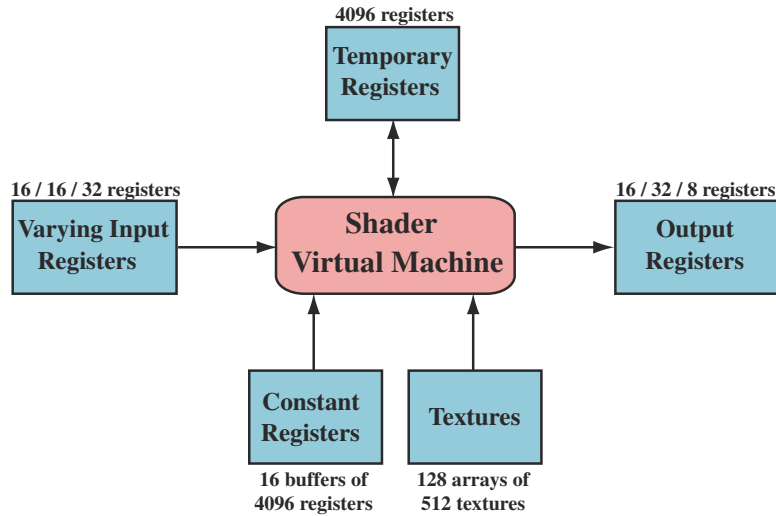


Figure 3.3. Unified virtual machine architecture and register layout, under Shader Model 4.0. The maximum available number is indicated next to each resource. Three numbers separated by slashes refer to the limits for vertex, geometry, and pixel shaders (from left to right).

or pixel, so there is a natural limit as to how many are needed. The uniform inputs are stored once and reused across all the vertices or pixels in the draw call. The virtual machine also has general-purpose *temporary registers*, which are used for scratch space. All types of registers can be array-indexed using integer values in temporary registers. The inputs and outputs of the shader virtual machine can be seen in [Figure 3.3](#).

Operations that are common in graphics computations are efficiently executed on modern GPUs. Shading languages expose the most common of these operations (such as additions and multiplications) via operators such as `*` and `+`. The rest are exposed through *intrinsic functions*, e.g., `atan()`, `sqrt()`, `log()`, and many others, optimized for the GPU. Functions also exist for more complex operations, such as vector normalization and reflection, the cross product, and matrix transpose and determinant computations.

The term *flow control* refers to the use of branching instructions to change the flow of code execution. Instructions related to flow control are used to implement high-level language constructs such as “if” and “case” statements, as well as various types of loops. Shaders support two types of flow control. *Static flow control* branches are based on the values of uniform inputs. This means that the flow of the code is constant over the draw call. The primary benefit of static flow control is to allow the same shader to be used in a variety of different situations (e.g., a varying numbers of lights). There is no thread divergence, since all invocations take the same code path. *Dynamic flow control* is based on the values of varying inputs, meaning that each

fragment can execute the code differently. This is much more powerful than static flow control but can cost performance, especially if the code flow changes erratically between shader invocations.

3.4 The Evolution of Programmable Shading and APIs

The idea of a framework for programmable shading dates back to 1984 with Cook's *shade trees* [287]. A simple shader and its corresponding shade tree are shown in Figure 3.4. The RenderMan Shading Language [63, 1804] was developed from this idea in the late 1980s. It is still used today for film production rendering, along with other evolving specifications, such as the *Open Shading Language* (OSL) project [608].

Consumer-level graphics hardware was first successfully introduced by 3dfx Interactive on October 1, 1996. See Figure 3.5 for a timeline from this year. Their Voodoo graphics card's ability to render the game *Quake* with high quality and performance led to its quick adoption. This hardware implemented a fixed-function pipeline throughout. Before GPUs supported programmable shaders natively, there were several attempts to implement programmable shading operations in real time via multiple rendering passes. The *Quake III: Arena* scripting language was the first widespread

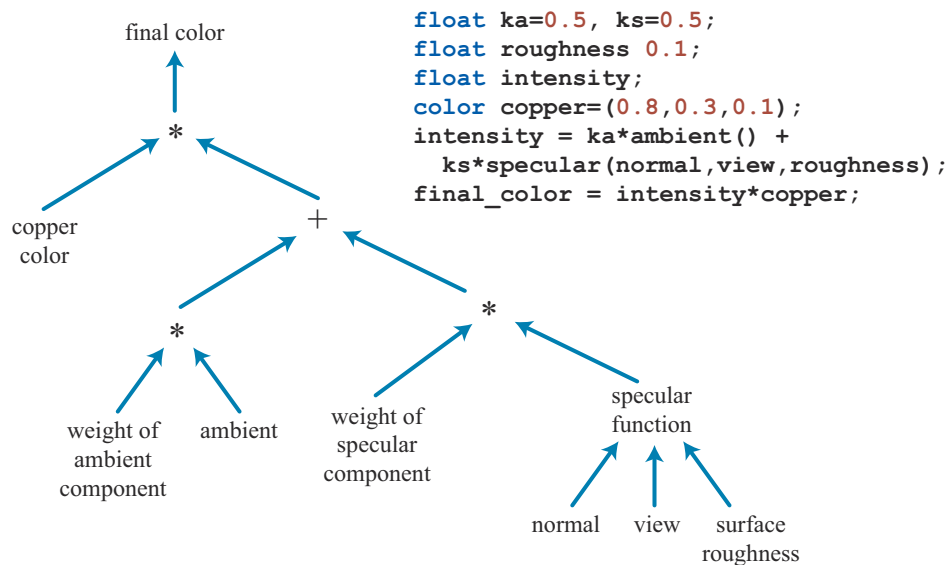


Figure 3.4. Shade tree for a simple copper shader, and its corresponding shader language program. (After Cook [287].)

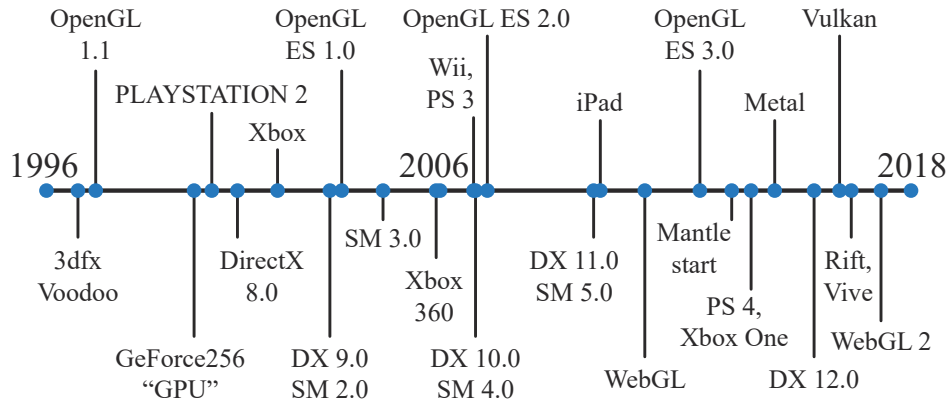


Figure 3.5. A timeline of some API and graphics hardware releases.

commercial success in this area in 1999. As mentioned at the beginning of the chapter, NVIDIA's GeForce256 was the first hardware to be called a GPU, but it was not programmable. However, it was configurable.

In early 2001, NVIDIA's GeForce 3 was the first GPU to support programmable vertex shaders [1049], exposed through DirectX 8.0 and extensions to OpenGL. These shaders were programmed in an assembly-like language that was converted by the drivers into microcode on the fly. Pixel shaders were also included in DirectX 8.0, but pixel shaders fell short of actual programmability—the limited “programs” supported were converted into texture blending states by the driver, which in turn wired together hardware “register combiners.” These “programs” were not only limited in length (12 instructions or less) but also lacked important functionality. Dependent texture reads and floating point data were identified by Peercy et al. [1363] as crucial to true programmability, from their study of RenderMan.

Shaders at this time did not allow for flow control (branching), so conditionals had to be emulated by computing both terms and selecting or interpolating between the results. DirectX defined the concept of a *Shader Model* (SM) to distinguish hardware with different shader capabilities. The year 2002 saw the release of DirectX 9.0 including Shader Model 2.0, which featured truly programmable vertex and pixel shaders. Similar functionality was also exposed under OpenGL using various extensions. Support for arbitrary dependent texture reads and storage of 16-bit floating point values was added, finally completing the set of requirements identified by Peercy et al. Limits on shader resources such as instructions, textures, and registers were increased, so shaders became capable of more complex effects. Support for flow control was also added. The growing length and complexity of shaders made the assembly programming model increasingly cumbersome. Fortunately, DirectX 9.0

also included HLSL. This shading language was developed by Microsoft in collaboration with NVIDIA. Around the same time, the OpenGL ARB (Architecture Review Board) released GLSL, a fairly similar language for OpenGL [885]. These languages were heavily influenced by the syntax and design philosophy of the C programming language and included elements from the RenderMan Shading Language.

Shader Model 3.0 was introduced in 2004 and added dynamic flow control, making shaders considerably more powerful. It also turned optional features into requirements, further increased resource limits and added limited support for texture reads in vertex shaders. When a new generation of game consoles was introduced in late 2005 (Microsoft's Xbox 360) and late 2006 (Sony Computer Entertainment's PLAYSTATION 3 system), they were equipped with Shader Model 3.0-level GPUs. Nintendo's Wii console was one of the last notable fixed-function GPUs, which initially shipped in late 2006. The purely fixed-function pipeline is long gone at this point. Shader languages have evolved to a point where a variety of tools are used to create and manage them. A screenshot of one such tool, using Cook's shade tree concept, is shown in Figure 3.6.

The next large step in programmability also came near the end of 2006. Shader Model 4.0, included in DirectX 10.0 [175], introduced several major features, such as the geometry shader and stream output. Shader Model 4.0 included a uniform

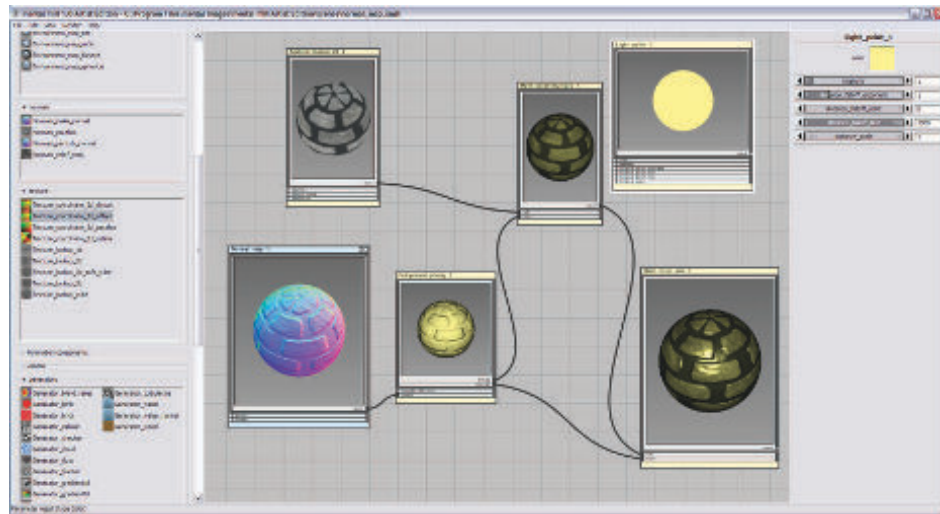


Figure 3.6. A visual shader graph system for shader design. Various operations are encapsulated in function boxes, selectable on the left. When selected, each function box has adjustable parameters, shown on the right. Inputs and outputs for each function box are linked to each other to form the final result, shown in the lower right of the center frame. (Screenshot from “mental mill,” mental images inc.)

programming model for all shaders (vertex, pixel, and geometry), the unified shader design described earlier. Resource limits were further increased, and support for integer data types (including bitwise operations) was added. The introduction of GLSL 3.30 in OpenGL 3.3 provided a similar shader model.

In 2009 DirectX 11 and Shader Model 5.0 were released, adding the tessellation stage shaders and the compute shader, also called DirectCompute. The release also focused on supporting CPU multiprocessing more effectively, a topic discussed in [Section 18.5](#). OpenGL added tessellation in version 4.0 and compute shaders in 4.3. DirectX and OpenGL evolve differently. Both set a certain level of hardware support needed for a particular version release. Microsoft controls the DirectX API and so works directly with independent hardware vendors (IHVs) such as AMD, NVIDIA, and Intel, as well as game developers and computer-aided design software firms, to determine what features to expose. OpenGL is developed by a consortium of hardware and software vendors, managed by the nonprofit Khronos Group. Because of the number of companies involved, the API features often appear in a release of OpenGL some time after their introduction in DirectX. However, OpenGL allows *extensions*, vendor-specific or more general, that allow the latest GPU functions to be used before official support in a release.

The next significant change in APIs was led by AMD's introduction of the Mantle API in 2013. Developed in partnership with video game developer DICE, the idea of Mantle was to strip out much of the graphics driver's overhead and give this control directly to the developer. Alongside this refactoring was further support for effective CPU multiprocessing. This new class of APIs focuses on vastly reducing the time the CPU spends in the driver, along with more efficient CPU multiprocessor support ([Chapter 18](#)). The ideas pioneered in Mantle were picked up by Microsoft and released as DirectX 12 in 2015. Note that DirectX 12 is not focused on exposing new GPU functionality—DirectX 11.3 exposed the same hardware features. Both APIs can be used to send graphics to virtual reality systems such as the Oculus Rift and HTC Vive. However, DirectX 12 is a radical redesign of the API, one that better maps to modern GPU architectures. Low-overhead drivers are useful for applications where the CPU driver cost is causing a bottleneck, or where using more CPU processors for graphics could benefit performance [\[946\]](#). Porting from earlier APIs can be difficult, and a naive implementation can result in lower performance [\[249, 699, 1438\]](#).

Apple released its own low-overhead API called Metal in 2014. Metal was first available on mobile devices such as the iPhone 5S and iPad Air, with newer Macintoshes given access a year later through OS X El Capitan. Beyond efficiency, reducing CPU usage saves power, an important factor on mobile devices. This API has its own shading language, meant for both graphics and GPU compute programs.

AMD donated its Mantle work to the Khronos Group, which released its own new API in early 2016, called Vulkan. As with OpenGL, Vulkan works on multiple operating systems. Vulkan uses a new high-level intermediate language called SPIR-V, which is used for both shader representation and for general GPU computing. Precompiled shaders are portable and so can be used on any GPU supporting the

capabilities needed [885]. Vulkan can also be used for non-graphical GPU computation, as it does not need a display window [946]. One notable difference of Vulkan from other low-overhead drivers is that it is meant to work with a wide range of systems, from workstations to mobile devices.

On mobile devices the norm has been to use OpenGL ES. “ES” stands for Embedded Systems, as this API was developed with mobile devices in mind. Standard OpenGL at the time was rather bulky and slow in some of its call structures, as well as requiring support for rarely used functionality. Released in 2003, OpenGL ES 1.0 was a stripped-down version of OpenGL 1.3, describing a fixed-function pipeline. While releases of DirectX are timed with those of graphics hardware that support them, developing graphics support for mobile devices did not proceed in the same fashion. For example, the first iPad, released in 2010, implemented OpenGL ES 1.1. In 2007 the OpenGL ES 2.0 specification was released, providing programmable shading. It was based on OpenGL 2.0, but without the fixed-function component, and so was not backward-compatible with OpenGL ES 1.1. OpenGL ES 3.0 was released in 2012, providing functionality such as multiple render targets, texture compression, transform feedback, instancing, and a much wider range of texture formats and modes, as well as shader language improvements. OpenGL ES 3.1 adds compute shaders, and 3.2 adds geometry and tessellation shaders, among other features. [Chapter 23](#) discusses mobile device architectures in more detail.

An offshoot of OpenGL ES is the browser-based API WebGL, called through JavaScript. Released in 2011, the first version of this API is usable on most mobile devices, as it is equivalent to OpenGL ES 2.0 in functionality. As with OpenGL, extensions give access to more advanced GPU features. WebGL 2 assumes OpenGL ES 3.0 support.

WebGL is particularly well suited for experimenting with features or use in the classroom:

- It is cross-platform, working on all personal computers and almost all mobile devices.
- Driver approval is handled by the browsers. Even if one browser does not support a particular GPU or extension, often another browser does.
- Code is interpreted, not compiled, and only a text editor is needed for development.
- A debugger is built in to most browsers, and code running at any website can be examined.
- Programs can be deployed by uploading them to a website or Github, for example.

Higher-level scene-graph and effects libraries such as three.js [218] give easy access to code for a variety of more involved effects such as shadow algorithms, post-processing effects, physically based shading, and deferred rendering.

3.5 The Vertex Shader

The vertex shader is the first stage in the functional pipeline shown in [Figure 3.2](#). While this is the first stage directly under programmer control, it is worth noting that some data manipulation happens before this stage. In what DirectX calls the *input assembler* [[175](#), [530](#), [1208](#)], several streams of data can be woven together to form the sets of vertices and primitives sent down the pipeline. For example, an object could be represented by one array of positions and one array of colors. The input assembler would create this object's triangles (or lines or points) by creating vertices with positions and colors. A second object could use the same array of positions (along with a different model transform matrix) and a different array of colors for its representation. Data representation is discussed in detail in [Section 16.4.5](#). There is also support in the input assembler to perform *instancing*. This allows an object to be drawn several times with some varying data per instance, all with a single draw call. The use of instancing is covered in [Section 18.4.2](#).

A triangle mesh is represented by a set of vertices, each associated with a specific position on the model surface. Besides position, there are other optional properties associated with each vertex, such as a color or texture coordinates. Surface normals are defined at mesh vertices as well, which may seem like an odd choice. Mathematically, each triangle has a well-defined surface normal, and it may seem to make more sense to use the triangle's normal directly for shading. However, when rendering, triangle meshes are often used to represent an underlying curved surface, and vertex normals are used to represent the orientation of this surface, rather than that of the triangle mesh itself. [Section 16.3.4](#) will discuss methods to compute vertex normals. [Figure 3.7](#) shows side views of two triangle meshes that represent curved surfaces, one smooth and one with a sharp crease.

The vertex shader is the first stage to process the triangle mesh. The data describing what triangles are formed is unavailable to the vertex shader. As its name implies, it deals exclusively with the incoming vertices. The vertex shader provides a way

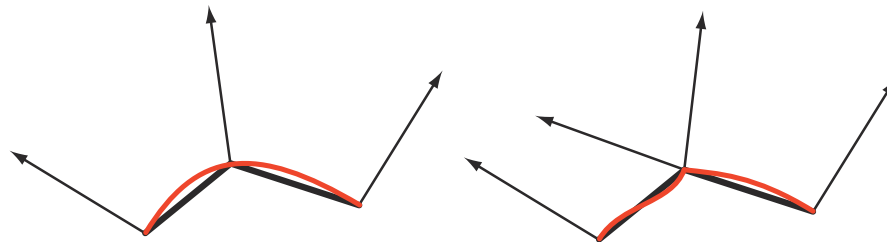


Figure 3.7. Side views of triangle meshes (in black, with vertex normals) representing curved surfaces (in red). On the left smoothed vertex normals are used to represent a smooth surface. On the right the middle vertex has been duplicated and given two normals, representing a crease.

to modify, create, or ignore values associated with each triangle's vertex, such as its color, normal, texture coordinates, and position. Normally the vertex shader program transforms vertices from model space to homogeneous clip space ([Section 4.7](#)). At a minimum, a vertex shader must always output this location.

A vertex shader is much the same as the unified shader described earlier. Every vertex passed in is processed by the vertex shader program, which then outputs a number of values that are interpolated across a triangle or line. The vertex shader can neither create nor destroy vertices, and results generated by one vertex cannot be passed on to another vertex. Since each vertex is treated independently, any number of shader processors on the GPU can be applied in parallel to the incoming stream of vertices.

Input assembly is usually presented as a process that happens before the vertex shader is executed. This is an example where the physical model often differs from the logical. Physically, the fetching of data to create a vertex might happen in the vertex shader and the driver will quietly prepend every shader with the appropriate instructions, invisible to the programmer.

Chapters that follow explain several vertex shader effects, such as vertex blending for animating joints, and silhouette rendering. Other uses for the vertex shader include:

- Object generation, by creating a mesh only once and having it be deformed by the vertex shader.
- Animating character's bodies and faces using skinning and morphing techniques.
- Procedural deformations, such as the movement of flags, cloth, or water [802, 943].
- Particle creation, by sending degenerate (no area) meshes down the pipeline and having these be given an area as needed.
- Lens distortion, heat haze, water ripples, page curls, and other effects, by using the entire framebuffer's contents as a texture on a screen-aligned mesh undergoing procedural deformation.
- Applying terrain height fields by using vertex texture fetch [40, 1227].

Some deformations done using a vertex shader are shown in [Figure 3.8](#).

The output of the vertex shader can be consumed in several different ways. The usual path is for each instance's primitives, e.g., triangles, to then be generated and rasterized, and the individual pixel fragments produced to be sent to the pixel shader program for continued processing. On some GPUs the data can also be sent to the tessellation stage or the geometry shader or be stored in memory. These optional stages are discussed in the following sections.



Figure 3.8. On the left, a normal teapot. A simple shear operation performed by a vertex shader program produces the middle image. On the right, a noise function creates a field that distorts the model. (Images produced by *FX Composer 2*, courtesy of *NVIDIA Corporation*.)

3.6 The Tessellation Stage

The tessellation stage allows us to render curved surfaces. The GPU’s task is to take each surface description and turn it into a representative set of triangles. This stage is an optional GPU feature that first became available in (and is required by) DirectX 11. It is also supported in OpenGL 4.0 and OpenGL ES 3.2.

There are several advantages to using the tessellation stage. The curved surface description is often more compact than providing the corresponding triangles themselves. Beyond memory savings, this feature can keep the bus between CPU and GPU from becoming the bottleneck for an animated character or object whose shape is changing each frame. The surfaces can be rendered efficiently by having an appropriate number of triangles generated for the given view. For example, if a ball is far from the camera, only a few triangles are needed. Up close, it may look best represented with thousands of triangles. This ability to control the *level of detail* can also allow an application to control its performance, e.g., using a lower-quality mesh on weaker GPUs in order to maintain frame rate. Models normally represented by flat surfaces can be converted to fine meshes of triangles and then warped as desired [1493], or they can be tessellated in order to perform expensive shading computations less frequently [225].

The tessellation stage always consists of three elements. Using DirectX’s terminology, these are the *hull shader*, *tessellator*, and *domain shader*. In OpenGL the hull shader is the *tessellation control shader* and the domain shader the *tessellation evaluation shader*, which are a bit more descriptive, though verbose. The fixed-function tessellator is called the *primitive generator* in OpenGL, and as will be seen, that is indeed what it does.

How to specify and tessellate curves and surfaces is discussed at length in [Chapter 17](#). Here we give a brief summary of each tessellation stage’s purpose. To begin, the input to the hull shader is a special *patch* primitive. This consists of several control points defining a subdivision surface, Bézier patch, or other type of curved element. The hull shader has two functions. First, it tells the tessellator how many triangles should be generated, and in what configuration. Second, it performs processing on each of the control points. Also, optionally, the hull shader can modify the incoming

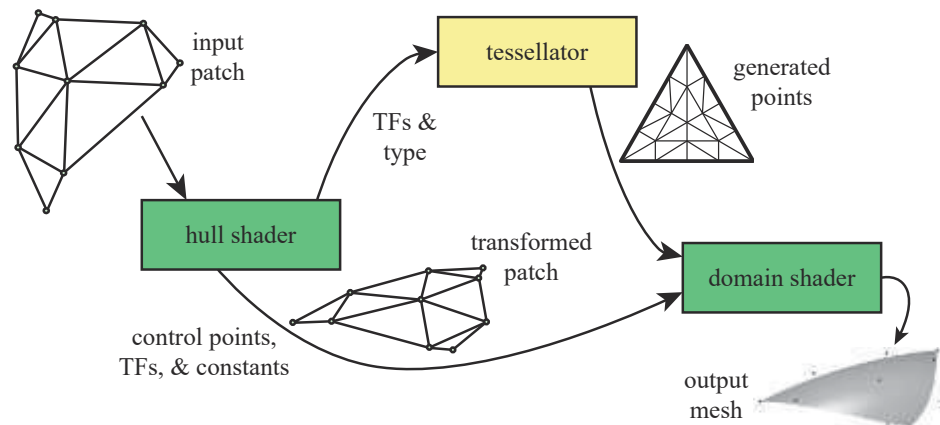


Figure 3.9. The tessellation stage. The hull shader takes in a patch defined by control points. It sends the tessellation factors (TFs) and type to the fixed-function tessellator. The control point set is transformed as desired by the hull shader and sent on to the domain shader, along with TFs and related patch constants. The tessellator creates the set of vertices along with their barycentric coordinates. These are then processed by the domain shader, producing the triangle mesh (control points shown for reference).

patch description, adding or removing control points as desired. The hull shader outputs its set of control points, along with the tessellation control data, to the domain shader. See [Figure 3.9](#).

The tessellator is a fixed-function stage in the pipeline, only used with tessellation shaders. It has the task of adding several new vertices for the domain shader to process. The hull shader sends the tessellator information about what type of tessellation surface is desired: triangle, quadrilateral, or isoline. Isolines are sets of line strips, sometimes used for hair rendering [1954]. The other important values sent by the hull shader are the tessellation factors (*tessellation levels* in OpenGL). These are of two types: inner and outer edge. The two inner factors determine how much tessellation occurs inside the triangle or quadrilateral. The outer factors determine how much each exterior edge is split ([Section 17.6](#)). An example of increasing tessellation factors is shown in [Figure 3.10](#). By allowing separate controls, we can have adjacent curved surfaces' edges match in tessellation, regardless of how the interiors are tessellated. Matching edges avoids cracks or other shading artifacts where patches meet. The vertices are assigned barycentric coordinates ([Section 22.8](#)), which are values that specify a relative location for each point on the desired surface.

The hull shader always outputs a patch, a set of control point locations. However, it can signal that a patch is to be discarded by sending the tessellator an outer tessellation level of zero or less (or not-a-number, NaN). Otherwise, the tessellator generates a mesh and sends it to the domain shader. The control points for the curved surface from the hull shader are used by each invocation of the domain shader to compute the



Figure 3.10. The effect of varying the tessellation factors. The Utah teapot is made of 32 patches. Inner and outer tessellation factors, from left to right, are 1, 2, 4, and 8. (Images generated by demo from Rideout and Van Gelder [1493].)

output values for each vertex. The domain shader has a data flow pattern like that of a vertex shader, with each input vertex from the tessellator being processed and generating a corresponding output vertex. The triangles formed are then passed on down the pipeline.

While this system sounds complex, it is structured this way for efficiency, and each shader can be fairly simple. The patch passed into a hull shader will often undergo little or no modification. This shader may also use the patch's estimated distance or screen size to compute tessellation factors on the fly, as for terrain rendering [466]. Alternately, the hull shader may simply pass on a fixed set of values for all patches that the application computes and provides. The tessellator performs an involved but fixed-function process of generating the vertices, giving them positions, and specifying what triangles or lines they form. This data amplification step is performed outside of a shader for computational efficiency [530]. The domain shader takes the barycentric coordinates generated for each point and uses these in the patch's evaluation equation to generate the position, normal, texture coordinates, and other vertex information desired. See Figure 3.11 for an example.

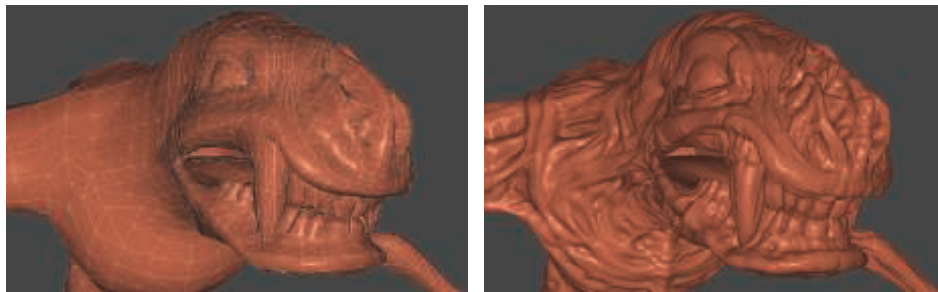


Figure 3.11. On the left is the underlying mesh of about 6000 triangles. On the right, each triangle is tessellated and displaced using PN triangle subdivision. (Images from NVIDIA SDK 11 [1301] samples, courtesy of NVIDIA Corporation, model from Metro 2033 by 4A Games.)

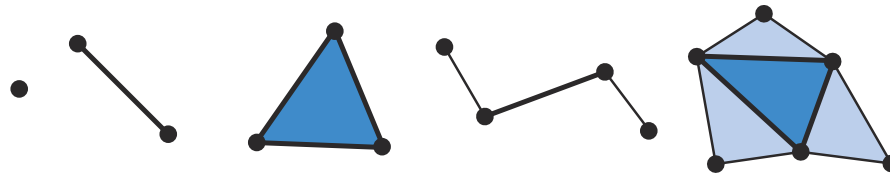


Figure 3.12. Geometry shader input for a geometry shader program is of some single type: point, line segment, triangle. The two rightmost primitives include vertices adjacent to the line and triangle objects. More elaborate patch types are possible.

3.7 The Geometry Shader

The geometry shader can turn primitives into other primitives, something the tessellation stage cannot do. For example, a triangle mesh could be transformed to a wireframe view by having each triangle create line edges. Alternately, the lines could be replaced by quadrilaterals facing the viewer, so making a wireframe rendering with thicker edges [1492]. The geometry shader was added to the hardware-accelerated graphics pipeline with the release of DirectX 10, in late 2006. It is located after the tessellation shader in the pipeline, and its use is optional. While a required part of Shader Model 4.0, it is not used in earlier shader models. OpenGL 3.2 and OpenGL ES 3.2 support this type of shader as well.

The input to the geometry shader is a single object and its associated vertices. The object typically consists of triangles in a strip, a line segment, or simply a point. Extended primitives can be defined and processed by the geometry shader. In particular, three additional vertices outside of a triangle can be passed in, and the two adjacent vertices on a polyline can be used. See Figure 3.12. With DirectX 11 and Shader Model 5.0, you can pass in more elaborate patches, with up to 32 control points. That said, the tessellation stage is more efficient for patch generation [175].

The geometry shader processes this primitive and outputs zero or more vertices, which are treated as points, polylines, or strips of triangles. Note that no output at all can be generated by the geometry shader. In this way, a mesh can be selectively modified by editing vertices, adding new primitives, and removing others.

The geometry shader is designed for modifying incoming data or making a limited number of copies. For example, one use is to generate six transformed copies of data to simultaneously render the six faces of a cube map; see Section 10.4.3. It can also be used to efficiently create cascaded shadow maps for high-quality shadow generation. Other algorithms that take advantage of the geometry shader include creating variable-sized particles from point data, extruding fins along silhouettes for fur rendering, and finding object edges for shadow algorithms. See Figure 3.13 for more examples. These and other uses are discussed throughout the rest of the book.

DirectX 11 added the ability for the geometry shader to use instancing, where the geometry shader can be run a set number of times on any given primitive [530, 1971]. In

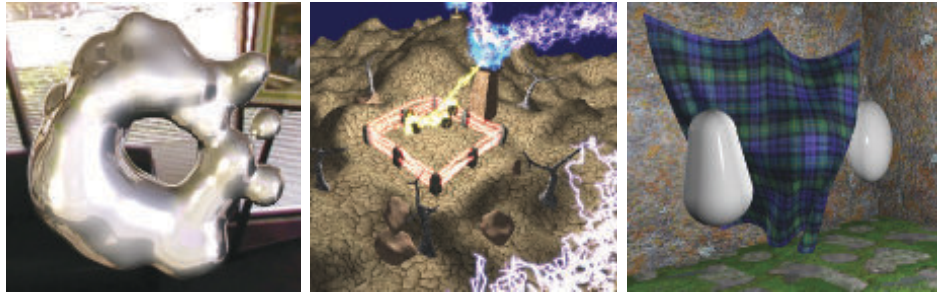


Figure 3.13. Some uses of the geometry shader (GS). On the left, metaball isosurface tessellation is performed on the fly using the GS. In the middle, fractal subdivision of line segments is done using the GS and stream out, and billboards are generated by the GS for display of the lightning. On the right, cloth simulation is performed by using the vertex and geometry shader with stream out. (Images from NVIDIA SDK 10 [1300] samples, courtesy of NVIDIA Corporation.)

OpenGL 4.0 this is specified with an invocation count. The geometry shader can also output up to four *streams*. One stream can be sent on down the rendering pipeline for further processing. All these streams can optionally be sent to stream output render targets.

The geometry shader is guaranteed to output results from primitives in the same order that they are input. This affects performance, because if several shader cores run in parallel, results must be saved and ordered. This and other factors work against the geometry shader being used to replicate or create a large amount of geometry in a single call [175, 530].

After a draw call is issued, there are only three places in the pipeline where work can be created on the GPU: rasterization, the tessellation stage, and the geometry shader. Of these, the geometry shader's behavior is the least predictable when considering resources and memory needed, since it is fully programmable. In practice the geometry shader usually sees little use, as it does not map well to the GPU's strengths. On some mobile devices it is implemented in software, so its use is actively discouraged there [69].

3.7.1 Stream Output

The standard use of the GPU's pipeline is to send data through the vertex shader, then rasterize the resulting triangles and process these in the pixel shader. It used to be that the data always passed through the pipeline and intermediate results could not be accessed. The idea of *stream output* was introduced in Shader Model 4.0. After vertices are processed by the vertex shader (and, optionally, the tessellation and geometry shaders), these can be output in a stream, i.e., an ordered array, in addition to being sent on to the rasterization stage. Rasterization could, in fact, be turned off entirely and the pipeline then used purely as a non-graphical stream processor. Data

processed in this way can be sent back through the pipeline, thus allowing iterative processing. This type of operation can be useful for simulating flowing water or other particle effects, as discussed in [Section 13.8](#). It could also be used to skin a model and then have these vertices available for reuse ([Section 4.4](#)).

Stream output returns data only in the form of floating point numbers, so it can have a noticeable memory cost. Stream output works on primitives, not directly on vertices. If meshes are sent down the pipeline, each triangle generates its own set of three output vertices. Any vertex sharing in the original mesh is lost. For this reason a more typical use is to send just the vertices through the pipeline as a point set primitive. In OpenGL the stream output stage is called *transform feedback*, since the focus of much of its use is transforming vertices and returning them for further processing. Primitives are guaranteed to be sent to the stream output target in the order that they were input, meaning the vertex order will be maintained [530].

3.8 The Pixel Shader

After the vertex, tessellation, and geometry shaders perform their operations, the primitive is clipped and set up for rasterization, as explained in the previous chapter. This section of the pipeline is relatively fixed in its processing steps, i.e., not programmable but somewhat configurable. Each triangle is traversed to determine which pixels it covers. The rasterizer may also roughly calculate how much the triangle covers each pixel's cell area ([Section 5.4.2](#)). This piece of a triangle partially or fully overlapping the pixel is called a *fragment*.

The values at the triangle's vertices, including the z -value used in the z -buffer, are interpolated across the triangle's surface for each pixel. These values are passed to the *pixel shader*, which then processes the fragment. In OpenGL the pixel shader is known as the *fragment shader*, which is perhaps a better name. We use "pixel shader" throughout this book for consistency. Point and line primitives sent down the pipeline also create fragments for the pixels covered.

The type of interpolation performed across the triangle is specified by the pixel shader program. Normally we use perspective-correct interpolation, so that the world-space distances between pixel surface locations increase as an object recedes in the distance. An example is rendering railroad tracks extending to the horizon. Railroad ties are more closely spaced where the rails are farther away, as more distance is traveled for each successive pixel approaching the horizon. Other interpolation options are available, such as screen-space interpolation, where perspective projection is not taken into account. DirectX 11 gives further control over when and how interpolation is performed [530].

In programming terms, the vertex shader program's outputs, interpolated across the triangle (or line), effectively become the pixel shader program's inputs. As the GPU has evolved, other inputs have been exposed. For example, the screen position of the fragment is available to the pixel shader in Shader Model 3.0 and beyond. Also,

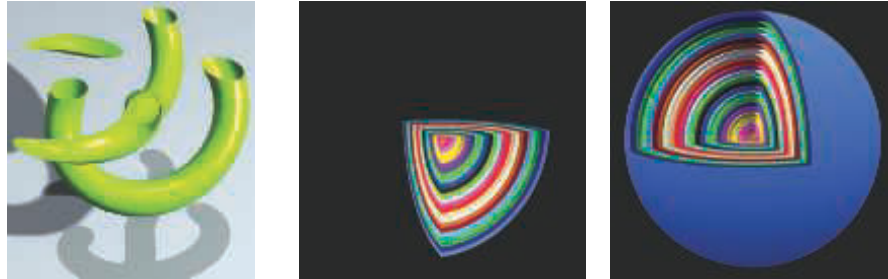


Figure 3.14. User-defined clipping planes. On the left, a single horizontal clipping plane slices the object. In the middle, the nested spheres are clipped by three planes. On the right, the spheres' surfaces are clipped only if they are outside all three clip planes. (From the *three.js* examples *webgl_clipping* and *webgl_clipping_intersection* [218].)

which side of a triangle is visible is an input flag. This knowledge is important for rendering a different material on the front versus back of each triangle in a single pass.

With inputs in hand, typically the pixel shader computes and outputs a fragment's color. It can also possibly produce an opacity value and optionally modify its z -depth. During merging, these values are used to modify what is stored at the pixel. The depth value generated in the rasterization stage can also be modified by the pixel shader. The stencil buffer value is usually not modifiable, but rather it is passed through to the merge stage. DirectX 11.3 allows the shader to change this value. Operations such as fog computation and alpha testing have moved from being merge operations to being pixel shader computations in SM 4.0 [175].

A pixel shader also has the unique ability to discard an incoming fragment, i.e., generate no output. One example of how fragment discard can be used is shown in Figure 3.14. Clip plane functionality used to be a configurable element in the fixed-function pipeline and was later specified in the vertex shader. With fragment discard available, this functionality could then be implemented in any way desired in the pixel shader, such as deciding whether clipping volumes should be AND'ed or OR'ed together.

Initially the pixel shader could output to only the merging stage, for eventual display. The number of instructions a pixel shader can execute has grown considerably over time. This increase gave rise to the idea of *multiple render targets* (MRT). Instead of sending results of a pixel shader's program to just the color and z -buffer, multiple sets of values could be generated for each fragment and saved to different buffers, each called a *render target*. Render targets generally have the same x - and y -dimensions; some APIs allow different sizes, but the rendered area will be the smallest of these. Some architectures require render targets to each have the same bit depth, and possibly even identical data formats. Depending on the GPU, the number of render targets available is four or eight.

Even with these limitations, MRT functionality is a powerful aid in performing rendering algorithms more efficiently. A single rendering pass could generate a color image in one target, object identifiers in another, and world-space distances in a third. This ability has also given rise to a different type of rendering pipeline, called *deferred shading*, where visibility and shading are done in separate passes. The first pass stores data about an object's location and material at each pixel. Successive passes can then efficiently apply illumination and other effects. This class of rendering methods is described in [Section 20.1](#).

The pixel shader's limitation is that it can normally write to a render target at only the fragment location handed to it, and cannot read current results from neighboring pixels. That is, when a pixel shader program executes, it cannot send its output directly to neighboring pixels, nor can it access others' recent changes. Rather, it computes results that affect only its own pixel. However, this limitation is not as severe as it sounds. An output image created in one pass can have any of its data accessed by a pixel shader in a later pass. Neighboring pixels can be processed using image processing techniques, described in [Section 12.1](#).

There are exceptions to the rule that a pixel shader cannot know or affect neighboring pixels' results. One is that the pixel shader can immediately access information for adjacent fragments (albeit indirectly) during the computation of gradient or derivative information. The pixel shader is provided with the amounts by which any interpolated value changes per pixel along the x and y screen axes. Such values are useful for various computations and texture addressing. These gradients are particularly important for operations such as texture filtering ([Section 6.2.2](#)), where we want to know how much of an image covers a pixel. All modern GPUs implement this feature by processing fragments in groups of 2×2 , called a *quad*. When the pixel shader requests a gradient value, the difference between adjacent fragments is returned. See [Figure 3.15](#). A unified core has this capability to access neighboring data—kept in different threads on the same warp—and so can compute gradients for use in the pixel shader. One consequence of this implementation is that gradient information cannot be accessed in parts of the shader affected by dynamic flow control, i.e., an “if” statement or loop with a variable number of iterations. All the fragments in a group must be processed using the same set of instructions so that all four pixels' results are meaningful for computing gradients. This is a fundamental limitation that exists even in offline rendering systems [\[64\]](#).

DirectX 11 introduced a buffer type that allows write access to any location, the *unordered access view* (UAV). Originally for only pixel and compute shaders, access to UAVs was extended to all shaders in DirectX 11.1 [\[146\]](#). OpenGL 4.3 calls this a *shader storage buffer object* (SSBO). Both names are descriptive in their own way. Pixel shaders are run in parallel, in an arbitrary order, and this storage buffer is shared among them.

Often some mechanism is needed to avoid a *data race condition* (a.k.a. a *data hazard*), where both shader programs are “racing” to influence the same value, possibly

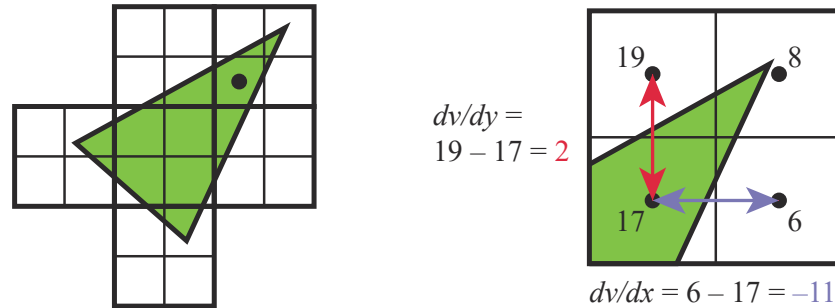


Figure 3.15. On the left, a triangle is rasterized into quads, sets of 2×2 pixels. The gradient computations for the pixel marked with a black dot is then shown on the right. The value for v is shown for each of the four pixel locations in the quad. Note how three of the pixels are not covered by the triangle, yet they are still processed by the GPU so that the gradients can be found. The gradients in the x and y screen directions are computed for the lower left pixel by using its two quad neighbors.

leading to arbitrary results. As an example, an error could occur if two invocations of a pixel shader tried to, say, add to the same retrieved value at about the same time. Both would retrieve the original value, both would modify it locally, but then whichever invocation wrote its result last would wipe out the contribution of the other invocation—only one addition would occur. GPUs avoid this problem by having dedicated atomic units that the shader can access [530]. However, atomics mean that some shaders may stall as they wait to access a memory location undergoing read/modify/write by another shader.

While atomics avoid data hazards, many algorithms require a specific order of execution. For example, you may want to draw a more distant transparent blue triangle before overlaying it with a red transparent triangle, blending the red atop the blue. It is possible for a pixel to have two pixel shader invocations for a pixel, one for each triangle, executing in such a way that the red triangle's shader completes before the blue's. In the standard pipeline, the fragment results are sorted in the merger stage before being processed. *Rasterizer order views* (ROVs) were introduced in DirectX 11.3 to enforce an order of execution. These are like UAVs; they can be read and written by shaders in the same fashion. The key difference is that ROVs guarantee that the data are accessed in the proper order. This increases the usefulness of these shader-accessible buffers considerably [327, 328]. For example, ROVs make it possible for the pixel shader to write its own blending methods, since it can directly access and write to any location in the ROV, and thus no merging stage is needed [176]. The price is that, if an out-of-order access is detected, a pixel shader invocation may stall until triangles drawn earlier are processed.

3.9 The Merging Stage

As discussed in [Section 2.5.2](#), the merging stage is where the depths and colors of the individual fragments (generated in the pixel shader) are combined with the frame-buffer. DirectX calls this stage the *output merger*; OpenGL refers to it as *per-sample operations*. On most traditional pipeline diagrams (including our own), this stage is where stencil-buffer and *z*-buffer operations occur. If the fragment is visible, another operation that takes place in this stage is color blending. For opaque surfaces there is no real blending involved, as the fragment's color simply replaces the previously stored color. Actual blending of the fragment and stored color is commonly used for transparency and compositing operations ([Section 5.5](#)).

Imagine that a fragment generated by rasterization is run through the pixel shader and then is found to be hidden by some previously rendered fragment when the *z*-buffer is applied. All the processing done in the pixel shader was then unnecessary. To avoid this waste, many GPUs perform some merge testing before the pixel shader is executed [530]. The fragment's *z*-depth (and whatever else is in use, such as the stencil buffer or scissoring) is used for testing visibility. The fragment is culled if hidden. This functionality is called *early-z* [1220, 1542]. The pixel shader has the ability to change the *z*-depth of the fragment or to discard the fragment entirely. If either type of operation is found to exist in a pixel shader program, *early-z* then generally cannot be used and is turned off, usually making the pipeline less efficient. DirectX 11 and OpenGL 4.2 allow the pixel shader to force *early-z* testing to be on, though with a number of limitations [530]. See [Section 23.7](#) for more about *early-z* and other *z*-buffer optimizations. Using *early-z* effectively can have a large effect on performance, which is discussed in detail in [Section 18.4.5](#).

The merging stage occupies the middle ground between fixed-function stages, such as triangle setup, and the fully programmable shader stages. Although it is not programmable, its operation is highly configurable. Color blending in particular can be set up to perform a large number of different operations. The most common are combinations of multiplication, addition, and subtraction involving the color and alpha values, but other operations are possible, such as minimum and maximum, as well as bitwise logic operations. DirectX 10 added the capability to blend two colors from the pixel shader with the framebuffer color. This capability is called *dual source-color blending* and cannot be used in conjunction with multiple render targets. MRT does otherwise support blending, and DirectX 10.1 introduced the capability to perform different blend operations on each separate buffer.

As mentioned at the end of the previous section, DirectX 11.3 provided a way to make blending programmable through ROVs, though at a price in performance. ROVs and the merging stage both guarantee draw order, a.k.a. output invariance. Regardless of the order in which pixel shader results are generated, it is an API requirement that results are sorted and sent to the merging stage in the order in which they are input, object by object and triangle by triangle.

3.10 The Compute Shader

The GPU can be used for more than implementing the traditional graphics pipeline. There are many non-graphical uses in fields as varied as computing the estimated value of stock options and training neural nets for deep learning. Using hardware in this way is called *GPU computing*. Platforms such as CUDA and OpenCL are used to control the GPU as a massive parallel processor, with no real need or access to graphics-specific functionality. These frameworks often use languages such as C or C++ with extensions, along with libraries made for the GPU.

Introduced in DirectX 11, the *compute shader* is a form of GPU computing, in that it is a shader that is not locked into a location in the graphics pipeline. It is closely tied to the process of rendering in that it is invoked by the graphics API. It is used alongside vertex, pixel, and other shaders. It draws upon the same pool of unified shader processors as those used in the pipeline. It is a shader like the others, in that it has some set of input data and can access buffers (such as textures) for input and output. Warps and threads are more visible in a compute shader. For example, each invocation gets a thread index that it can access. There is also the concept of a *thread group*, which consists of 1 to 1024 threads in DirectX 11. These thread groups are specified by x -, y -, and z -coordinates, mostly for simplicity of use in shader code. Each thread group has a small amount of memory that is shared among threads. In DirectX 11, this amounts to 32 kB. Compute shaders are executed by thread group, so that all threads in the group are guaranteed to run concurrently [1971].

One important advantage of compute shaders is that they can access data generated on the GPU. Sending data from the GPU to the CPU incurs a delay, so performance can be improved if processing and results can be kept resident on the GPU [1403]. Post-processing, where a rendered image is modified in some way, is a common use of compute shaders. The shared memory means that intermediate results from sampling image pixels can be shared with neighboring threads. Using a compute shader to determine the distribution or average luminance of an image, for example, has been found to run twice as fast as performing this operation on a pixel shader [530].

Compute shaders are also useful for particle systems, mesh processing such as facial animation [134], culling [1883, 1884], image filtering [1102, 1710], improving depth precision [991], shadows [865], depth of field [764], and any other tasks where a set of GPU processors can be brought to bear. Wihlidal [1884] discusses how compute shaders can be more efficient than tessellation hull shaders. See Figure 3.16 for other uses.

This ends our review of the GPU's implementation of the rendering pipeline. There are many ways in which the GPU's functions can be used and combined to perform various rendering-related processes. Relevant theory and algorithms tuned to take advantage of these capabilities are the central subjects of this book. Our focus now moves on to transforms and shading.



Figure 3.16. Compute shader examples. On the left, a compute shader is used to simulate hair affected by wind, with the hair itself rendered using the tessellation stage. In the middle, a compute shader performs a rapid blur operation. On the right, ocean waves are simulated. (Images from *NVIDIA SDK 11* [1301] samples, courtesy of NVIDIA Corporation.)

Further Reading and Resources

Giesen’s tour of the graphics pipeline [530] discusses many facets of the GPU at length, explaining why elements work the way they do. The course by Fatahalian and Bryant [462] discusses GPU parallelism in a series of detailed lecture slide sets. While focused on GPU computing using CUDA, the introductory part of Kirk and Hwa’s book [903] discusses the evolution and design philosophy for the GPU.

To learn the formal aspects of shader programming takes some work. Books such as the *OpenGL Superbible* [1606] and *OpenGL Programming Guide* [885] include material on shader programming. The older book *OpenGL Shading Language* [1512] does not cover more recent shader stages, such as the geometry and tessellation shaders, but does focus specifically on shader-related algorithms. See this book’s website, realtimerendering.com, for recent and recommended books.