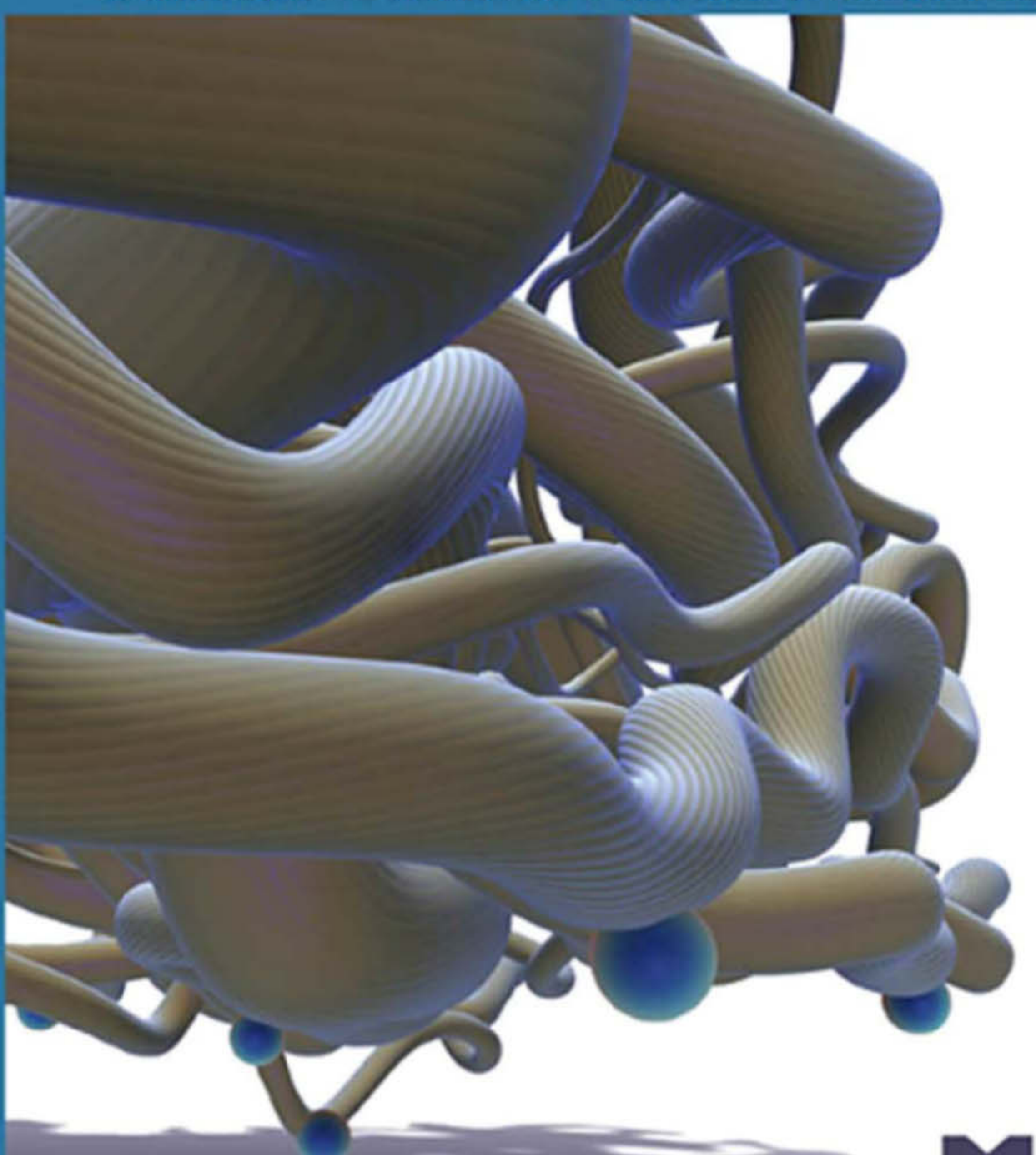
in silico

3D ANIMATION AND SIMULATION OF CELL BIOLOGY WITH MAYA AND MEL



Jason Sharpe Charles J Lumsden Nicholas Woolridge

MK MORGAN KAUFMANN

In Silico: 3D Animation and Simulation of Cell Biology with Maya and MEL

This page intentionally left blank

In Silico: 3D Animation and Simulation of Cell Biology with Maya and MEL

Jason Sharpe
AXS Biomedical Animation Studio

Charles John Lumsden
University of Toronto

Nicholas Woolridge
University of Toronto

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO
Morgan Kaufmann Publishers is an imprint of Elsevier

Acquisitions Editor: Tiffany Gasbarrini
Publishing Services Manager: George Morrison
Project Manager: Mónica González de Mendoza
Assistant Editor: Matt Cater
Cover Design: Jason Sharpe / Alisa Andreola
Cover Illustration: Jason Sharpe

Morgan Kaufmann Publishers is an imprint of Elsevier.
30 Corporate Drive, Suite 400, Burlington, MA 01803, USA

This book is printed on acid-free paper.

© 2008 Jason Sharpe, Charles Lumsden, Nicholas Woolridge. Published by Elsevier, Inc. All rights reserved.

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances in which Morgan Kaufmann Publishers is aware of a claim, the product names appear in initial capital or all capital letters. All trademarks that appear or are otherwise referred to in this work belong to their respective owners. Neither Morgan Kaufmann Publishers nor the authors and other contributors of this work have any relationship or affiliation with such trademark owners nor do such trademark owners confirm, endorse or approve the contents of this work. Readers, however, should contact the appropriate companies for more information regarding trademarks and any related registrations.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, scanning, or otherwise—without prior written permission of the publisher.

All images © the authors unless otherwise stated in the text. Certain images and materials contained in this publication were reproduced with the permission of Autodesk, Inc. © 2007. All rights reserved. Autodesk and Maya are registered trademarks of Autodesk, Inc., in the U.S.A. and certain other countries.

The information in this book and accompanying CD-ROM disk is distributed on an "as is" basis, without warranty. Although due precaution has been taken in the preparation of this work, neither the authors nor the publisher shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this book and accompanying CD-ROM disk, including, without limitation, any software, whether in object code or source code format.

Permissions may be sought directly from Elsevier's Science & Technology Rights Department in Oxford, UK: phone: (+44) 1865 843830, fax: (+44) 1865 853333, E-mail: permissions@elsevier.com. You may also complete your request online via the Elsevier homepage (http://elsevier.com), by selecting

"Support & Contact" then "Copyright and Permission" and then "Obtaining Permissions."

Library of Congress Cataloging-in-Publication Data
Sharpe, Jason.
In Silico: 3D Animation and Simulation of Cell Biology with Maya and MEL / Jason Sharpe, Charles John Lumsden, Nicholas Woolridge.
p. ; cm.
Includes bibliographical references and index.
ISBN-13: 978-0-12-373655-0 (pbk. : alk. paper) 1. Cytology—Computer simulation. 2. Maya (Computer file) 3. Computer animation. 4. Computer graphics. 5. Three-dimensional display systems. I. Lumsden, Charles J., 1949– II. Woolridge, Nicholas. III. Title. IV. Title: Cell biology art and science with Maya and MEL.
[DNLM: 1. Cells—Programmed Instruction. 2. Computational Biology—Programmed Instruction. 3. Models, Biological—Programmed Instruction. 4. Motion Pictures as Topic—Programmed Instruction. 5. Programming Languages—Programmed Instruction. QU 18.2 S532s 2008]
QH585.5.D38S53 2008
571.601′13—dc22

2007053013

ISBN: 978-0-12-373655-0

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

08 09 10 11 12 13 10 9 8 7 6 5 4 3 2 1

Printed in China

Working together to grow
libraries in developing countries

www.elsevier.com | www.bookaid.org | www.sabre.org

ELSEVIER BOOK AID International Sabre Foundation

CONTENTS

Preface xiii

Who is this book for? xiv
Why Maya? xiv
What the book offers xv
Computer hardware and software xxi
About the authors xxii
Acknowledgments xxiii

Part 1 Setting the stage 1

01 Introduction 3

The challenge 4
Wetware for seeing 5
Visualization in science 6
Organizational hierarchy: Keys to biology in vivo and in silico 8
Enter Maya 13
Endless possibilities 19
References 19

02 Computers and the organism 21

Introduction 22
Information and process 22
Language and program 23
High and low 26
Interpret or compile? 27
The Backus watershed 28
Stored programs 30

Conditional control 33
The computed organism 35
The computational organism 36
OOPs and agents 39
Summary 41
References 43

03 Animating biology 45

Introduction 46
Animation and film perception 46
The animator's workflow 49
The three-stage workflow 51
Putting it all together 67
References 67

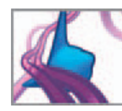
Part 2 A foundation in Maya 69

04 Maya basics 71

Getting started 72
How Maya works (briefly) 78
Maya's UI 82
Summary 99

05 Modeling geometry 101

Introduction 102
NURBS modeling 103
Polygonal modeling 107
Tutorial 05.01: NURBS primitive modeling 109
Tutorial 05.02: Deform the sphere using components 117
Tutorial 05.03: Make and deform a polygon primitive 119
Tutorial 05.04: Construction history 122



Tutorial 05.05: Create a NURBS “fiber” 129

Summary 134

References 135

06 Animation 137

Introduction 138

Animation 138

Tutorial 06.01: A keyframe animation 145

Animation nodes in the Hypergraph and Attribute Editor 151

Tutorial 06.02: A simple procedural animation 151

Summary 154

07 Dynamics 157

Introduction 158

The Dynamics module 160

Tutorial 07.01: Rigid body dynamics 166

Tutorial 07.02: Particles in a container 173

Tutorial 07.03: Create a playblast 184

Summary 185

08 Shading 187

Introduction 188

The Render menu set 190

Shading 191

Tutorial 08.01: Shading 203

Summary 214

09 Cameras 215

Maya Cameras 217

Tutorial 09.01: A camera on hemoglobin 222

Summary 230

10 Lighting 231

Lighting 232

Tutorial 10.01: Lighting the hemoglobin scene 235

Summary 241

11 Action! Maya rendering 243

Rendering 244

Advanced rendering techniques with the mental ray for Maya renderer 249

Tutorial 11.01: Batch rendering 252

Tutorial 11.02: Playback using fCheck 257

Summary 259

12 MEL scripting 261

Introduction 262

The origins of MEL 263

In a word: *Scripting* 264

Getting started 266

MEL syntax 269

Values 270

Variables 271

Mathematical and logical expressions 277

The MEL command 280

Attributes in MEL 286

Conditional statements 288

Loops 289

Procedures 291

Animation expressions 292

Putting it all together: The MEL script 301

Tutorial 12.01: Building a MEL script 302

Debugging your scripts 306
Random number generation in Maya 308
Summary 309

13 Data input/output 311

Introduction 312
Translators 313
Reading and writing files with MEL 315
Tutorial 13.01: Visualizing cell migration 322
Summary 337

Part 3 Biology in silico—Maya in action 339

14 Building a protein 341

Introduction 342
Problem overview 346
Methods: Algorithm design 354
Methods: Encoding the algorithm 354
Results: Running the script 368
Results: Rendering your molecule 372
Summary 380
References 381

15 Self-assembly 383

Introduction 384
Problem overview 385
Methods: Actin geometry 394
Methods: Diffusion and reaction events 399
Methods: Reaction rates and probabilities 403
Methods: Algorithm design 409

Methods: Encoding the algorithm 412
Results: Running your simulation 437
Summary 441
References 442

16 Modeling a mobile cell 443

Introduction 444
Problem overview 445
Model definition 449
Methods: Generating pseudopods 451
Methods: Algorithm design 453
Methods: A cell locomotion engine 454
Methods: Encoding the algorithm 466
Methods: Loading the script 475
Results: Running the script 476
Summary 477
References 477

17 Growing an ECM scaffold 479

Introduction 480
Problem overview 481
Model definition 483
Methods: Algorithm design 486
Methods: Encoding the algorithm 494
Methods: Grow your scaffold! 512
Results: Parameter effects 516
Summary 517
References 517

18 Scaffold invasions: Modeling 3D populations of mobile cells 519

Introduction 520

Problem overview 521

Model definition 525

Methods: Model design 528

Methods: Encoding the algorithm 538

Methods: Running the simulation 565

Results: Data output 572

Summary 573

References 573

19 Conclusion: A new kind of seeing 575

Explanations, simulations, speculations 576

Maya's role 578

The path so far 578

The future 579

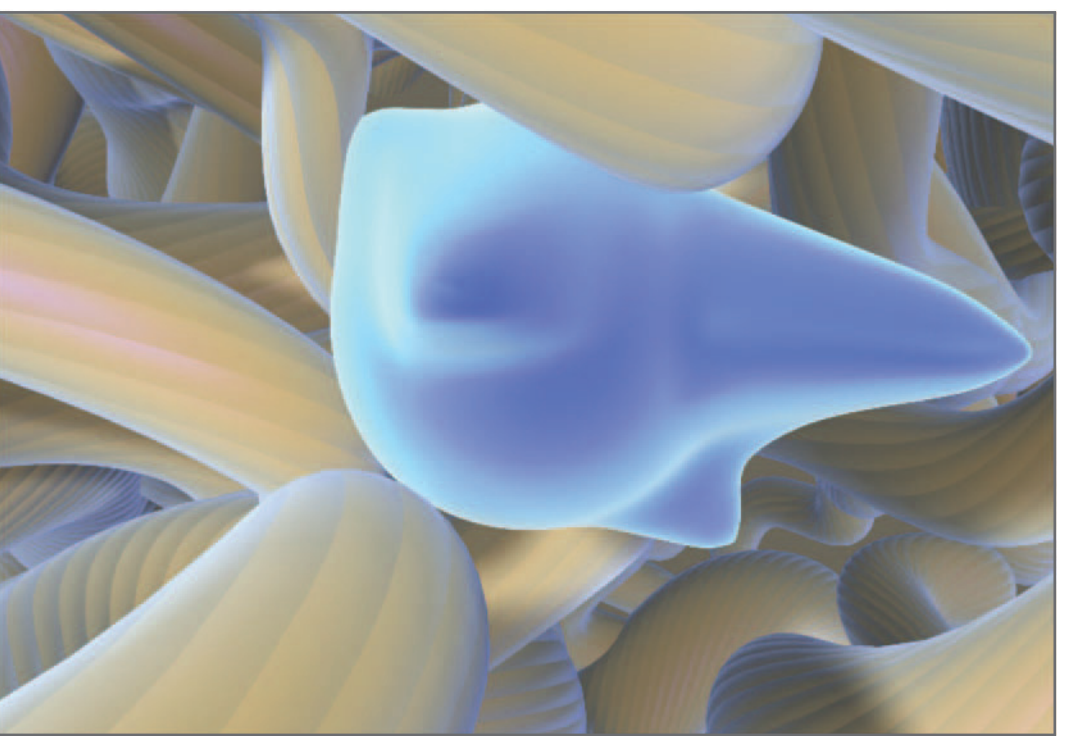
References 582

Further reading 585

Glossary 593

Index 607

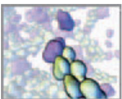
This page intentionally left blank



Still image from a Maya simulation model of cell migration in a 3D scaffold. The cell extends protrusions in search of scaffold fibers. When it contacts a fiber, the protrusion adheres to it. The cell body then contracts, pulling it in the direction of the adhesion. Maya's extensive 3D modeling toolset and programming capabilities make it well suited to 3D visual simulations of biological phenomena such as cell migration.

Courtesy and © 2006 Donald Ly.

Preface



Who is this book for?

If, like us, you are involved with the study of cells and cell biology, or if your work takes inspiration from the organic world, this book is for you. We have written *In Silico* for the diverse creative community—scientists, artists, media designers, students, and hobbyists—now deeply involved with the living cell as a key to unlocking the complexity of organic matter and a gateway to powerful new understanding of disease. In the scientific area, cell and molecular biologists and their research partners today have little time to spare developing complex computer programs from the ground up. High-end three-dimensional (3D) computer programs like Autodesk Maya provide the busy scientist with a robust, flexible development environment in which state-of-the-art computer methods can be used to analyze, model, and visualize cell data. Equipped with deeply customizable user and application programming interfaces, Maya and other top-tier 3D animation programs afford rapid prototyping of data analysis and models through advanced graphics, physics, and rendering systems. Output capability embraces both crisp numerical data and polished 3D dynamic visualizations of cell physiology. These tools have enough programming flexibility that the working researcher can concentrate on the functional aspects of the data mapping or simulation capability they wish to create.

In the communications field are individuals and groups immersed in the burgeoning marketplace of biocommunications, especially medical and scientific animation. The telling of stories is a human universal, common to all peoples and cultures. The increasingly complex world enabled by science and technology makes the accurate, compelling telling of scientific stories more important than ever. Constantly, animators of medical and scientific subjects are called on to present ever more intricate, unusual phenomena involved in understanding how cells work and what goes wrong with them to cause devastating illnesses like cancer and heart disease. At the same time, the expectations of a media-savvy public for concise, truthful, entertaining visual stories rise even higher. Taking control of a program like Maya can empower the media artist to better interpret and visualize wonderfully intricate cellular phenomena—such as the crowded molecular landscapes of the cell interior, the cell waves coursing through the embryo's interior, or the skein of blood vessels healing a wound—that would be impractically tedious or impossible to animate by hand.

And too numerous to count, surely, are the artists and citizens everywhere who draw inspiration from biology and the natural world, and who dream of imparting some facet of organic vitality and complexity to their creative work or personal appreciation of nature. The ideas and methods of this book will, we believe, inform and inspire everyone with such interests. Although the focus of our applications is the exciting realm of the living cell, those whose interests embrace other parts of living nature will find the knowledge and techniques they learn here of useful in many different ways.

Why Maya?

Although Maya is a top-tier product used worldwide for 3D animation in entertainment, gaming, and manufacturing, this Academy Award® winning program does not stand alone in representing the cutting edge of high-end 3D. Superb tools such as SoftImage XSI, Maxon Cinema 4D, NewTek LightWave 3D, Autodesk 3ds Max, and

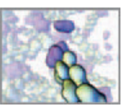
Side Effects Software's Houdini, stand alongside Maya to define the state of the art in 3D animation capability. Maya is our subject in this book for three reasons. First, despite the excellence of alternative tools Maya currently enjoys a pre-eminent status in top-end 3D animation work. Second, the Maya programming interfaces—accessed through a C++ application toolset (the API—which we plan to deal with in a subsequent book), via scripting in the Python language, and through Maya's own scripting language MEL, which we treat in this book—allow enormous power and flexibility in customizing Maya for scientific applications. Third, the academic outreach initiatives supported by Autodesk, the firm that makes and sells Maya, have enabled us to test Maya and some of its predecessors (such as Alias PowerAnimator) in demanding real-world science projects in cell and medical science. As a threesome, we have between us accumulated roughly 40 person-years of experience across a wide range of such applications. We find Maya worthy of close attention whenever there is a need to model and visualize 3D cell biology using a computer. Since our origins trace back to the early days, in which such computer methods were lab-written custom jobs in languages like Fortran, C++, and OpenGL, Maya for us means shorter time to software completion while increasing the power of the animated visualization.

If you are already a user of a 3D animation package other than Maya, you will still find considerable useful material in the pages to follow. The book is going to show you how to approach complex biological problems effectively, by means of a workflow in 3D visual computing. We have developed this workflow over the years of our medical and biocommunications research and use it daily in our teaching and scientific investigation. By working through the book's projects and case studies, you will be able to adapt our workflow to other 3D animation products as well as take them much further in Maya itself.

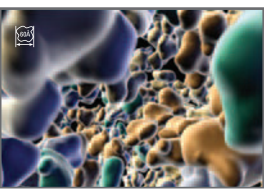
What the book offers

In the world of computer graphics software, Maya is a relatively complicated application. Learning and, eventually, some degree of genuine mastery, take time, but don't despair. Page by page, the learning map we have set up will take you from one productive result to the next. You will deal throughout with learning content that has genuine interest and significance in the world of science and cell biology. In *Part 1* you will meet the key ideas and terms from scientific computer graphics needed to dive into Maya while assessing its historic relevance to leading edge visualization. In *Part 2*, you will receive a self-contained introduction to Maya and to our workflow that will take you from starting the program through to a polished animation rendering of a complex protein. With this foundation you are ready to meet MEL, the programming language by which you will harness Maya's ability to model and render complex events. Then in *Part 3*, we put this all to work. You will develop a portfolio of case studies ranging from the single biological molecule to populations of interacting macromolecules, and then on to mobile cells as they move through their tissue environment. As you complete each element in the portfolio, you will have taken command of powerful new strategies for using MEL to control Maya's numerical and visual rendering activity.

Here's what you can expect in the rest of the book.

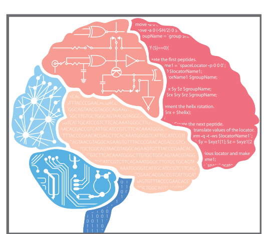


Part 1: Setting the stage



01 Introduction

To get started, we attempt to answer the question: "Why visualize?" We briefly discuss the power of visual perception in human learning and discovery, and how we can leverage our innate visual intelligence to advance understanding in science. The role of structural hierarchy in biology is explored, and we take this opportunity to introduce some of the "major players" at the levels of molecules, cells, and tissues. Maya is introduced, and some of its history traced. Finally, we celebrate the advances in 3D computer animation that have provided powerful, yet affordable tools for conducting visual explorations of complex systems.



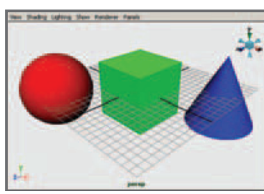
02 Computers and the organism

This chapter will survey the basic idea of computation and how it should be done automatically, by a machine. We will see to that a core tenet of information processing, *conditional control*, is used by both computer programs and living organisms to regulate activity. This sets the stage for understanding how computer programs can illuminate the structures and functions of biological systems.

03 Animating biology

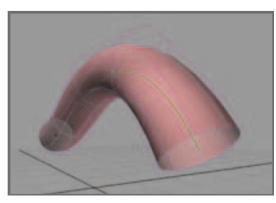
In this chapter, you'll explore the standard animation workflow, and see how it can be adapted to the needs of a biomedical researcher or animator. We examine the preproduction process, where a story is developed and refined, and a plan for the execution of the film is made. In the production phase, the hard work of building, texturing, animating, and rendering of the story elements takes place. In postproduction, the media developed in production are composited, edited, and packaged for delivery. These steps are applicable to most science communication contexts, and we propose a modified version of them to accommodate the unique requirements of biological systems visualization.

Part 2: A foundation in Maya



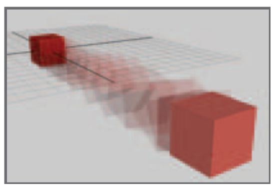
04 Maya basics

This chapter will get you immediately familiar with Maya, via a tour of the primary features of the user interface (UI). You'll learn about Maya's program architecture—the proprietary Dependency Graph and Scene Hierarchy—and get a sense of what's actually happening when you start pressing Maya's buttons. A basic understanding of "Maya behind the scenes" will greatly extend what you can accomplish with the software. We'll continue to develop this understanding in the subsequent chapters.



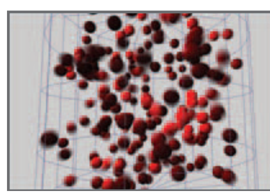
05 Modeling geometry

In this chapter you will learn to make geometric models. A discussion of different model types and their components gives an understanding of how complex surfaces are created from relatively simple beginnings. You'll also see how models are composed of nodes and attributes—the stuff of Maya's Dependency Graph—via practical examples.



06 Animation

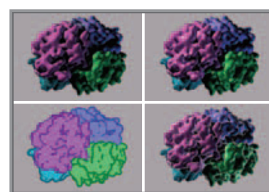
With animation, you'll bring your models to life. In Maya, to animate is to change some attribute over time—be it position, color, or speed, for example. You will see this definition applied as you learn to work with the tools of animation—keyframes and animation curves—to make objects move around and change shape. You'll wrap up the chapter with your first procedural—or algorithm-driven—animation, and a taste or what's possible when you set aside the standard UI animation tools and begin using written expressions to simulate motion.



07 Dynamics

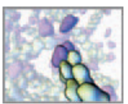
One of the truly powerful features of Maya is that it's a sophisticated, built-in dynamics engine that you can use to simulate real-world physics. It calculates forces and collision dynamics for soft- and rigid-bodied objects and for entities called particles. In this chapter you will create animations driven entirely by Maya Dynamics, in which objects are moved about by forces and collide with one another. These ready-made physics simulation capabilities are a boon not only to visual effects artists looking to emulate real-world phenomena, but also to the computational biologist looking to breadboard dynamic modeling scenarios before going through the effort and expense of building a custom physics engine.

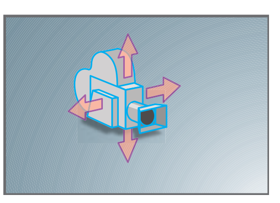
With Maya, you have at your fingertips the same tools for rendering proteins, cells, and tissues that professional CGI artists use to create the stunning imagery that has revolutionized Hollywood visual effects. In each of the following four chapters, you'll focus on an aspect of Maya's extensive rendering capabilities. Together these chapters will take you through the process of preparing an animated scene (showing the four subunits of the blood protein hemoglobin) for rendered output.



08 Shading

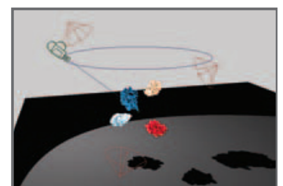
In this, the first chapter on the rendering process, you'll learn how to make and apply shading networks, or *shaders* for short. Shaders work with the lights in a scene to determine the appearance—color, texture, opacity, etc.—of objects in your finished renderings. You'll learn how to quickly create and apply shaders to multiple objects in preparation for rendering.





09 Cameras

Like a real movie camera, a Maya camera defines what your audience will see. Many features are available with a real camera are embodied in the Maya version, allowing you to set up and record shots in virtual 3D space much as you would in the real world. The Maya camera also defines your view of the 3D scene as you work with it, and is therefore an indispensable tool, whether or not you plan to make finished (rendered) movies with Maya. By the end of this chapter, you'll know how to set up and animate a camera along a track called a motion path—much the way a movie camera is set up on a track to move as it records the action.

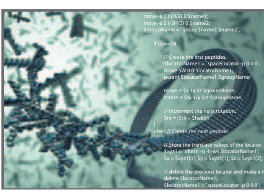


10 Lighting

If the camera is a cinematographer's *brush*, then light is the paint. Just like in the real world, light defines what is visible in your Maya scenes, and the quality of its appearance. We'll show you how to achieve professional illumination with minimal effort in order to get the most out of your images.

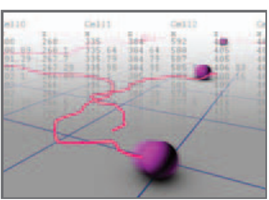
11 Action! Maya rendering

In this final chapter on the rendering process, you'll see how Maya integrates shaders, camera view, and lights to produce one or more image files. We'll explore the different render "engines" available in Maya and their relative advantages.



12 Mel scripting

At this point in the book, you'll know your way around the UI and be familiar with the concepts and terminology involved in modeling, animating, and rendering in Maya. You'll be ready to depart somewhat from the standard UI tools and start exploring Maya's scripting capabilities. This chapter introduces Maya's scripting (or programming) language, MEL (short for Maya Embedded Language). You'll learn how to run individual MEL commands and how to compose a script—or short computer program—out of multiple MEL statements in order to automate tasks in Maya. Readers new to computer programming will learn the basic concepts—syntax, variables, operators, flow control, etc.—in the context of MEL. Those with previous programming experience can scan the chapter to pick up the MEL basics. In either case, plentiful examples and a short tutorial will have you coding Maya tasks using MEL in no time.



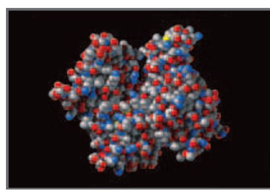
13 Data input/output

Ready-made software plug-ins are available for porting some of the more common 3D data formats to and from Maya. However, if you're working with a format for which no plug-in exists, such as experimental data formatted in a spread sheet, you may want to create your own importer

or exporter. This chapter shows you how to do just that using a suite of MEL commands for reading and writing external files. You'll also learn the MEL commands useful for formatting the text that you read and write. In the chapter's tutorial, you'll extract 3D coordinates from a cell migration data file, use them to visualize the moving cells, and then save out a report summarizing key migration statistics.

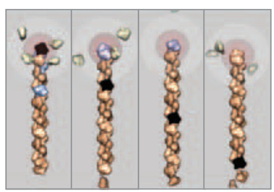
Part 3: Biology in silico—Maya in action

In this part of the book, you'll explore and use a workflow for in silico modeling and simulation that builds on your knowledge of Maya's UI and scripting capabilities. We present five tutorial-style projects, each dealing with a different level of biological organization—from a single protein up to a population of cells in a tissue matrix. In each project we'll guide you, step by step, through the composition of custom MEL scripts that automate the model building and/or dynamic simulation. Whether you're a scientist looking to explore Maya techniques in 3D computation or an artist visualizing topics in cell science, you'll learn a range of useful techniques that can subsequently be applied to your own projects.



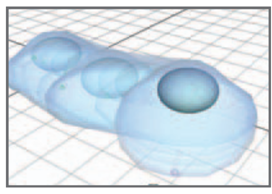
14 Building a protein

The ability to work with molecular models is essential to any 3D in silico approach to cell (and molecular) biology. To begin, one must first be able to build models using structural data. Once built, these models can be used to study and simulate a range of phenomena from protein folding to shape complementarity. In this chapter, you'll build a custom script to make a protein model using an external Protein Data Bank (PDB) file. You'll be able to use this script to make models from other PDB files and revise it to suit other data formats. Moreover, the chapter doesn't end when your model is built: we'll guide you through setting up and rendering a finished picture worthy of a book cover or wall poster.



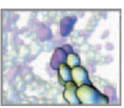
15 Self-assembly

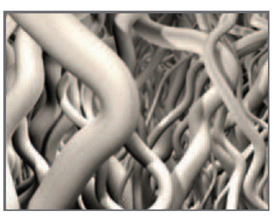
The self-assembly of macromolecular structures is key to the organization and function of cells and tissues. In this chapter you'll create a dynamic model of regulated self-assembly featuring an actin protein filament. You'll do this with custom MEL scripts that emulate molecular diffusion and chemical reaction dynamics.



16 Modeling a mobile cell

The study of mobile cells spans a huge range of biomedical research, from the spread of cancer to tissue regeneration. In this chapter you will create a simple cell model in Maya and make it crawl in response to a simulated chemical stimulus. By setting up parameters that control the cell's motion, including the degree to which it responds to the stimulus, you'll see how such a model could be extended to simulate and predict different modes of cell behavior.





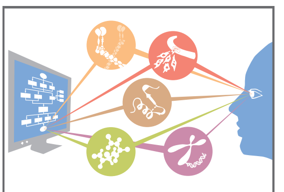
17 Modeling an ECM scaffold

In the body, cells live in complex 3D environments of the various tissue types. Research in regenerative medicine is increasingly focused on the relationships between cells and their surroundings, with a growing awareness that 3D tissue architecture plays a key role in cell behavior. In this project you'll use our in silico workflow to build a fibrous tissue matrix. A set of model parameters will let you vary the structure of each matrix you create. You'll see that, given a set of model criteria, you can leverage MEL to create structures of a complexity that would be impractical to attempt using the standard modeling tools available through Maya's UI.

18 Scaffold invasions

In this, the final project of the book, you'll model the penetration of your tissue matrix by a mobile group of cells—using only MEL and some custom methods we developed for mapping 2D cell motion onto 3D surfaces.

In no way does this chapter represent the limit of what's possible for modeling cell biology in Maya. On the contrary, we have only scratched the surface! We hope that this and the projects before it will inspire you to create new developments in this exciting field of 3D in silico biology.



19 Conclusion

In this chapter we revisit the themes and methods covered in the book and look ahead to the future of biocommunications and computational cell science.

Further reading

We tour the cell biology, 3D visual computing, and Maya tools and techniques in sufficient detail to advance you quickly and efficiently through each chapter in the book. Nonetheless, practical constraints have made it necessary to be brief in our treatment of many of the subjects. Where you desire more information, we encourage you to explore the *Further reading* we've listed according to topic.

Glossary

This book was written for artists and scientists alike. Depending on your field of work or study, you may encounter terminology and concepts that are new to you. In the *Glossary*, we've compiled many of the key terms used throughout the book. They are listed with references to the pages on which they're used.

CD-ROM and companion Website

Everything you need to work through the examples, tutorials, and projects—background information, step-by-step instructions, and MEL code listings—is provided on the printed pages. In addition, we've enclosed a CD-ROM with supplemen-

tary material. It includes MEL scripts, Maya files, and rendered animations from various chapters. The `read_me.txt` file in the root directory of the CD-ROM includes an index of the enclosed computer files.

On the books's companion Website you'll find updates and corrections (when necessary) to the files provided on the CD-ROM.

www.insilico.book.net.

Computer hardware and software

The Maya files and MEL scripts listed in this book and included on the CD-ROM were created and tested on a mid-range consumer-level PC with the following specifications:

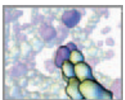
Software	Maya 8.5 for Windows
OS	Windows XP Professional 2002 (Service Pack 2)
PC	Dell Dimension 8300
CPU	Pentium 4, 3.20 GHz
RAM	1 GB
Graphics adapter	ATI Radeon 9800 XT, 256 MB DDR

The book's tutorials and projects have been developed over a number of versions of Maya, both in Windows and Mac OS. They have been **tested to work in Maya 8.5 for Windows**. Users of older versions of Maya may have to look around for commands whose names have changed, but the MEL code will probably work largely unaltered. As this book went to press, a new version was announced (Maya 2008). Although we have not had the opportunity to test our projects against Maya 2008, we have no reason to believe that the techniques we rely on would have altered enough to have broken them.

Similarly, the instructions for accessing Maya menus and tools, along with references to the Maya Help Library, are specific to Maya 8.5 for Windows. With a little adaptation they can readily be applied to learning Maya in other environments, namely Mac OS and Linux.

If you are considering purchasing Maya, we strongly recommend you ensure its compatibility with your hardware and software configuration by consulting the *system requirements* and *qualified hardware* specifications available via Autodesk's website:

www.autodesk.com/fo-products-maya



About the authors

Jason Sharpe is a cofounder of the award-winning AXS Biomedical Animation Studio in Toronto. Trained in mechanical engineering at Queen's University, fine arts at Emily Carr Institute of Art and Design and biomedical communications at the University of Toronto, he has worked on a wide range of Maya-based 3D animation projects for research, education, and entertainment.

Charles J Lumsden is Professor of Medicine at the University of Toronto. Trained as a theoretical physicist, he studies the mathematical logic behind illnesses such as Alzheimer's disease and cancer. He and his students have explored and championed a variety of 3D graphics software as aids to biomedical discovery, including top-tier commercial tools such as Maya and MEL.

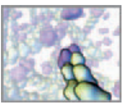
Nicholas Woolridge, Associate Professor of Biomedical Communications at the University of Toronto, has played a major role in the development of the visualization design field in the university's renowned Master's Degree in Biomedical Communications. His current research focuses on the optimization of visual media for medical research and education.

Acknowledgments

The splendid staff at Morgan Kaufmann, our publisher, has given us essential aid—mixed with clearheaded expertise and unquenched enthusiasm—as *In Silico* found its way through the press and into your hands. Tim Cox, then a senior editor at Morgan Kaufmann, saw sense in our idea that time was right for a richly cross-disciplinary book exploring Maya and its programming language, MEL, as tools for adventure and discovery in biology and medicine. Tim also got behind our conviction that such a book would be at its best if written for a use by a diverse audience of artists, scientists, and highly motivated private citizens. Morgan Kaufmann is a world leader in producing texts that map the subtle intricacies of MEL programming; we were, and remain, honored to have *In Silico* at home in this distinguished setting. Once Tim had the project launched, our Editor, Tiffany Gasbarrini, and Assistant Editors Michele Cronin and Matt Cater, helped us survive the twists and turns of bringing the book to life. Through our publisher we benefited from the comments of expert readers, who responded to drafts of *In Silico* either in whole or in part. Our thanks to these hard-working colleagues for their generous allotment of time and attention: Prof. Klaus Mueller of Stony Brook University; David F. Wiley, President and CEO of Stratovan Corporation; Azam Khan, research scientist at Autodesk Corporation; and five anonymous reviewers. Their input, uniformly deft and relevant, has helped *In Silico* complete its journey with enhanced strength.

In addition, two student reviewers—Lori Waters (of the Biomedical Communications graduate program) and Tatiana Lomasko (PhD candidate in the Institute of Medical Science), both at the University of Toronto—completed many of the tutorials, providing valuable feedback that helped us to hone our approach.

Throughout their history, Maya and MEL were invented and advanced by a community of brilliant computer graphics innovators principally located in Toronto, Canada (with colleagues in offices in Paris and California). The software was originally developed by Alias, Inc., and is now under the banner of the Autodesk Corporation. We cannot overstate our appreciation to Autodesk and to its staff of Maya and MEL experts in assisting us on occasional technical questions and allowing us to present the many illustrations in which Maya's user interface is depicted. As well, *In Silico* takes the view that influential inventions like Maya are what they are not only through the genius of their creators, but also because they appear at a specific time and place in human history. Therefore, appreciating historic trends in computer technology, computer programming, and 3D computer animation gives us better understanding of Maya and MEL. The history of Maya and MEL has not been written up extensively, and what sources exist we found to be occasional and widely scattered. We are therefore most grateful to Autodesk for granting us discussions with members of its staff, who number among the original inventors of Maya and MEL. These incredibly busy people answered our questions about origins and inspirations with patience, grace, and good humor. We are delighted to be able to incorporate the gist of those discussions here, by way of introducing you to the depths of Maya and MEL. In particular we must thank Joyce Janczyn, lead designer of MEL, as well Mike Taylor,



Duncan Brinsmead, and Jos Stam for talks that opened our eyes to the inner life Maya.

Ravi Jagannadhan gave considerably of his own time to review and test the many MEL scripts published here. And, during this entire time Azam Khan (research scientist at Autodesk) never tired of his informal role as our advisor and principal facilitator amidst the elite world of those charged with inventing the latest versions of Maya and Maya programming.

Since this book hopes to be useful to readers who are new to computers, computer programming, or 3D animation—as well as an efficient self-contained resource for experienced science researchers and computer artists—we have used key moments from computer history and animation history to lay newcomers a congenial path to MEL programming. It is a pleasure to thank all the computer historians, collectors, and archivists who helped us with information, recollections, and photographs. In particular we must note the extended assistance generously given our history frame by: portraitist Louis Fabian Bachrach III for his photograph of programming language pioneer John Backus, lead inventor of the Fortran language; computer scientist John Bennett (Sydney, Australia) for his assistance and support in presenting his early computer graphic of structure pattern data for the protein myoglobin; Deirdre Bryden, Queen's University (Kingston, Ontario) archivist, and Marnee Gamble, University of Toronto archivist, for mainframe history and photographs at these Canadian research centers; Martin Campbell-Kelly, University of Warwick (Coventry, UK), for early computer history and photographs, especially the EDSAC; Annette Faux, archivist at Cambridge University's Molecular Biology Laboratory, for early 3D models of the myoglobin protein; PDP-8 microcomputer collector and archivist David Gesswein, his wife Janet Walz, and their cats Khym and Py for the PDP-8 microcomputer photograph shot specially for the book; Calvin Gotlieb, University of Toronto, for access to his archives on that institution's computer center history; Bonnie Ludt, California Institute of Technology Archives, for her help with the Linus Pauling photographs; Dawn Stanford of the IBM Corporate Archives for assistance with IBM mainframe history; Peter Strickland, Managing Editor of the *Acta Crystallographica* journals, for his assistance with early computer visualizations of protein structure; Bjarne Stroustrup, inventor of the universally used C++ programming language, for his photograph; Marcia Tucker, Institute for Advanced Study Archives (Princeton, NJ) for assistance with the John von Neumann photograph; and Martin Zwick, Portland State University (Portland, OR), for information and photographs on key early work in molecular computer graphics. Our photo editor, Jane Affleck, also gave us strong assistance in sourcing hard-to-find images.

In Silico celebrates as well creative work by many of our colleagues who advance the visual interpretation of cell structure and dynamics through 3D computer graphics and animation. We especially thank Drew Barry, Marc Dryer, Stephen Ellis of Ellis Entertainment, David Goodsell, and Jenn Platt for letting us include their work here; Eddy Xuan and Sonya Amin of AXS Studio for their tremendous support and generous contributions to the book's illustrations; and Christina Jennings of Shaftesbury Films for letting us include animation stills from her pioneering dramatic series, *Regenesis*. Stunning visualizations in biology and medicine of course use technology other than computer graphics, such as photographic microscopy and video capture. We are indebted to: Peter Friedl and Katarina Wolf, University of Würzburg, Germany; Sylvia Papp and Michal Opas, University of Toronto; and Alexis Armour, Hôpital Hôtel-Dieu du CHUM, Université de Montréal, for their help and consent in

using their micrographs and/or video capture of cellular and tissue engineering materials. A special thanks goes to John Semple of Sunnybrook Health Sciences Centre for his expertise and guidance in regenerative medicine that helped shape the book's two final projects. John, who is both an artist and a scientist, also provided feedback at an early stage that helped us craft the book for researchers and artists alike.

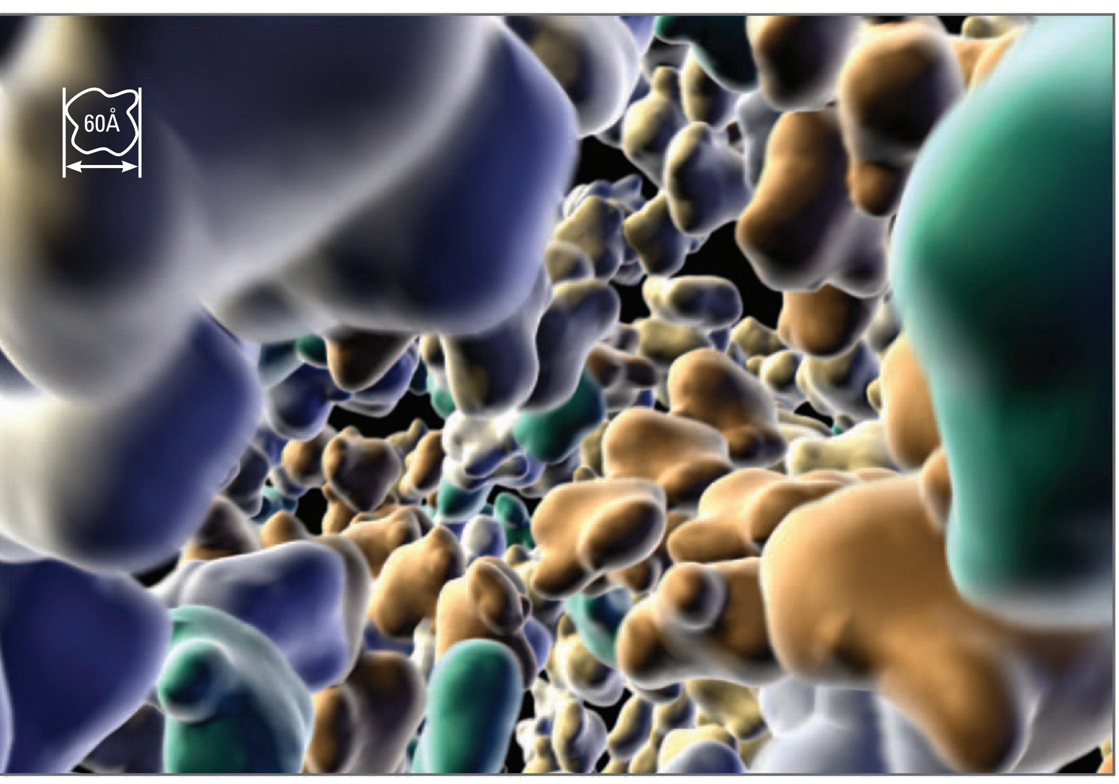
This book would not exist without the support we received from NSERC, the Natural Sciences and Engineering Research Council of Canada, in the form of a three-year grant under NSERC's Collaborative Research and Development (CRD) program. The CRD program brings University-based researchers in Canada together with companies that share common interests in science and technology—in this case the idea that a top-tier 3D animation package like Maya (itself a Canadian invention) can be a powerful tool in the hands of biomedical scientists and teachers. The NSERC-CRD initiative seeks outcomes with broad relevance to the advanced training needs and research application requirements of citizens in Canada and indeed worldwide. It has therefore been a special pleasure to design our work under this grant program, through NSERC-CRD Grant Number CRD 270158-03, entitled "Interpretive Visualization: Understanding cell systems dynamics through computer animation"; so that our findings can communicated in a book for working professionals and for trainees in both the sciences and the digital media arts. Our corporate partners, Bell Canada Enterprises in grant year 1 via the Bell University Grants Program at the University of Toronto, and Alias (now Autodesk) in grant years 2 and 3, were essential to the success of our CRD project and we are deeply grateful for their participation. At each step NSERC personnel at various levels—Eileen Jessop, Pamela Moss, Anne-Marie Monteith, Sylvie Boucher, and Lise Desforges—assisted us with practical guidance and patient advice.

No book large or small gets done without evenings, late nights, and weekends nipped from time otherwise owed to kith and kin. So we must end with our deepest thanks to our families, who have put up with all the stolen hours and steadfastly supported us throughout the book's creation.

This page intentionally left blank

Part 1
Setting the stage

This page intentionally left blank



Still image from a simulation model of molecular diffusion in Maya. The molecules are actin protein monomers and filaments which you'll meet in *Chapters 14* and *15*. In the 3D model on the left, different colors indicate different filament lengths. Width of one actin monomer ≈ 60 Å.

01 Introduction

The challenge

"I see."

With these words, human beings convey their understanding. This pervasive metaphor, of sight as the stand-in for comprehension, tells us something about the nature of thought.

Visual exploration is fundamental to human learning, problem-solving, and innovation. A surprisingly large portion of the cerebral cortex is devoted to decoding what we see with our eyes. So sophisticated is our visual perception that we are scarcely aware of its activity, and the cognitive sciences are only beginning to understand its complexity.

Since the Renaissance, the sciences have gradually awoken to the power of images to facilitate understanding. Andreas Vesalius' *De Humani Corporis Fabrica* (1543 C.E.) is considered the founding *text* of scientific anatomy, but its popularity and impact—it was serially and repeatedly plagiarized—was due to its exquisite dissection imagery. William Playfair, the inventor of many statistical graphics, opened the eyes of 18th century mathematicians and economists to the astonishing power of bar charts and scatter plots to condense pages of tabular data into readily apprehensible visual form.

Technologies of representation, reproduction, and mass communication were often first exploited for scientific communication. In the 18th and 19th centuries, the development of color reproduction technologies, so common in our mass media world, was driven by the demand of medical publication, where topics like dermatology required the accurate rendition of color.

Through the 20th century, many imaging technologies, such as electron and confocal microscopy, CT and MRI scanning, and ultrasound, were developed to satisfy science and medicine's demand for more and better evidence.

Now, in the 21st century, computer-generated imagery (**CGI**) is yet again expanding the scope of our visual exploration. The power to map complex esoteric data into images, expand and compress time and scale, and flexibly render concepts and processes in multiple forms have made the computer an essential component of many research endeavors. But there are gaping holes in the toolset available to researchers, and if commercial tools are not available, modern researchers usually have to contemplate building their own. Phenomena at the cellular and molecular levels are the principal focus of modern medical and bioscience research. At these scales, structures and events are often difficult or impossible to see in the lab, in real time, as they happen: the distances are too small, the times too short, the events too unusual. Instead, they are measured—mapped as numerical properties. But how can we envision what the numbers mean? New tools are needed that leverage the power of computer graphics (**CG**) to see into the complex web of structure and function in cells and tissues.

We, and other researchers in the emerging field of computational biology, are meeting this challenge with CGI: using the computer as a visual information machine to harness the brain's enormous prowess for insight into the knowledge encoded in cellular data and computer models. We call this approach **in silico**, since it is focused on computer methods for research discovery that will complement traditional **in vivo** and **in vitro** methods. Our approach is perhaps novel, in using tools built for another field entirely, to **breadboard** and simulate complex cell-scale phenomena. In this book you are going to learn about Maya, one of the most powerful computer programs for CGI, and how to use Maya (or tools like it) to represent, model, animate, and visualize in

In vivo is a Latin term common in medical research used to refer to things, processes, or experiments "in a living thing"; it is often used in opposition to in vitro, meaning "in glass", or, in other words, in an experimental apparatus.

silico diverse aspects of cell biology. This is an exciting new area bridging the sciences and the arts, and we have written the book with both scientists and artists vividly in mind.

In this chapter, we will begin by looking at why we visualize and how visualization in science can be characterized. Then we will approach topic of our visualization exercises: a hierarchical cast of biological characters, from atoms to tissues. Finally, we will look at the origins of our chosen tool, Maya.

Wetware for seeing

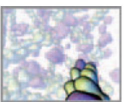
> "The drawing shows me at one glance what might be spread over ten pages in a book."
>
> —Ivan Turgenev, *Fathers and Sons*[6].

Why do we so easily refer to sight when we want to express understanding? Nature has equipped us with remarkable brains: a compact, energy-demanding organ that contains about 100 billion neurons, each neuron densly connected with up to 10,000 other neurons. The most distinctive part of the human brain is its cortex: a slab of gray matter equivalent to a sheet approximately $50\,\text{cm} \times 50\,\text{cm}$, and 2–4 cm deep, densely folded up and packed into our skulls. The cortex is central to our "higher" brain functions, like language and consciousness. A surprisingly large portion—40%[1]—of the cerebral cortex is devoted to vision. Why do we need such a large proportion of our brains devoted to decoding what we see with our eyes?

Human visual perception is a pattern recognizer of extraordinary speed, power, and discrimination. And yet, on a day to day basis, we remain scarcely aware of the astonishing suite of tasks performed by visual perception. It's as if we walk through the world with an incomprehensibly powerful supercomputer behind our eyes, and as we employ that supercomputer to navigate stairs or read the cereal box in the morning, we remain completely oblivious to it.

Vision is a source of deep, and paradoxically *invisible*, intelligence; harnessing that intelligence is one of the goals of scientific visualization. A comprehensive understanding of vision would perhaps help us map scientific goals to standardized design criteria; alas, such an understanding is as yet a work in progress. The mechanisms underlying much of the process of visual perception are largely mysterious. Decades of effort by cognitive psychologists and neuroscientists have begun to unravel the mystery, but they are far from the complete story, and a comprehensive explanation of vision may be tied to even more hard-won understanding (such as the elusive nature of consciousness itself). The fact that computer scientists and artificial intelligence (**AI**) researchers have yet to mimic vision's power in even the most rudimentary way (they have yet to create a robot that can "see" anywhere near as well as a human toddler) is a testament to the difficulty of the challenge. In the interim, we can draw inspiration from the understanding of visual perception that currently exists and from existing heuristics in the realms where meaning is concentrated in the form of images: film, illustration, and art.

Let's give vision's computational might its due and create some images that exploit its power, solving scientific problems in the process. At the very least, given the above, the failure to take advantage of our most complex and subtle faculty, the failure to visualize—when it would enhance our productivity or understanding—would be a terrible waste of the processing power inside our heads.



The quote from Turgenev is perhaps the source of the familar proverb: "a picture is worth a thousand words." It rings true, even though, as a cynic would point out, it is hard to imagine a picture that could express the sentiment embodied in the phrase.

In the hurly-burly of science, the potential power of visual expression occasionally takes a back seat to the practical importance of verbal expression, especially for "serious" communication. Papers must be written, presentations prepared, and posters assembled, all depending principally or exclusively on words and numbers. This is understandable; as far as we can tell, language is essential to human intelligence and is sometimes considered the "stuff of thought". But if we could open up our heads and "listen in" on our thoughts, they would be far different, and more confusing, than the transcription of an "internal conversation." The "stuff of thought" contains images (or their mental counterparts!), as well as numerous other sense impressions, such as sound, tactility, body position, and physiological state.

We are a multimedia species. We hear repeatedly that we live in an increasingly visual, media-saturated world. New literacies are being formed around the sophisticated media objects we consume, and science is becoming open to the idea that exploiting such literacies will facilitate scientific communication and discovery.

That we may take vision for granted—not just in our everyday lives, but in the process of scientific understanding—should not obscure its power. Indeed, it is hard to think of a revolution in scientific understanding over the past 3,000 years—astronomy, medicine, physics, chemistry, engineering, geography, and so on—that has not relied, in whole or in part, on a breakthrough technology for seeing the world in new ways. CG and animation are working this transformative effect on modern biology and medical science—especially in the realm of the otherwise small, invisible cells where the first steps to aging and disease are born.

Visualization in science

The verb *visualize* has two meanings: the conjuring up of an image in the mind's eye; and—more importantly for science—to make visible to the eye. Visualization in science has a long history (as noted above) and has taken on numerous forms, serving numerous purposes. A comprehensive survey of visualization in science would include everything from classroom blackboard sketches to supercomputer renderings, and all the problem-solving visual representations in between. Some representations are direct mappings of perceptible phenomena (a simulation of a storm cloud) and some are visual analogies or metaphors which aid interpretation of the phenomena (a diagram of the "plumbing" of a cell-signaling pathway). Some images simplify the phenomena and some seek to represent it in its full, empirical complexity.

It can be helpful to think of visualization as existing within a potential "design-space", with "level of interpretation" along one axis, and "level of complexity" along another (Figure 01.01). We don't intend this scheme to be definitive or exhaustive, but it can help to frame the available possibilities.

Visualization goals may be positioned at various points within this design-space. At the upper left of the space (high in interpretation and low in complexity), images and

High
Metaphors & conceptual approaches (cartoons, etc)
INTERPRETIVE VISUALIZATION
Abstractions (Process-flow diagrams, etc.)
Level of Interpretation
simple
Illustrative approaches
detailed
Empirical (Micrographs, CT Scans, etc.)
Low
Low Level of complexity High

High
Level of Interpretation
Low
Low Level of complexity High

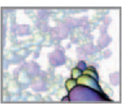
FIGURE 01.01

Above: One potential visualization "design-space".

Below: Three representations of the bacterial flagellum and their respective positions in the proposed design-space.

events are simplified and analogized, perhaps in order to teach, to clarify, to show trends, or to convey an overall impression. At the lower right (high in complexity and low in interpretation), the image is derived from an imaging method or linked directly to data (indeed, it is usually a direct rendering of that data). While teaching and showing trends are possible with such images, they are designed to be as accurate as possible, such that it may stand as a source of measurement, evaluation, or diagnosis.

Valuable work can be done anywhere within the design-space, but it is important to know the purpose of the visualization and its intended audience. The same phenomena can be visualized in a number of ways depending on the audience; imagine explaining bacterial self-propulsion to young



schoolchildren, or to undergraduate biology students, or to doctoral students in a subspecialty of cell biology.

We term what we do **interpretive visualization** (regardless of whether the final media are bound for a lay audience or experts) since we are modeling systems invisible to the naked eye and since we are using visual computing to represent and explore specific ideas about how cells work and how diseases begin with changes in cell function. Where necessary, we simplify the representations in order to communicate most effectively and, where appropriate, we take advantage of various representational strategies to make the images more intelligible. Interpretive visualization inhabits most of the upper left of our design-space, leaving a small area for empirically derived images. We don't want to convey the impression, however, that "interpretive" means that these visualizations are less rigorous. Across the frontier of in silico biology, you can be involved with developing visualization models that fuel learning and innovation.

The organizational context of visualization will prove important in choices about the methods and approaches we will explore later in this book. We will now turn our attention to the organizational context of biology, which informs the very phenomena we wish to represent.

Organizational hierarchy: Keys to biology *in vivo* and *in silico*

Any sufficiently detailed examination of the structure of living things reveals an astonishing hierarchy of organization (see Figure 01.02). From the simplest amino acids upwards, nature builds on the underlying structure in fascinating ways. Understanding this organization is crucial to comprehending how living matter operates and how those operations can be represented and visualized using computers. Indeed, in *Chapter 02* we will explain some of the analogies between the hierarchical nature of computation and the functional activity of living physiology. We will sketch here some of the more important components of this hierarchy; readers with a background in biology may safely skip this section. Readers new to some of these ideas may wish to complement this survey with further reading in an introductory college-level biology text, or in a popular explication such as David Goodsell's *The Machinery of Life*. Complete references are listed in the *Further reading* section of the book.

Atoms and molecules

Atoms are the base units for our consideration of living structure. These particles are the smallest unit which retain an element's chemical properties. The chemical properties of atoms essentially define whether they attract or repel other atoms and under what conditions. Molecules form when two or more atoms form an arrangment due to mutual chemical bonds.

Amino acids

Amino acids are the small molecules that are strung together, using instructions from our DNA, into proteins. As the building blocks of all proteins, they can be considered the base of organismal hierarchy (Figure 01.02).

Level	Examples	Scale
Tissue	Collagen matrix	100 μm
Cell	Mitotic cell	8 μm
Multimer	Actin filament; Microtubule	24 nm
Protein	Insulin; Actin	4 nm
Molecule	Water; Phenylalanine (amino acid)	2 nm
Atom	Oxygen; Carbon	2 nm

FIGURE 01.02

A "cast of characters" representative of selected levels of organismal hierarchy.

Proteins

Proteins are an amazingly diverse group of biomolecules, all composed of long chains of amino acids folded into complex, and functional, three-dimensional (**3D**) shapes. At this moment there are about 10,000 types of protein actively at work in your body: digesting your food (pepsin); carrying oxygen in your blood (hemoglobin); clearing a path through collagen for a migrating white blood cell (matrix metalloproteinase); self-assembling into cell-skeletons (actin, tubulin, and vimentin); transcribing DNA into RNA (RNA polymerase); and translating RNA into fresh proteins (the multi-protein ribosome) to name just a few.

Molecular arrays: Protein societies

Proteins sometimes work on their own as single large molecules, as in the case of soluble enzymes. More often they are part of larger, multi-part structures like cell membranes. Many structural proteins, such as those involved in the cytoskeleton, self-assemble into long polymers, which further join together into networks providing deformability and structural resilience to cells. You'll be seeing much more of one such protein, actin, as you work through this book.

Other proteins link together in pairs (forming dimers), trios (trimers), or in multi-protein complexes (multimers). Some of these structures can be very elaborate, perform complex tasks, and exhibit strikingly machinelike behaviors. Some of these multimeric structures in turn make up larger structures, such as the organelles we see inside cells.

The bacterial flagellum (Figure 01.03), for instance, is composed of the molecular equivalents of an engine, drive-shaft, bearings, and a propeller all made of protein molecules that, with the aid of other cellular components, assemble into a complex array. The result of this molecular assembly is a highly functional molecular machine that can operate at an astonishing 200–1,000 rpm, propelling the bacteria through its aqueous environment.

DNA

Although it doesn't fit neatly into our hierarchy, we should mention an essential (perhaps *the* essential) molecule of life, which has its own unique structural hierarchy and a unique information storage capacity. DNA (deoxyribonucleic acid) is a long, polymeric nucleic acid which takes the form of two complementary strands arranged in a double-helix. The two strands are linked by the bases adenine (A), thymine (T), cytosine (C), and guanine (G). DNA is the foundation of the molecular basis of heredity: the sequence of base pairs (A,T,C,G) forms a long serial code organized into genes (discrete, protein encoding sequences), regulatory sequences, and regions of unknown function. Genes code for the sequence of amino acids in a protein molecule; the Human Genome Project has revealed that our cells contain about 25,000 genes. Genes are translated into messenger RNA, which is transcribed into proteins by a complex molecular machine called a ribosome.

The whole cell

At the level of the whole cell, various multi-molecular complexes—such as the those composing the cell's structural framework, the **cytoskeleton**—are large enough to be visible to light microscopy. The cytoskeleton is composed of actin filaments, intermediate

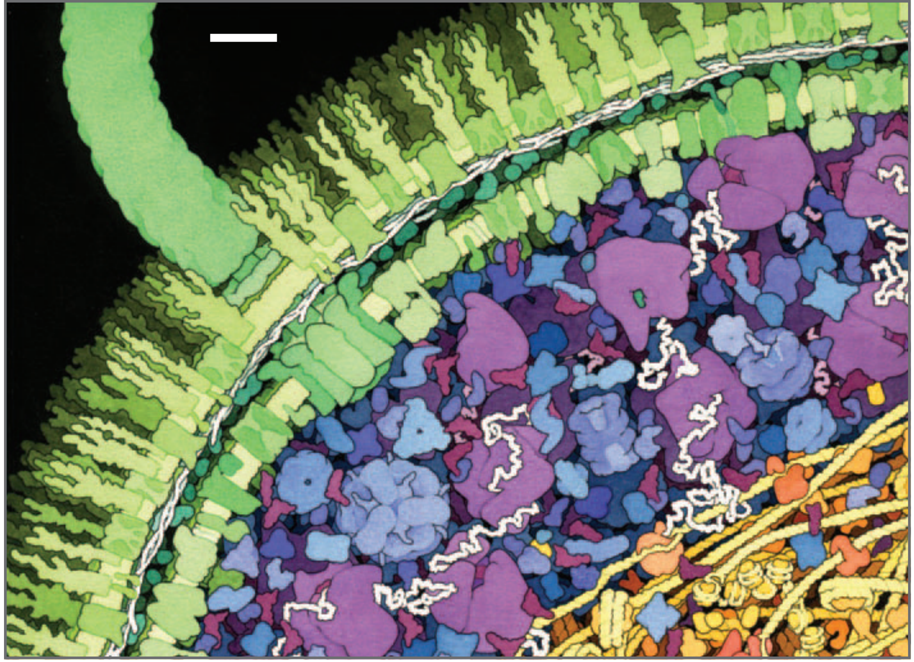


FIGURE 01.03

Illustration of the molecular machinery comprising a bacterial flagellum—an elegant example of multimeric protein organization. Scale bar ≈10 nm.

Courtesy and © David Goodsell. Used with permission.

filaments, and microtubules, and plays an essential role in cell structure, motility, division, and the transport of substances within the cell.

The cytoskeleton is a dynamic, hierarchical mesh (Figure 01.04). It grows and shrinks, degrades and reassembles. Actin filaments exemplify this activity and are vital to cell deformation and movement, as well as muscular contraction. Actin filaments are 0.7 nm thin polymers, made up of twinned helical, rope-like chains of F-actin.

A startling example of the dynamic nature of cell organization is mitosis, or cell division. In this act of cellular reproduction, many events have to carefully coordinate, beginning with the duplication of the cell's genetic material (so that there is enough for each of the daughter cells). The envelope surrounding the DNA (the nuclear membrane) dissolves and the duplicated DNA condenses into paired chromatids. Microtubules extend from anchor points within the cell to attach to the central regions of the chromatids and then pull the newly minted chromosomes apart. Once the genetic material has been cleanly split, two nuclear envelopes can reform. A cleavage furrow, powered by actin and myosin (the same proteins that allow muscles to contract), forms around the center of the elongated cell and pinches the cell into two.

Tissues and organs

Some cells are lone actors, like the patrolling lymphocytes of the immune system, which migrate throughout our bodies looking for foreign invaders. Most cells, however, aggregate by the thousands or millions into tissues, whose composition and hierarchy serve some functional goal.

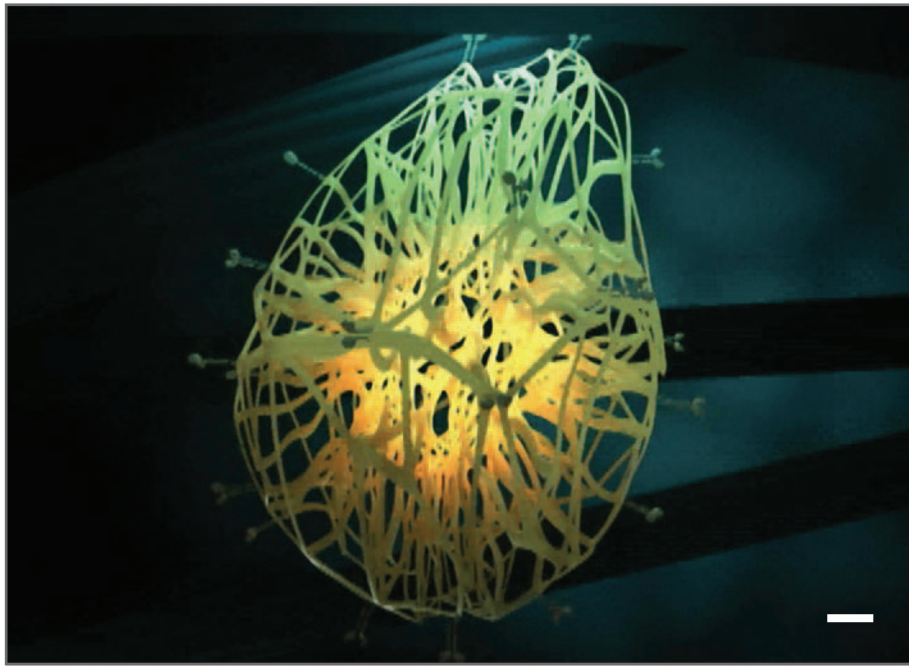


FIGURE 01.04

Still frame from an animation (created in Maya) demonstrating concepts of cytoskeleton dynamics as put forth by Harvard cell biologist Don Ingber. Scale bar ≈1 µm.

Courtesy and © 2004, Eddy Xuan.

For example, connective tissue is essential to the structural integrity of most multicellular organisms; it is a major component of cartilage and bones and underlies the structural resilience of many tissues and organs, such as skin. Connective tissue, which will form the basis for one of your Maya projects in this book (Figure 01.05), is primarily composed of long polymers of various types of the protein collagen. Collagen is deposited as structural meshes by various cell types in the body.

Engineered connective tissue scaffolds are an area of modern research. Currently, creating scaffolds optimized for particular research or therapeutic purposes is a time-consuming endeavor. In some of our research work we use computational models to experiment, in silico, with variously structured virtual scaffolds. This approach may one day speed the development of engineered tissues vital to future therapies in wound-healing, spinal injuries, and more.

Micro to macro

As we have seen, living matter is organized via a deep structural hierarchy: amino acids build proteins; proteins (often) build polymers; proteins and polymers build cells; cells and cell products (e.g. extracellular matrix) build tissues and organs; and tissues and organs build organisms. We've simplified here, especially with respect to the many other cell components (fats, carbohydrates, micronutrients) that are necessary inputs to, and products of, living physiology. But this hierarchy will be our guide; in *Part 3* of the book you will build computer models that represent biology at several of these key levels of structure: a single protein; a protein polymer; a single cell; a tissue; and a cell population.

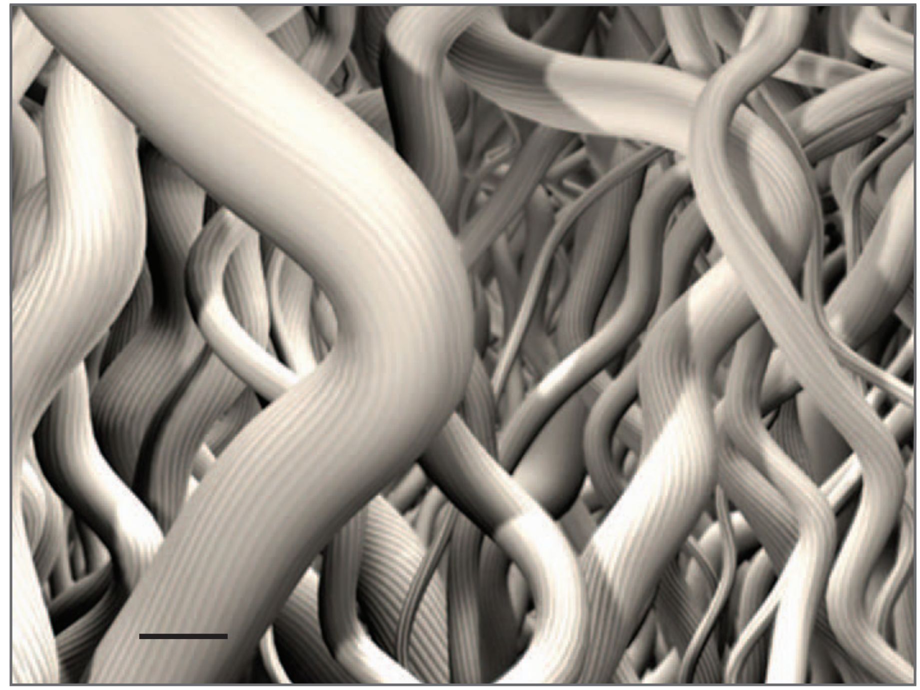


FIGURE 01.05

The connective tissue scaffold you will build using Maya in *Chapter 17*. Scale bar ≈10 μm.

Enter Maya

Even the most amazing idea for new ways of seeing the world is powerless without the tools and technical means of bringing the new vision into practical use. This book is not about all of computational biology or even about all the important ways cell biology is being visualized on computers. The science and art are already too vast (and our expertise too limited!) to cover all of that. Instead, this book is going to introduce you to a specific means of visual simulation—using modern high-end 3D animation software to accelerate development—via a specific tool: the Maya animation program. Having used many of the traditional alternatives (Figure 01.06) in our research and teaching during the past 30 years, we are convinced that this more recent approach and tool is an important addition to the visual computing arsenal. Maya and the methods it supports are not panaceas and will not displace key special tools already in place (or yet to be invented). As a mediator of exploration and rapid hypothesis prototyping, however, Maya (and software like it) is powerful and accessible to fast deployment by users from either scientific or artistic backgrounds.

Maya is a general purpose modeling, animation, and rendering application with a sophisticated dynamics engine for simulating physical forces and collisions. Users can import or create geometry of varying types (**polygonal** and **spline**-based surfaces), arrange these objects in a virtual 3D world, and change their positions and deformations over time. Numerous tools are provided within a well-designed user interface (**UI**)

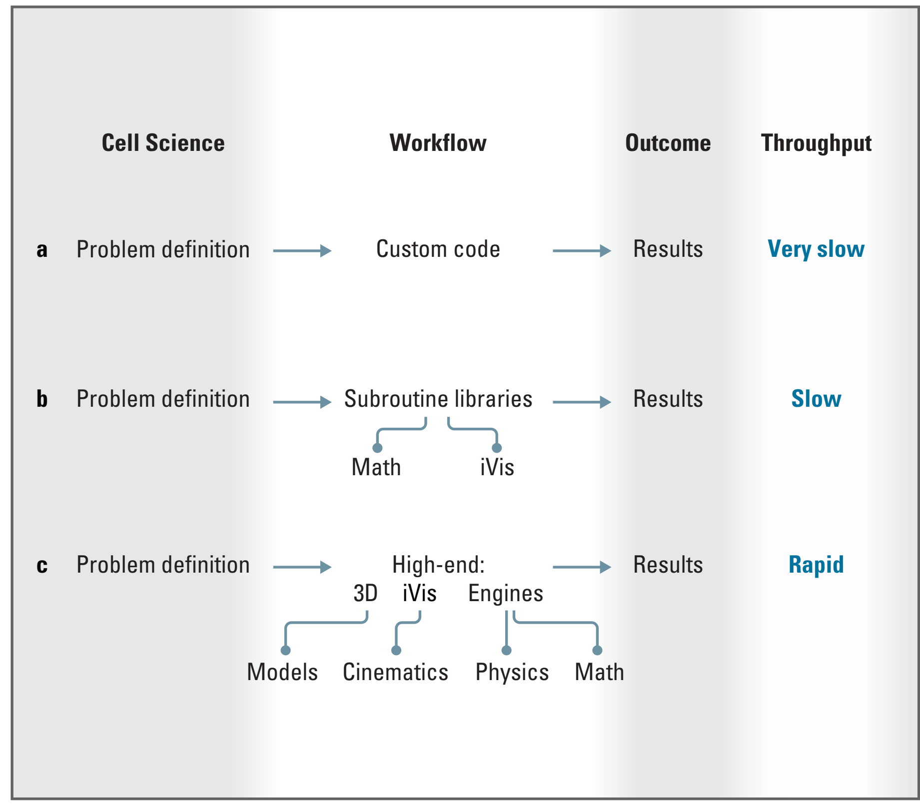


FIGURE 01.06

In silico workflows: theoretical advantages in throughput of using advanced tools like Maya.

to allow for the creation of sophisticated animations, from the articulation of synthetic characters for film and television to the explosion of a dying cell.

Maya: A brief history

In 2003, Alias|Wavefront became Alias Systems Corporation. Then in 2006, the company was bought by Autodesk, makers of AutoCAD drafting software and 3Ds Max, another high-end 3D modeling and animation package. Autodesk continues to develop and market the Alias Systems product line: Studio Tools; SketchPad; MotionBuilder; and Maya.

Maya was released commercially in 1998 by the CG software firm Alias|Wavefront (**A|W** for short), which was headquartered in Toronto, Canada. As the top-tier animation and visual effects (**VFX**) package it was by no means the work of a single software company in a single development effort. Maya incorporated the product lines of Toronto's Alias Research (Alias for short), Wavefront Technologies (Wavefront for short) of Santa Barbara, and Thompson Digital Images (**TDI**) of Paris, into a single package. Interestingly, each of the three contributing companies began developing commercial products in 1984, independent of one another. Between them they came to dominate the global markets in computer animation, special effects, and industrial design (**ID**) software—markets that the three were in fact largely responsible for creating.

Wavefront was founded by Mark Sylvester, Larry Barels, and Bill Kovacs. During their first year, under the production leadership of John Grower, Wavefront created some of the earliest CGI for television—notably, opening sequences for National

Geographic Explorer, BRAVO, and Showtime. Their first commercial software offering, called Preview, was adopted by NBC Television, game developer Electronic Arts, and NASA, among others. Subsequently, Wavefront branched into the ID and scientific visualization fields with desktop software they co-developed with computer maker Silicon Graphics Inc. (**SGI**). Wavefront's core 3D modeling, animation, and rendering package for movies and television was called The Advanced Visualizer (**TAV**). In 1992 they released Dynamation, a sophisticated particle dynamics tool developed by Jim Hourihan. Hourihan wrote a scripting language called **Sophia** that allowed users to automate tasks in Dynamation with user-friendly computer code. Also in 1992, Wavefront released Kinemation with SmartSkin, a sophisticated inverse kinematics (**IK**) package used for character rigging and animation. In 1993, Wavefront bought TDI for their modeling and rendering technologies. The combined team of Wavefront and TDI began working on a next-generation CGI package called Cyclone which would combine elements of TDI's Explore and Wavefront's TAV, Dynamation, and Kinemation. The project was short-lived, however: another corporate merger would bring more graphics technology into the mix in another next-generation package—Maya.

For his role in creating Wavefront's Dynamation software Jim Hourihan received a Scientific and Engineering Award from the Academy of Motion Picture Arts and Sciences in 1996.

Alias Research was formed in 1984 by film and animation enthusiast Stephen Bingham, programmer Susan McKenna, CG specialist Nigel McGrath, and artist/programmer David Springer. Their first commercial offering was a spline-based modeling and animation package called Alias/1 (we will explore the two cardinal surface modeling techniques—spline-based and polygonal—in *Chapter 05: Modeling geometry*). The Alias product line was used by TV post-production facilities for animation and motion graphics and in manufacturing for ID. One of the first and longest-standing Alias customers was General Motors, which used Alias software to realize efficiencies in automotive design. Like rival company Wavefront, Alias worked closely with SGI on the implementation and distribution of their ID product line (this relationship would later be cemented in the merger of Alias and Wavefront under SGI). The advances in modeling and rendering embodied in the Alias ID software (now Autodesk StudioTools) have helped major manufacturers worldwide—BMW, Honda, Volvo, Ford, Apple, GE, Sony, Kraft, Motorola, and many others—design, previsualize, and showcase their products, while reducing the time from concept to production.

The ingenuity that made Alias software so valuable to the manufacturing sector was also turning up opportunities in Hollywood. With Alias/2 (and later PowerAnimator), VFX artists were able to tackle problems that were previously unimaginable. In 1989, former Alias employee Steve Williams used Alias/2 at Industrial Light and Magic (**ILM**) to create the "living water" creature effects in James Cameron's *The Abyss*. The movie won an Oscar for Best Visual Effects in 1990. Another Best Visual Effects Oscar followed in 1992 for James Cameron's *Terminator 2: Judgment Day*, recognizing the chromium villain VFX which Steve Williams created using PowerAnimator. Yet another Oscar acknowledging VFX created with PowerAnimator came in 1993 for the dinosaurs in Steven Spielberg's *Jurassic Park*. PowerAnimator was both the state of the art and the industry standard for CGI in Hollywood, used by most major animation and VFX studios, including ILM, Pixar, Walt Disney, Sony Pictures Image Works, Dream Quest Images, and Warner Brothers. Later, the developers of PowerAnimator—John Gibson, Rob Krieger, Milan Novacek, Glen Ozymok, and Dave Springer—were recognized for technical achievement in the 1998 Academy Awards, as were Wavefront's Bill Kovacs, Roy Hall, Jim Keating, Michael Warhman, and Richard Hollander for the development of Advance Visualizer.

The same year Jurassic Park dinosaurs caused a stir, Alias began work on their next-generation animation software—code named Maya (a Sanskrit word referring to "the illusion or appearance of the phenomenal world"[2]). Originally conceived as an add-on animation module for PowerAnimator, Maya would soon become a platform for the leading technologies in surface modeling, animation, dynamics, and rendering. In 1995, SGI bought strategic partners Alias and Wavefront and merged them into a single company: Alias|Wavefront (the purchase included Wavefront-owned TDI and its Explore software). Rather than duplicate work in separate product lines, it was decided to combine the resources of the three development teams. They were tasked with creating a single product, one that would incorporate the best of what PowerAnimator, TAV, Dynamation, Kinemation, and Explore had to offer, and continue to serve each of the original three customer groups. The new software took its core architecture—the Dependency Graph and Scene Hierarchy (which you'll meet in *Chapter 04*)—and its name from the Alias Maya project, along with a fledgling scripting architecture based on Tcl and used to build the UI and to run commands. Alias also contributed its extensive spline modeling code base to the software. Wavefront brought its dynamics and IK engines as well as the Sophia scripting language. Sophia replaced Tcl and was developed into MEL, the Maya Embedded Language (you'll use MEL extensively throughout the projects in Part 3 of the book). TDI contributed its polygon modeling engine.

Maya 1 was released in 1998, three years after its development *officially* began. It integrated the top CG advances to date and over a decade of R&D and industry experience spanning three companies and their clients. In its early days, due to its price tag, Maya was not widely accessible to CG artists. At a cost of about $50 K an install of several Maya seats represented a significant investment for all but the largest studios. Its price eventually came down to the level of competitors SoftImage, 3D Studio Max, and Cinema 4D (among others), so that smaller studios and individuals could afford access to Maya's capabilities.

Like its predecessors—PowerAnimator, TAV, Kinemation, and Dynamation—Maya dominated 3D computer animation and CGI in Hollywood and earned A|W an Oscar for Scientific and Technical Achievement in 2003. Since Maya evolved in the design and entertainment industries, it's not surprising that a high priority has been placed on the visual quality of its output. Maya provides users with powerful control over simulated lights, cameras, and textures in order to control the appearance of the finished renderings. These are important qualities not just for the development of movies and games, but also for relating scientific concepts and visualizing data, where they are critical to making complex processes readily understood.

Beyond the obvious reasons related to visual sophistication, Maya's continued popularity among high-end visual effects companies grows from its flexibility and openness. Along with its scripting language, MEL, Maya has a well-documented API (Application Programming Interface) based in C++ for writing custom plug-ins—software modules design to automate special, often repetitive, tasks. Maya's unique node-based Dependency Graph architecture, along with its API and scripting capabilities allows for its integration into custom workflows—where an animation studio's "home-built" tools are mated with commercial software to create solutions specific to the needs of a particular project.

Maya and interpretive visualization

In the book's projects, you'll use Maya to create visual interpretations of cell biology data and phenomena. As molecular biologists and biochemist know, there exist many special-purpose tools for the representation and manipulation of computer-simulated biological structures, especially at the molecular level. These include excellent free software (**freeware**) applications. Popular molecule viewing applications like UCSF Chimera[3] and Jmol[4] make it easy to view biomolecules in a variety of representations, including ball-and-stick, ribbon, and solvent-accessible surface. Simulation software such as the NIH Visual Molecular Dynamics (**VMD**)[5] application enables users to model molecular interactions in small numbers using structure and reaction data. These powerful tools were designed for specific tasks and users, and therefore don't fulfill a complete range of visualization needs. For instance, camera, lighting, shading, and animation options are limited in even the most advanced molecule viewing applications, such as UCSF Chimera. However, through an integrated workflow with more sophisticated visual software packages like Maya, users can leverage the combined power of bioinformatics-driven modeling and advanced data visualization.

For more information on biomolecular visualization history and resources, please refer to the *Further reading* section that begins on page 585.

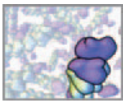
Moreover, the ease with which complex, dynamic systems of interacting objects can be built, animated and visualized in Maya make it an effective tool for the rapid prototyping of molecular dynamics simulations like the one shown in Figure 01.07 (we'll touch on the history and conventions of molecular representation in *Chapter 14*).

Biological visualization and simulation software must satisfy several requirements. First is the ability to import structure and interaction data directly from publicly accessible databases. Second is a physics engine that helps rapidly prototype critical events, such as diffusion and chemical reaction inside the cell. And third is a robust suite of visualization tools to allow the user complete control over how images are recorded and presented. This last point is especially important, whether the end goal is to demonstrate a biotechnology product or to identify a previously unknown event in a sequence of biochemical reactions. As you will see, Maya offers diverse strengths in the context of these criteria.

MEL: Maya's in silico language for interpretive visualization

The MEL language uses familiar programming conventions to allow control over virtually every aspect of Maya's operation, from its UI to model-making, from simple translational motion to complex dynamic behaviors (Figure 01.08). At the heart of MEL, as with most programming languages, is math. It follows that just about any structure or process that can be described mathematically—from simple molecular diffusion to a complex AI scenario—can be expressed using MEL. This means that, in addition to employing Maya's built-in physics capabilities, you can program the rules of cell activity relevant to a specific project, in essence writing your own in silico biology engine. Parameters and equations then can be adjusted to test hypotheses or to make predictions in silico in advance of expensive, time-consuming real-world experimental work.

There are two important features that separate Maya from other mathematics-based research tools such as MATLAB and Mathematica. First is its facility for physical modeling, whereby surfaces and structures of non-trivial complexity can be created



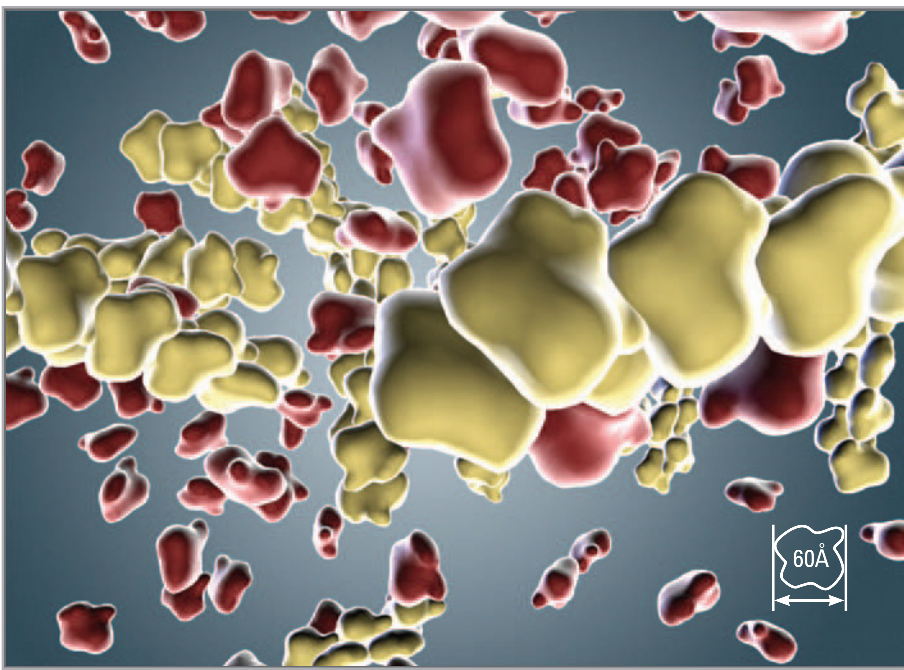


FIGURE 01.07

Maya provides a robust, programmable environment in which to prototype molecular structures and events. Shown here is a still from one of our Maya-based simulations of the dynamics of actin protein assembly. Width of one actin monomer ≈60 Å.

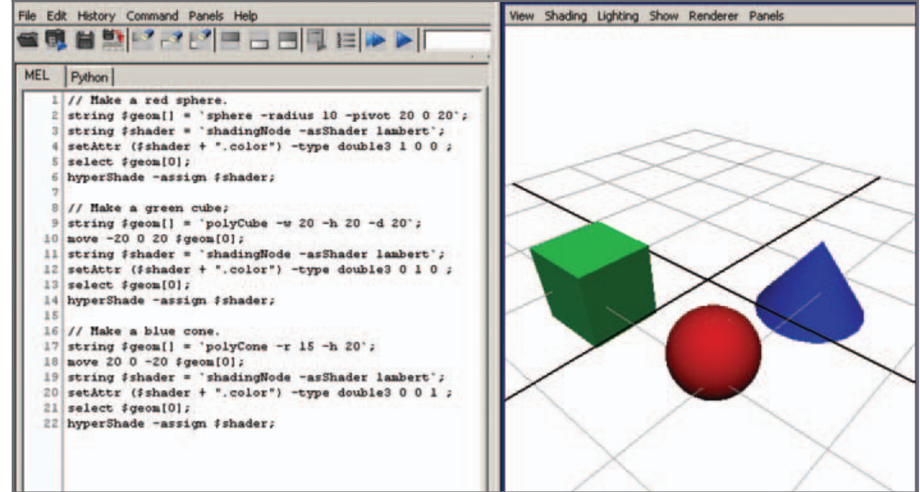


FIGURE 01.08

A simple program, or script, written in MEL, populates a scene with the objects on the right.

in 4D space (3D-space plus time), with a sophisticated UI for interactive exploration, a task that is not yet possible with strictly numerical, non-visual software. Second is Maya's visual flexibility in producing striking 4D visual representations of the underlying functions at work in a simulation. This is significant both for investigators and

those with whom they must communicate, since it allows researchers to tailor the representation to best communicate their work. The MEL-based simulation environment can satisfy both this visual, or qualitative, requirement and the need for hard, quantitative data.

Endless possibilities

The scientific study of the living cell brings countless opportunities to apply computing and visualization to crucial problems, to map the cell's molecular world, and develop new treatments for disease. There is now so much data on how cells work, and so many possible paths to understanding, that only computers and mathematics, aided by the human brain's incredible ability to see, can tame the complexity. Today, the cutting edge is the adventurous, effective use of the latest high-end tools for digital graphics and animation programming, as top-tier aids to achieving these goals. In just a few short years, visual computing has become an indispensable engine of discovery in cell biology. In order to get the most out of our time with Maya and MEL, let us therefore take a closer look at the nature of computing and of animation to see why tools like Maya are making such an impact.

References

1. Ware C: *Information Visualization: Perception for Design*, 2nd ed. Morgan Kaufmann, San Francisco, 2004.
2. Soanes C, Stevenson A: *Oxford Dictionary of English*. Oxford University Press, London, 2003.
3. Pettersen EF, Goddard TD, Huang CC, Couch GS, Greenblatt DM, Meng EC, Ferrin TE: UCSF Chimera—a visualization system for exploratory research and analysis. *Journal of Computational Chemistry* 25: 1605–1612, 2004.
4. Jmol: An Open-Source Java Viewer for Chemical Structures in 3D. (website): http://www.jmol.org, accessed August 2, 2007.
5. Humphrey W, Dalke A, Schulten K: VMD—visual molecular dynamics. *Journal of Molecular Graphics* 14: 33–38, 1996.
6. In this quote from *Fathers and Sons*, the character Bazarov touches on the power of visualization to capture rich fields of information. This translation from Ivan Turgenev's 1861 Russian novel is by Richard Hare, in Chapter 16 of the 1948 edition of Fathers and Sons from Hutchinson & Co., London.

This page intentionally left blank

02 Computers and the organism

Introduction

Terms like "bioinformatics" and "computational biology," which we met in *Chapter 01*, sound impressive. But what *is* a computer and what does one mean by computation? For that matter, why should either matter to the scientific study of cells, organisms, and the diseases afflicting them? Since this book is a meeting place for readers from many backgrounds, where science and the visual arts come together on common frontiers, we would be remiss in overlooking such questions before plunging into the details of Maya, MEL, and biological simulation.

In this chapter we are going to give you some additional background and context for the many things you'll learn about Maya and MEL as you work through the book. We hope this will not only increase your enjoyment of the technical material, but will strengthen your ability to apply what you will learn in exciting new ways. Inventions like Maya and MEL are high impact breakthroughs not just because brilliant people created them. They also have the good fortune of appearing at the right time and right place in human history. Therefore, appreciating historic trends in computer technology, computer programming, and 3D computer animation can enhance our grasp of Maya and MEL. You won't uncover any secrets of Maya programming in what follows. If you are eager to press ahead into MEL coding, you can skip to the next chapter and return here when you want to explore some of the context of in silico biology via MEL and Maya.

This context is sending ripples through the life sciences, and the waves of change are spreading far past debates about numerical methods best suited to crunch biological data. As we write these words, research biologists are mapping the molecular logic that drives living matter's structure and function. How is this previously hidden logic to be decoded, understood, and applied? It turns out that ideas invented by computer engineers to help build the 20th century's information processing machines—ideas about codes, algorithms, procedures, and so on—are revolutionizing the way biologists think about the origins and operation of living systems. And, remarkably, the discoveries biologists are making about how cells operate and communicate are, in turn, giving computer scientists ideas for radical new computers with unprecedented capabilities, based on principles of information processing in living matter. Later on in the chapter we'll take a brief look at these exciting trends, in which computer engineering is shaping biology and vice versa, and where they might be heading.

Our approach will be an historical quick start, moving rapidly from crucial ideas to the discoveries of a few major pioneers. We must, yielding to considerations of space in the book, omit mention of many influential people and milestones on the path to modern computation. The absence of a specific person, invention, or acronym from our short account in no way implies a belief they lack historical relevance: quite the opposite. What follows is a tightly focused sampler, meant to ground the specialized discussion of our later chapters. Hopefully it will also whet your interest to delve more deeply into the people and inventions that took us from the abaci and measuring rods of yesteryear to the supercomputers of today. The readings at the end of this chapter and in *Further reading* section will take you further in your historical explorations.

Information and process

By a computer we mean a machine that transforms patterns of information into other patterns of information by following a strictly defined procedure or algorithm.

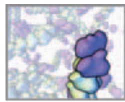
Computable procedures are those we can write down in sufficient detail for their correct, automatic execution and completion by a non-conscious device. Computers are remarkable artifacts because, when supplied with power, they are capable of going through a sequence of changes in their physical state that is equivalent to going through the steps of an algorithm.

In the mechanical computers of the early 1900s the state was a particular configuration of the machine's rods, cams, and gears. As the gears turned, the computer state changed and input was transformed into output. Calculating a different algorithm meant changing the mechanical connections holding the machine together—literally rebuilding the device. With the arrival of digital electronic circuits, the physical state could be much more ductile patterns of electric charge in vacuum tubes or of magnetization in solid state circuit chips. The early mechanical computers had to be built with specific types of problems, like ballistic trajectory prediction, in mind. In the electronic computing machines, changing algorithms was as easy as changing to another magnetic pattern, and thus by comparison hugely versatile.

Since computers are machines, they must obey the laws of physics. The digital electronic computer must be built so the flow of electricity through its circuits (its hardware) can change a stored digital pattern in a manner equivalent to the operations of logic, arithmetic, and storage/retrieval on the information represented by the pattern (the software). For example, if addition takes place, a part of the machine must, at just the right moment, move into a physical state capable of triggering the appearance of the digital pattern for the sum in the computer's circuits. We say the instruction for addition has been carried out. It does so by setting in motion the electrical events for the sum operation in the hardware. For traditional binary digital computers, the instruction would be written as a string of 1s and 0s, each numeral specifying the active (1) or inactive (0) state of a digital element in the instruction-sensing circuit (called a register in certain hardware designs, see below). The set of instructions recognized by the digital hardware is the machine language of the computer.

Language and program

Once coded into the instruction language of its machine hardware, the procedure followed by the computer is called the (computer) program. People who write programs are called computer programmers. Programmers of the earliest electronic computers had no option to coding in machine language—a time consuming, error-prone, tedious practice that immediately motivated the development of more human-friendly instruction sets or programming languages. To see the burdens of a workflow based on coding in machine language, consider this instance of an (entirely hypothetical) computer in which three binary digits are used to distinguish among the instructions, and digital memory locations are allocated as bytes, or sequences of eight binary digits. Our hypothetical machine therefore has a memory capable of holding $2^8 = 256$ numbers, a capacity not out of keeping with the earliest digital machines. Again hypothetically, we'll suppose that setting the binary string 101 in the instruction register triggers the addition operation, placing the results in a temporary storage location (an arithmetic register) of the circuitry. Once loaded in the instruction register, the string 011 triggers another sequence of changes in the electrical state of the machine, which end with the contents of the arithmetic register copied to a specified location in the computer's memory and the arithmetic register cleared to zero for the next time it is needed.



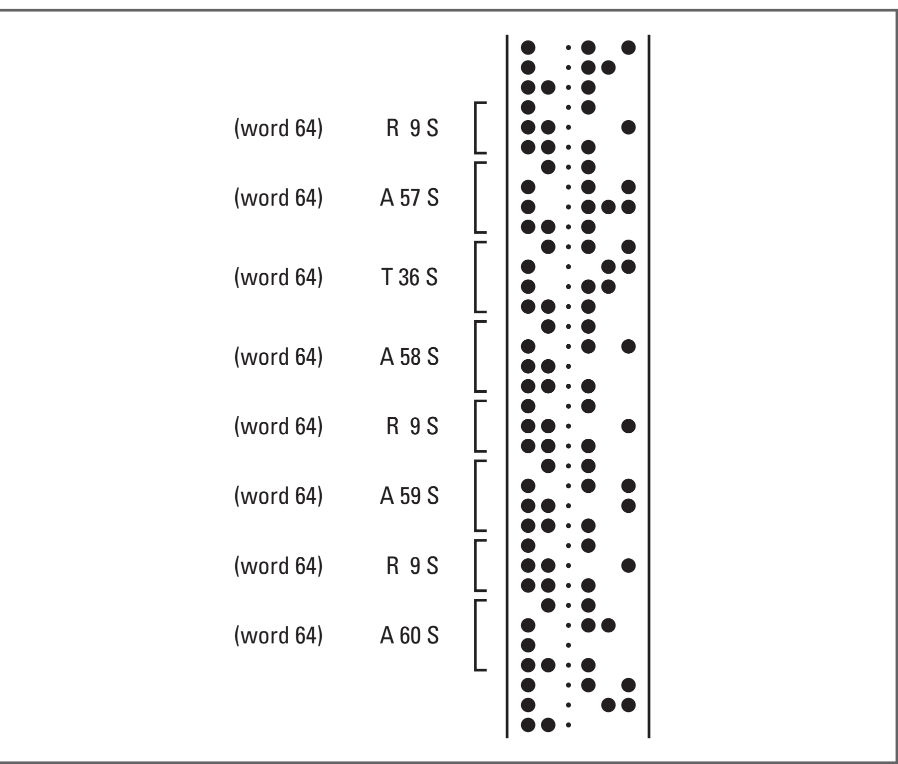


FIGURE 02.01

Instruction code for the EDSAC computer at University of Cambridge, from Maurice Wilkes' 1949 program[1] to solve Airy's differential equation $d^2y/dx^2 = xy$. The equation describes rainbows and other electromagnetic wave phenomena in the atmosphere. The Wilkes program is one of the earliest science calculations (perhaps the very first) to run on the EDSAC or any other stored program digital electronic computer. The program instructions are the letter–number–letter combinations in the center of the drawing. The corresponding section of the paper tape is at the right. Programs were input to the EDSAC through its paper tape reader. Maurice Wilkes lead EDSAC's development at Cambridge and would soon co-author the world's first computer programming textbook.[2]

Illustration by and courtesy of Martin Campbell-Kelly. Copyright ©1992 IEEE. Used with permission.[1]

At the start of our addition sequence we (somehow) know that the numbers to add are stored in the memory locations 10011001 and 01100011. Note that these are the locations of the numbers, not the values of the numbers stored there. We have also (somehow) determined that their sum is best stored at memory location 01111100. Our machine language code for the addition is then expressed by the binary string:

1011001100101100011
(i.e. instruction 101 acting on the values at locations 10011001 and 01100011)

followed by the string

0110111110000000000
(i.e. instruction 011 acting to move the arithmetic register contents to location 01111100)

One of the first instruction sets to use mnemonics is Betty Holberton's C-10 language (1949) for the BINAC (and later UNIVAC) computer.

Imagine having to code an entire video game or a complex ecological simulation this way! With the benefit of a half century's hindsight in software history, we can see ways to ease the programmer's task. For example, we might write a machine language program that can accept alphabetic acronyms or mnemonics for the machine code instructions as input, along with some white space between the address bytes, then output the packed binary strings for processing by the hardware. Aided by such a language processor, our code snippet already looks better:

ADD 10011001 01100011
(which our language processor outputs as 1011001100101100011)

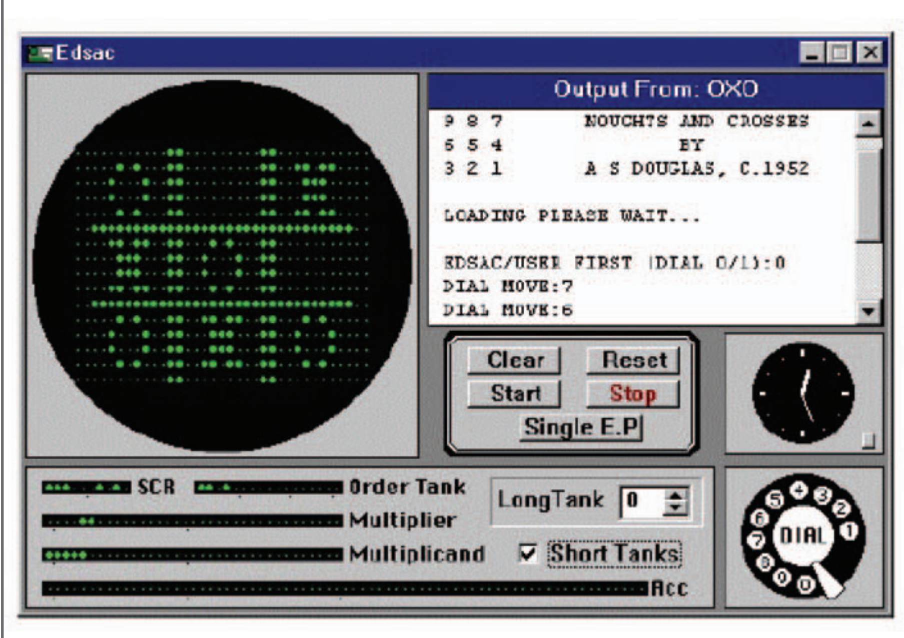


FIGURE 02.02

User interface of Professor Campbell-Kelly's EDSAC simulator, a PC program that lets you experience what it was like to code and run programs on a first-generation von Neumann machine[3]. See http://edsac.net for documentation and downloads. The circular display at the top left visualizes the contents of the EDSAC memory. Registers involved in processing instructions and arithmetic operations are below. The tic-tac-toe pattern visible on the memory tube is no accident. While a Ph.D. student at University of Cambridge, Alexander S "Sandy" Douglas wrote one of the earliest computer games—an EDSAC program that played tic-tac-toe and used the memory tube to visualize the game state as a 35 × 16 element bitmap.

Courtesy Martin Campbell-Kelly.

MV 01111100
(output as 0110111110000000000 by the language processor)

Suppose we next enhance our language processor with the capacity use a symbol table, in which we tell it that certain alphanumeric character strings stand for certain memory location bytes (call the command for that DFL, or define memory location); we can then write:

DFL X 10011001

DFL Y 01100011

DFL Z 01111100

ADD X Y

MV Z

which looks even better. If the language processor is made smart enough to build the symbol table on its own, then we are relieved of the duty to figure out all the memory allocations manually, ahead of time. Now the machine can do that automatically, once the program in input. All we need write is:

ADD X Y

MV Z

and let the language processor take over. Language processors with this type of capacity for acronym substitution are known as assemblers, and the languages often referred to as assembly languages.

The Initial Orders program, developed c. 1948–1951 by David Wheeler, Maurice Wilkes, and Stanley Gill to process user input to the EDSAC computer (Figure 02.01), was an early assembler. Running the Initial Orders also bootstrapped the machine and loaded the assembled user code into memory.

FIGURE 02.03

With the 1960s came minicomputers, smaller and more affordable by far than the mainframes. Sold by the tens of thousands, they helped popularize assembly language programming and ignited progress in computer networks and graphics. Shown here is one of the most famous, the diminutive PDP-8 introduced in 1965 by the Digital Equipment Corportation ("DEC") of Maynard, Massachusetts. With a core memory of 4K–32K 12-bit words, an entry price under $20K (i.e. under $100K in 2006 dollars), assembler and compilers, the mass-produced PDP-8 was a runaway success. In this photo Khym, a domestic short-hair feline who resides with PDP-8 archivist-collector David Gesswein (http://www.pdp8online.com), helps us see just how compact a PDP-8 is (compare the mainframe setups in Figures 02.05 and 02.08). The DEC VR14 graphics monitor beneath Khym is driven by the minicomputer—here a PDP-8/I—via the multi-purpose AX08 Laboratory Peripheral interface unit, just above left. The VR14 rendered graphics on its display screen as patterns of discrete spots of variable brightness. On the 8/I's front panel run the rows of activity lights and rocker switches (a hallmark of PDP-8 design) used to set register and memory contents and control machine activity. Arriving three years after the original PDP-8, the 8/I was the first in the series to use integrated circuits in place of discrete transistors.

Photograph by, courtesy of, and copyright © 2007, David Gesswein, The Online PDP-8 Home Page, http://www.pdp8online.com

High and low

Although assemblers mark a big step toward a more human-friendly programming language (Figure 02.03), some important limitations are evident. For example, every time we need to carry out a multi-step mathematical or logical operation we must code in the lines for the sequence. It would be helpful to have libraries of often-used instruction sequences, say for sorting lists or evaluating common mathematical functions such as the logarithm or the cosine, and to be able to refer to these sequences by an acronym code (a macro). The assembler would then recognize the code, extract the machine instruction sequence (the subroutine) from the program library, and drop it into place in the binary instruction sequence. Modern assemblers make extensive use of macros and subroutine libraries.

You can see, however, that despite such facilities there lingers a rather direct relationship between the arrangement of the coding statements in our assembly program and the sequence in which the machine hardware carries out its digital operations. Along with their syntactic elegance and their semantic concision, assemblers acknowledge the strictures of machine code. Users of computers, however, generally care about the logical structure of the problems they want to solve, and not as much about the inner workings of the computer's circuits. Users want high-level programming languages, whose elements and operations target their problem-solving needs. Other, automatic programs could then translate code written in a high-level language into the low-level language: the assembly code, or the machine code instructions of the computer's hardware.

For example, many of the earliest users of computers were scientists and engineers whose problems demanded a lot of algebra and formula manipulation. If instead of:

ADD X Y

MV Z

we can instead input the code line:

Z = X + Y

for the formula itself, we are programming the computer in a language that more directly expresses our mathematical problem. Moreover, if X, Y, and Z are not simple integers but more involved mathematical entities like complex numbers or arrays, the required sequence of machine language instructions—to work out the sum of two complex numbers or two arrays—could be quite lengthy indeed. These long sequences in machine language are "summed up" by the elegant statement Z = X + Y in our high-level programming language. MEL, the programming language of Maya, is a very high-level language. MEL of course enables you to instruct Maya in algebraic operations. But with MEL you can also invoke and operate on a diversity of elements specific to the needs of 3D computer graphics and animation. With a few keystrokes you can create virtual movie cameras and set their simulated optical properties, place and activate simulated lights and set their spectral characteristics, generate 3D shapes and move them in complicated ways, and so on. For example, the little command:

```
sphere –radius 5 –name "Cell";
```

causes a 3D sphere labeled Cell, of radius 5, to appear in your Maya world space at the origin. Behind the scenes, the MEL language processor embedded in Maya translates the sphere instruction and carries it out in your computer's machine language.

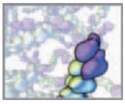
Interpret or compile?

The designer of high-level programming languages must deal with two linked issues. The first is the structure of the language itself: the problem-solving elements and operations it will allow (such as addition of arrays or the creation of 3D spheres), the rules by which such operations are expressed and combined, and the way in which control of the computer circuits is passed among the various parts of the program code. Much of our book deals with the principles of MEL language structure and their use in designing effective MEL programs.

The second issue confronting language designers is the manner in which programs written according to the language's rules will be processed and executed by the hardware. Today's high-level language processors usually work in one of two ways (or some modern hybrid of them). Both date from the earliest days of digital electronic computing:

1. The high-level code acts as instructions to a "virtual computer" running as a real-time simulation on the computer's hardware. The high-level instructions are thereby executed right away. The virtual computer, coded in machine language or assembly (or, today, even in another high-level language), drives the hardware to produce results consistent with the meaning of your program code. Such language processors are known as **interpreters**. Maya's language processor for MEL runs as an interpreter, giving you real-time access to the execution of MEL commands and MEL code files. The MEL interpreter is very efficient and you will be able to watch it operate and work with it in rapid-fire cycles of coding and code testing.

Early interpreters of interest to scientists and applied mathematicians included R. A. Brooker's FLOATCODE (1952) for the Ferranti Mark I computer at Manchester University in England, the Laning-Zierler algebraic formula interpreter (1953) for MIT's Whirlwind computer, and John Backus' Speedcoding (1953) for the IBM 701 mainframe.



2. The high-level code is submitted as data to a language processor that translates the high-level instructions into machine language. The machine language version of the program is then placed on the hardware to execute. Language processors that operate this way are called **compilers**. Maya itself runs in your computer as a program compiled into machine code.

To get the most out of MEL and Maya it is important to grasp the basic distinction between interpreters and compilers. Although both types of language processor accept high-level program code as input, their output is different. The output of an interpreter is the results of the executed program, e.g. data. The output of a compiler is another computer program, ready for execution (and data output) on the same or different hardware.

Early language processors with recognizable attributes of modern compiler pipelines were A-0 from Grace Hopper's team for the UNIVAC 1 (1952–1953), Alick Glennie's AUTOCODE (1952) for the Ferranti Mark I at Manchester, and the Transcode (1954) of J. Patterson Hume and Beatrice Worsley for the Mark I (the "FERUT") at the University of Toronto, Canada.

Modern compilers operate in several steps, in which the high-level code is translated into efficiently organized assembly language, required auxiliary programs are brought in from subroutine libraries, symbol tables are constructed, and the final executable machine language file is prepared to receive control of the computer hardware. Actually, current vernacular tends to limit use of the term compiler to the actions of the first of a series of programs that lead from your file of high-level code to its binary machine code version running on the hardware. Compilers massage the program into assembly language (or an equivalent) and hand off the tasks of library fetching, address resolution, and so on to partner programs with names like linkers and linkage editors. The final result is an executable program ready to go, and for simplicity we'll use compile to refer to the whole pipeline.

As of this writing, we are aware of no compilers for the MEL language per se. Projects needing to blend the advantages of Maya's real-time user interface with the potential efficiencies of compiled execution can transition from an all-MEL workflow to Maya's C++ application programming interface (API). This API lets you steer Maya with compiled C++ code. We look forward to exploring this API's uses in biomedical science in a future book.

The Backus watershed

BASIC was developed as a novice-friendly computer language under the direction of John Kemeny and Thomas Kurtz at Dartmouth College in 1964.

A common misunderstanding is that the coding rules of a high-level language (its grammar and syntax) determine whether the language will be a compiled language or an interpreted language. This is not the case. BASIC, for example, was first implemented as a compiled language but, with the rise of Microsoft, gained fame as a user-friendly interpreted language. True, some languages, like Fortran, are designed with highly efficient compilation in mind, and others (like JavaScript) with interpretation, but in today's world of fast, big capacity hardware the selection of a compiled or an interpreted language implementation tends to be driven by the vendor's understanding of user needs. Interpreters, for example, afford right-away execution and are thus well tailored to quick prototyping in real-time interaction between the user, the hardware, and the program results. On the downside, an interpreter's "virtual computer" emulation runs as code on top of the real machine's hardware, and so typically execute programs less quickly than the hardware will execute a well compiled version of the same program. Compilation, however, separates the user from the program results by a sequence of (potentially time-consuming) steps in which the high-level code is transformed into machine code, run, errors spotted, fixes made, the code re-compiled, and so on around the shakedown loop of program design.

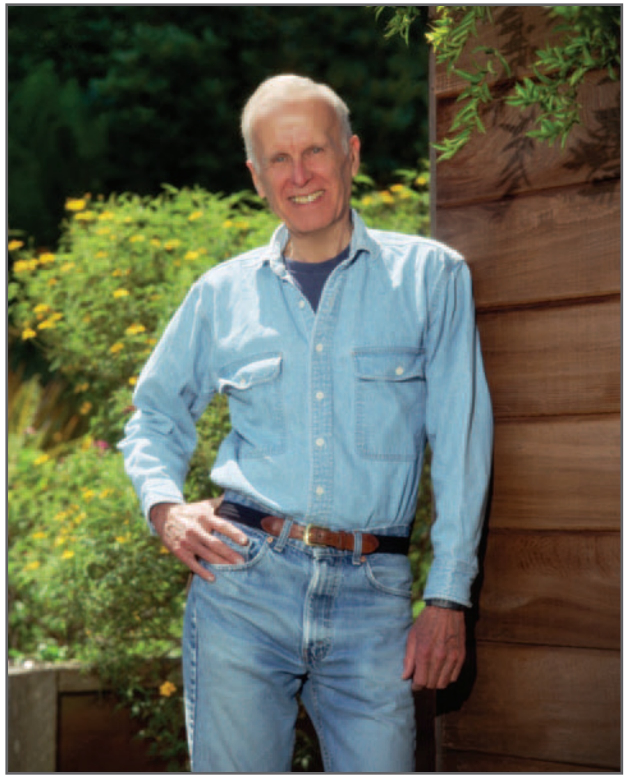


FIGURE 02.04

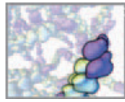
John Backus, lead inventor of the Fortran programming language, pauses for a photograph in 1997, 40 years after Fortran's official release in 1957. Before the ascension of the C++ language Fortran's easy learning curve, scientist-friendly syntax, and fast compiled code kept it unrivaled as the prime choice for high performance number crunching on digital computers. Photograph by Louis Fabian Bachrach III.

Louis Fabian Bachrach ©.

The "virtual computer" running inside an interpreter also can incorporate handy instructions and processing abilities absent in the machine hardware. This can ease the programmer's task. Early digital electronic computers did integer arithmetic fast and well, as they did juggling fractions between +1 and −1. Hardware circuitry for general floating-point calculations and array index management, however, was not marketed until the mid-1950s. Before that, number crunchers were obliged to figure out how their problems could be scaled into the integer and fraction-arithmetic range of the machine. Alternatively, they could run their problems on machines in which floating-point math and array handling were carried in software. By the early 1950s, time in floating-point subroutines accounted for a substantial fraction of processor time, to which interpreters added little additional overhead. These interpreters could emulate computers with floating point and array index management right in their instruction language. With the arrival of floating point and array index hardware this edge, enjoyed for several years by programming language interpreters, was lost.

We have heard it said, by die-hard scientific number crunchers, that all of programming language history since Backus is a series of footnotes to Fortran and its compiler. We would not go quite this far ourselves.

Nonetheless, professional programmers of the time remained skeptical about **compilers**. It seemed unlikely that a computer program, running as a compiler, could take high-level program code and transform it into machine code as efficient and reliable as the low-level code written by human specialists—let alone do this job quickly and automatically, across the general range of computer programming applications. The doubts, while understandable, soon disappeared. The breakthrough was instigated by John Backus (Figure 02.04) and his team at IBM. It took the form of an amazing compiler for their



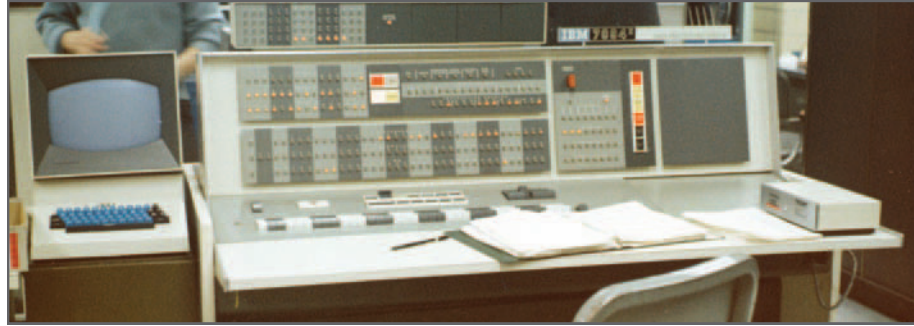


FIGURE 02.05

The IBM 7094, one of the early transistorized mainframes, was a workhorse of the batch processing environments available to scientists and engineers in the 1960s. Introduced by IBM in 1962, the 7094 carried 32K of 36-bit word core memory and double-precision (72-bit) floating-point arithmetic hardware. About three million 1960s-era US dollars (roughly $50 million today) would buy you a 7094 setup (not counting the system operators and the "glass room" environment needed to house it). Or if you could spare $60K or so a month, there was a rental plan. IBM documented a basic cycle time of two microseconds for the 7094s operation. Here: the 7094-II operating at the University of Toronto (mid-1960s).

Calvin Gotlieb Personal Records B2002–0003/001P(21), courtesy of the University of Toronto Archives and Records Management Services.

Fortran language specification (1954–1957). Through a series of ingenious optimizing steps, the Backus compiler for Fortran systematically output executable low-level code that closely matched, or even exceeded, what expert programmers could write by hand.

A consequence was the temporary eclipse of interpreters as an efficient medium for processing higher-level programming languages. With **compilers**, there was no "hit" from the overhead of the interpreter running the program. Jobs could be compiled and fed through the bulky mainframes of the time in large batches, which kept the pricey hardware well occupied (Figure 02.05). Users cooled their heels till the batch holding their compiled job was finally done. The speed and capacity of today's desktop computers, however, have again made interpreters a strategic option, especially in situations where, as noted above, rapid prototyping is carried out in real-time settings. Now most users in the sciences and the arts work with high-level languages for which efficient interpreters and/or **compilers** are available; problem coding in low-level languages like assembly and machine instruction sets has become a specialized skill for crucial niche applications like hardware device drivers. We have carefully designed the coding projects you will undertake in this book, to give you MEL programs that generate interesting results in reasonable time on mid-range desktop computers. We hope you will enjoy the brisk pace of MEL and Maya mastery this approach supports.

John von Neumann, b. Hungary 1903—d. USA 1957. Mathematician/chemical engineer celebrated for his fundamental work in diverse areas of mathematics and science, including logic and the foundations of mathematics, quantum physics, game theory, mathematical economics, computer design, and automata theory.

Stored programs

We have said that a computer is an algorithm-guided machine for transforming information. As most of us encounter it, however, the computer is not just any

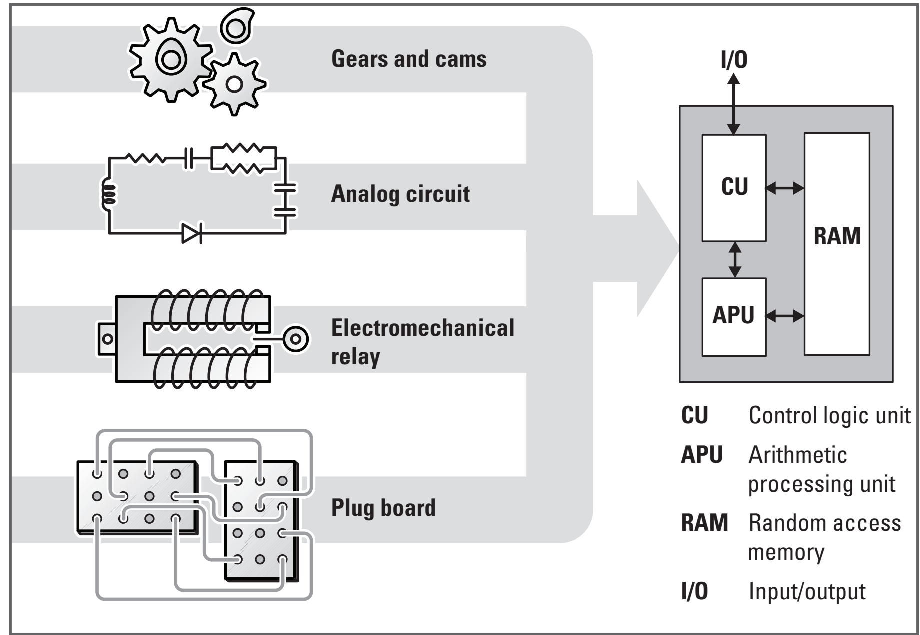


FIGURE 02.06

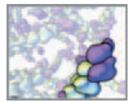
The von Neumann machine and some immediate ancestors.

old information-wrangling digital electronic machine. It is a device of a very specific kind, often called a von Neumann machine (Figure 02.06) in honor of John von Neumann (Figure 02.07), the mathematical genius whose writings launched computer engineering as a formal discipline grounded in the logic and mathematics of information processing circuitry.

von Neumann certainly did not stand alone in the 1940s rush of breakthroughs, which began the era of modern computing. A revolutionary new architecture for computer hardware design was "in the air." Far-ranging discussions involved von Neumann with other leaders of computer design and construction, such as J. Presper Eckert, John W. Mauchly, Herman Goldstine, and Arthur Burks. However, von Neumann's noted passion for writing things up combined with his prodigious insights in a set of documents, authored between 1945 and 1948, which amount to the first detailed plans for digital computer operation in recognizably modern form.

Untill the mid-1940s, machines capable of transforming information, and so calculating answers to scientific and engineering problems of the era, took many shapes as engineers debated the technology best suited to replace the "state of the art" (i.e., legions of error-prone humans cranking out numbers with the aid of pencil, paper, slide rules, logarithm tables, and adding machines). Ingenious arithmetical contraptions of diverse forms—those whirling collections of gears and cams we met earlier, nets of analog circuit boards, clacking banks of electric relays, simmering racks of vacuum tubes—took a turn in the limelight. These were the worthy ancestors of today's digital computers.

When von Neumann first met them in Summer 1944, Eckert and Mauchly were leading the ENIAC computer project at the University of Pennsylvania's Moore School of Electrical Engineering. A giant of its time, ENIAC sported as many as 17,000 vacuum tubes and 1,500 electromechanical relays in its circuitry. Eckert and Mauchly would go on to develop the UNIVAC line of commercial mainframe computers. von Neumann was retained for a time by IBM to advise on its early ventures in electronic computing.



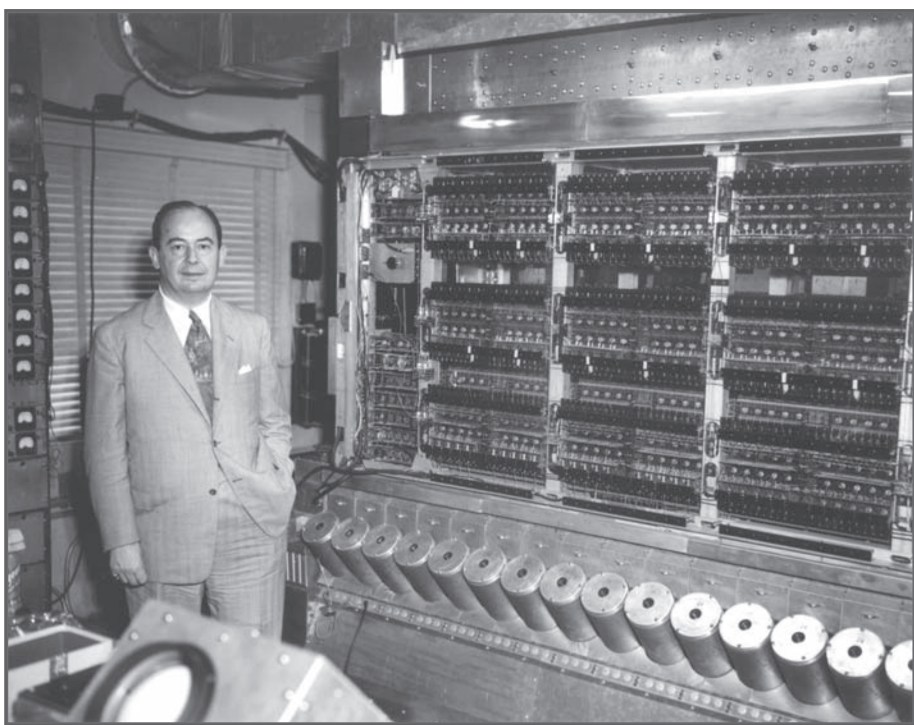


FIGURE 02.07

John von Neumann with the IAS computer at Princeton's Institute for Advanced Study. Computer historian William Aspray dates the picture to 1952, perhaps at the machine's official dedication. The cylinders running across the bottom front of the machine are electrostatic storage tubes, based on a design by FC Williams of Manchester University. They made up the machine's fast-access memory. Early first-generation computers adopting the IAS design had colorful acronymal names like ILLIAC, ORDVAC, MANIAC and JOHNNIAC. IBM also learned from the IAS in designing its System 701, the company's first all-electronic stored program digital computer.

Photograph by Alan Richards, courtesy of the Archives of the Institute for Advanced Study.

Officially dedicated in mid-1952 but in practical use since Spring 1951, the IAS machine (Figure 02.07) relied on some 3,000 vacuum tubes and weighed half a ton. The arithmetic unit could achieve around 30,000 addition operations and 1,600 division operations per second. The IAS was not the first digital all-electronic computer to successfully run a stored program. Computer history allocates that honor to machines built in England: the Manchester Small-Scale Experimental Machine (the SSEM) and Cambridge University's EDSAC (Figure 02.08). SSEM ran its first stored program in mid-1948, EDSAC by mid-1949.

Some of these calculating engines already could run programs of sequential instructions. These programs, however, typically resided outside the machine, coded as binary patterns of holes punched on paper tape or cards, or hard-wired into their circuitry. The program outside was read slowly into the machine, hole-by-hole, to guide the calculation from beginning to end—a sensible strategy so long as the hardware's speed of instruction processing did not tower over the speed at which instructions could be fed in. By the 1940s, vacuum tube switching elements (transistors and chips lay years in the future) could run instructions many times faster than punch cards or paper tape could input them. The digital electronic processors were hamstrung, so long as the computer program lay coiled up on a paper tape clunking through an external reader. The solution, so obvious in hindsight, was to store the complete program in the computer's circuits before starting the calculation. Program instructions could then run as fast as the machine hardware allowed. No more program bottleneck.

In the computing architecture named in von Neumann's honor, program and data sit together in high-speed memory. It is the computer as stored program machine. From the computer hardware's point of view, both the program and the data processed are forms of input information, and thus endowed with a strong family affinities. In the von Neumann architecture, electronic pulses from the fast-access memory feed program instructions to a hardware region, or logic/control unit, designed to trigger the calculation steps done on numbers in the arithmetic unit, into and out of which data whips from the fast-access storage locations in the memory (Figure 02.06). Arrays

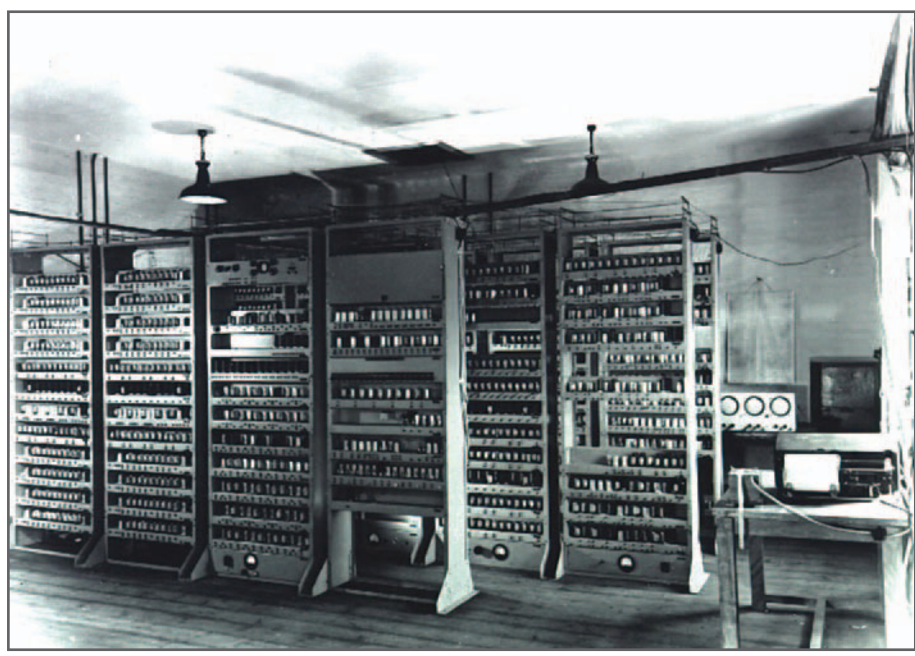


FIGURE 02.08

The University of Cambridge Mathematical Laboratory housed the original EDSAC, now regarded as the first all-electronic stored program digital computer to run application software on a regular basis. The imposing vacuum tube frames, clearly visible in foreground of the shot, implemented the system's arithmetic and control logic circuits. Memory capacity was 1,024 words of 17 bits, stored as vibrational patterns in liquid mercury. The operator's console, with three of the display tubes for inspecting the memory state (compare Figure 02.02), can be seen in the back right.

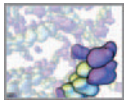
Copyright the Computer Laboratory, University of Cambridge. Reproduced by permission under Creative Commons License per http://creativecommons.org/licenses/by/2.0/uk/legalcode

of slower storage units, culminating ultimately in the input and output units we humans need to make sense of the digital events, keep the fast core fully primed. The IAS machine, the stored program computer built by von Neumann and his team at Princeton's Institute for Advanced Study to realize these ideas, became the template for many research and commercial designs. Serious use of analog computers would continue for at least two decades more, and the merits of radically different hardware architectures continue to be explored, especially for problems (like artificial intelligence) well suited to massively parallel manipulation of data. But the computer most of us sit in front of each day is a direct descendant of the IAS architecture and von Neumann's reports. To the extent that our global civilization is a postindustrial information economy, the engine of postindustrial commerce and innovation is the von Neumann machine.

Conditional control

The circuits of EDSAC and its kin could be wired up to perform the basic steps of arithmetic and of switching logic such as "AND" and "NOT." By sequencing these logical atoms adroitly, the power of the stored program would be limited only by the expressive power of logic and arithmetic itself. About a decade before these breakthroughs in computer engineering, the British mathematician Alan Turing had proposed that all computable information patterns could be generated as the output of elementary operations such as these, provided the operations were carried out in the right sequence. Turing showed the existence, in the form of a theoretical mathematical concept, of a computing machine (now called a Turing machine) capable of carrying out the task with the aid of a suitable program code. Stored program

Alan Turing, b. England 1912—d. England 1954. British logician and mathematician celebrated for his contributions to the formal theory of computation and the foundations of mathematical biology. Like von Neumann, Turing died far too young.



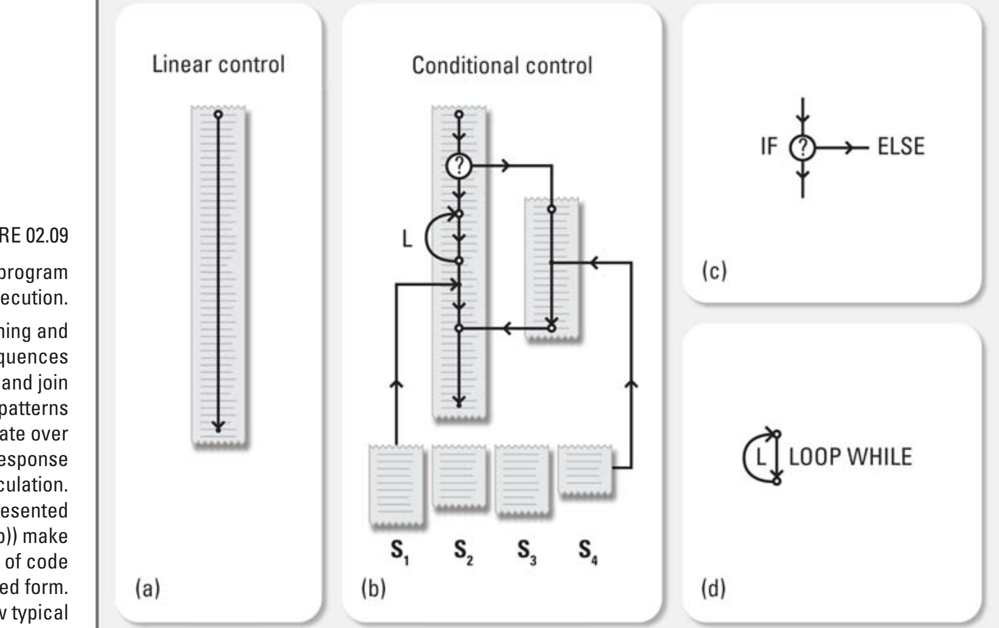


FIGURE 02.09

Conditional control of program execution.

Commands for branching and looping (L) take linear sequences of instructions (Panel a) and join then into more complex patterns (Panel b) that can iterate over cases and vary program response if needed during a calculation. Subroutine libraries (represented by S_1, S_2, S_3, S_4 (in Panel (b)) make frequently used chunks of code available in pre-packaged form. Panels (c) and (d) show typical branch and loop operations.

machines like the EDSAC were a practical realization of Turing's abstract "universal machine".

von Neumann and his colleagues realized that effective use of a stored program computer requires a well designed machine language: some arrangements of programming instructions are better than others. The program commands should allow exact but flexible routing of the hardware operations through the problem-solving task. The programming language should allow the program to branch between alternative pathways if certain conditions arise in the algorithm, and to jump or loop through sets of instructions when cyclic progressions are needed in the data processing or data array manipulation (Figure 02.09).

Computer languages since EDSAC embody this vision of conditional control. They allow the program code to switch among multiple streams of activity given suitable triggering events and provide for jumping and looping within the instruction sequence. Programming languages differ, however, in the relative complexity of the branching operations and looping types allowed by their command syntax. One of us (CJL) is old enough to have programmed in languages—the early Fortrans—in which conditional control was pretty much limited to IF ... , looping to DO ... , and the pleasures (plus nascent disasters) of jumping to the since much maligned GO TO ... command. Following the heritage of modern high-level programming languages, the MEL command set lets you exert a subtle diversity of control over your program's flow of information processing events. We will introduce these to you in *Chapter 12: MEL Scripting* in Part 2 of the book, where you'll meet conditional control variants like the WHILE ... and SWITCH ... CASE ... commands, and at once start using them to model and visualize biological projects in Maya.

```
++942343210    00              0000    11
97301233221001221 0
100012100110001 10
111110                      00       0
          00000100000                00   00
320   01110                    00110  012
431   13321     00000          0000   023
11    1232100111222 10                 00
331000110000000000
110       00110000
              00110000
   00000        00000000
00        0     00000001111100
10      0110000000111110011 10       000
11    01100  0000011110     010       00
0       0           00110

 000      0111000                   00
1110     01111100         0   001100000
1110000000   000000110       011000112
000             000    00    022210 011
            00              0122100001 1
                                000100
                     00        0111
                     1110    01110000
10000     000000     01110011210001 1
10000     0110000   00       1233210011
    0     010        000   01221100001
    00                 110   011100
           0110      010      01110
           011       01100    01110011
    00   0000     0022211110122211 23
(a)
```

(b)

FIGURE 02.10

Computer-based interpretive visualization at work.

(a) In 1952 Cambridge chemist John (later Sir John) Kendrew (1917–1997) and student John Bennett published this contour map tracing structure in the oxygen storage protein myoglobin.[4] They programmed the EDSAC computer to analyze myoglobin's crystal diffraction pattern and print the visualization. This is one of the earliest computer-generated images in the study of large biological molecules.

(b) The original EDSAC (which was followed by the yet more powerful EDSAC 2) supported structure mapping to a resolution of 6 Angstroms. Stacking the contour maps gave the world its first look at the myoglobin molecule. Kendrew's success in deciphering the 3D structure of myoglobin would earn him a Nobel Prize.

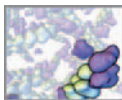
(a) From reference 4. Courtesy and © the International Union of Crystallography.

(b) Courtesy and © MRC Laboratory of Molecular Biology, Cambridge, UK.

The computed organism

The 20th century's revolutionary decades of computer and software engineering were also years of astonishing progress in the sciences of biochemistry and cell biology. Some of the exciting details we'll explore in the chapters ahead. Here we can briefly consider the implications of the basic fact, now universally accepted in the scientific community, that all living things—all organisms—are composed of chemicals, that is, molecules whose complex interactions set in motion the processes of life. There is no evidence for some mysterious, supernatural "life force" acting alongside the chemistry of living matter. A deep understanding of biological molecules and their interactions appears necessary and sufficient to answer the question "What is life?" if by that question we are seeking to understand the physical mechanisms sustaining biological activity.

Civilizations equipped with powerful computers must therefore be in a position to greatly accelerate the rate at which they gain understanding of living matter. Cells obey the laws of chemistry and chemistry obeys the laws of physics. The laws of physics and chemistry are, in turn, intensely and fundamentally mathematical. One must conclude that mathematical calculating machines—capable of large scale computing based on enormous volumes of data and information—can be used to analyze, predict, and ultimately re-design the biochemical activity of cells and organisms (Figure 02.10). The machines can do this by invoking mathematical operations to precisely represent the biochemical mechanism and calculate its properties (Figure 02.11). Computational biology is the application of computing to explore these intricate links of structure to function in living things. It is one of the fastest growing frontiers of biological and medical research.



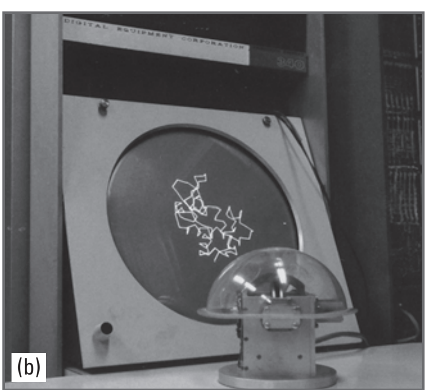


FIGURE 02.11

By the mid-1960s, mainframes and minicomputers were harnessed to interactive displays of protein and nucleic acid structure. The first cutting edge was defined by MIT physicist Cyrus Levinthal (1922–1990) and his students. They used the ESL (Electronic Systems Laboratory) Display Console, then known locally and ever after as the "Kluge", housed at MIT. Either directly or through a DEC model PDP-7 minicomputer, the 7094 mainframe at MIT's Project MAC drove a DEC Type 330 monochrome display in real time, producing vector images with 1024 × 1024 addressable-point resolution. The pipeline between the Kluge and the mainframe ran under the 7094's CTSS (Compatible Time-Sharing System), the first operating system to let multiple users to log on at once via dedicated terminals. Levinthal's team wrote software for building and rendering graphical descriptions of 3D protein structure on the Kluge. Levinthal's colleague Robert Langridge, then at Harvard, pioneered nucleic acid visualization on the Kluge.

(a) The DEC minicomputer and its tape units are in the background, teletypewriter in the middle, and ESL graphics scope and "Globe" 3D controller in front. By twisting the Globe, users could spin the molecule's image to a different 3D viewing angle.

(b) Close-up. The scope is showing the carbon atom backbone of the protein lysozyme. The Kluge ran its own visualization software, called GRAPHSYS, in turn authored in AED-0, a descendent of the Algol programming language. A core reference for the Kluge is the 1968 report by Thornhill and his Project MAC colleagues, which as of this writing is archived on line at the Bitsavers Project. See http://www.bitsavers.org/pdf/mit/lcs/tr/MIT-LCS-TR-056.pdf.

Photographs by and permission of Martin Zwick.

For much of the last century there was slow progress in teasing chemical information out of living matter. Now, exceedingly effective methods allow dozens or hundred or thousands of chemical species to be mapped at once inside the living cell's network of biochemical and genetic interactions. So much information is now available that scientists and health care researchers speak hopefully of grasping the systems biology of the living cell that is the totality of its operational biophysics and biochemistry. Computers are essential to systems biology research: there is too much data to organize, sift, and communicate by hand, and only computational biology offers a hope for deciphering the fundamental biochemistry these huge volumes of data encode. Each project chapter of this book will take you on a step through the use of Maya and MEL as exciting tools in computational and systems biology.

The computational organism

The computer pioneers were not alone in their obsessions with information and the secrets of its many transformations. Following 1900, an astonishing range of disciplines appeared in rapid succession to answer the call for new methods of managing complex events in nature and society. Each new discipline offered a vision in which matter, energy, and human behavior became abstract patterns of information ready for analysis and control: systems science, cybernetics, mathematical economics, control engineering, communications design, information theory. It was a large vanguard and a new technical vocabulary: information, control, regulation, feedback, communication, error, stability.

Those were heady times. For a while it seemed a new general science of information and control might leap into existence, providing the matrix by which complex systems of all stripes—natural and cultural—would fall smoothly into line for rational analysis and management. That an all-embracing mega-science failed to emerge is no slight on the accomplishments of each specialized field. Computational biology and systems biology, which we met above, number among the worthy offspring of this turbulent period. The paradigm of system/information/control influenced disciplines then far removed from computer engineering.

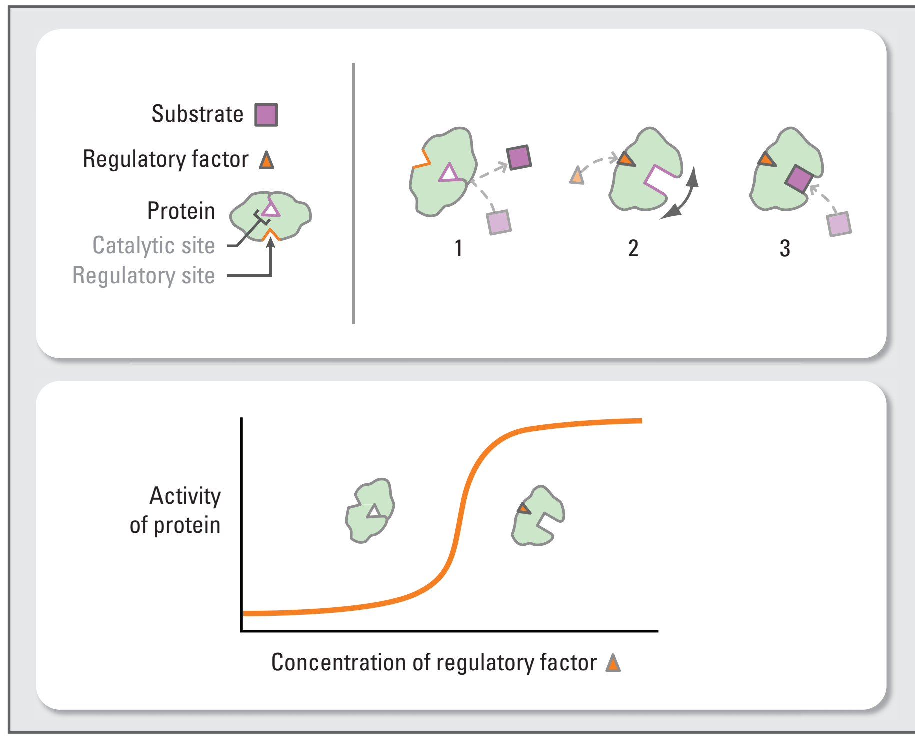
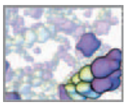


FIGURE 02.12

In the living cell, sensor regions of large molecules such as proteins bind other molecules acting as regulators. The response is often switch-like: the building block of cell activity regulation.

Prime case in point: reporting his discovery of biochemical control by negative feedback (where the product molecule of a biochemical pathway cuts down the rate of the pathway's first step), Edwin Umbarger, writing in 1953 in the pages of the august research journal Science, considered it natural to open his account with the importance of feedback loops in regulating industrial automation. No doubt the chemists and biologists intent on Umbarger's biochemical findings at once grasped the analogy. A little more than a decade later, so many cases of biochemical self-regulation by feedback had been reported that the same journal published a lengthy review. Today, 50 years after Umbarger, we understand that the logic of conditional control governs more than the action of computer programs and industrial assembly lines. It organizes and regulates the self-sustaining activities of the living cell. This includes branch points on diverging biochemical reaction pathways, at which conditions are tested and alternate outcomes selected for the activities of genes and proteins; loops, cycling the downstream effects of biochemical pathways to upstream feedback control points; and biochemical subroutines, triggering of entire modules of successive reaction steps by single control inputs.

Chemical response to other molecules provides the "hardware" for conditional control of cell activity. The biochemical hardware often acts in a binary, off/on manner. In the classic off/on response pattern illustrated in Figure 02.12, a small molecule called a regulatory factor binds to a specific site on a protein. The protein recognizes the factor. The recognition event triggers changes in the protein's shape, in turn altering



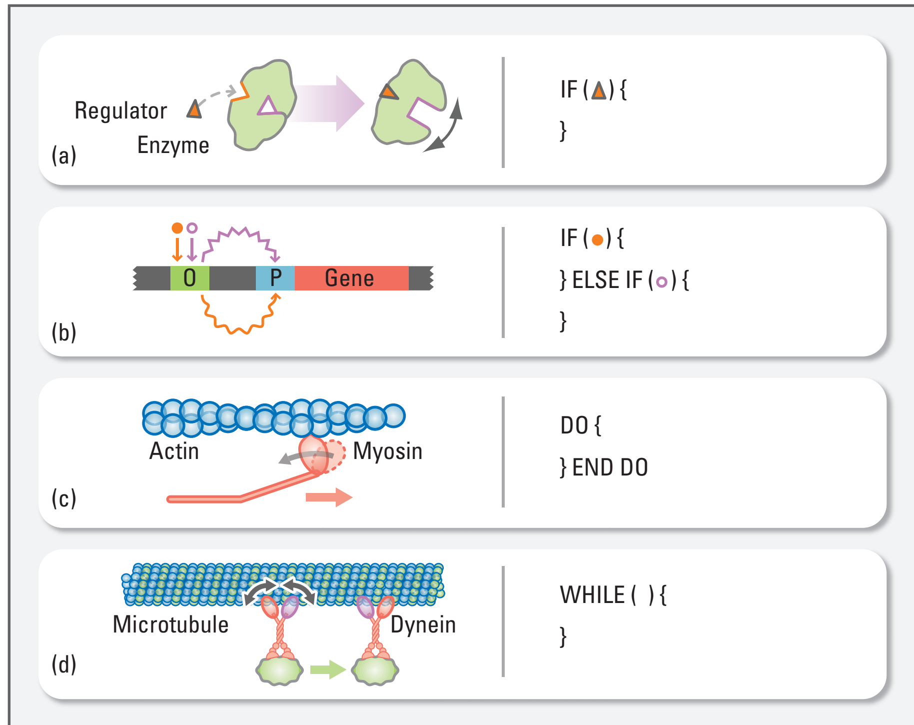


FIGURE 02.13

Binding of regulator factors produces chemical effects whose logic structure parallels familiar commands for programming computers.

the activity of its catalytic site. As you can see in the graph, the shape of the response resembles an S-curve: the protein acts like a binary switching element in its reaction to the input signal.

By combining these binary switch-like responses across successive chemical reactions, the cell can build up biochemical pathways that carry out more complex tasks of conditional control. We have illustrated some examples in Figure 02.13. In (a) the binding of a regulatory factor to a protein enacts a biochemical "IF ... THEN ..."; i.e. IF (factor bound) THEN {increase catalytic rate}. In (b), the binding of alternative gene regulatory factors to an operator region O of a genome causes the gene sequence region to be read off at different rates, enacting a biomolecular IF; ELSE IF .... P is the gene's promoter region, where the read-out starts. In (c), the repeated cyclic binding, release, and motion of the myosin molecular motor along an actin filament (as in muscle contraction) gives a protein-encoded DO loop: DO {repeat motor cycle until muscle cell contraction signal stops}. Elsewhere in the cell (d), a two-footed dynein molecular motor walks along a microtubule, dragging its cargo molecule (shaded block) from a source point in the cell to its delivery point: WHILE (no end-of-route signal detected) {continue walking}.

Cells therefore may be chemical engines, but they are also information processing machines self-assembled from complex organic molecules. The cell's DNA molecule encodes its genetic information. The activity of its biochemical network encodes

FIGURE 02.14

Bjarne Stroustrup, the inventor of C++.

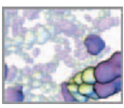
Courtesy Bjarne Stroustrup. Used with permission.

metabolic and behavioral information. Nature and computer engineers, working with vastly different materials and at very different size scales, have converged on common strategies for managing information flow. von Neumann, with key mysteries of computer design solved and new interests calling, was by the late 1940s at work on a new mathematical theory of self-reproducing machines. The treatment would have placed the living cell's puzzling ability to duplicate itself in the clarifying light of mathematical logic. To history's loss, a fatal cancer took von Neumann in the midst of his new project, leaving just partially completed notes for others to carry on. One can only wonder what computers and computational biology would today be like if this remarkable polymath had lived on, into the era of the DNA double helix, the genetic code, the Human Genome Project, and integrative systems biology.

OOPs and agents

In some applications the computing process is used to change one pattern of information (the input) into another (the output). The output might be a set of numbers (in a scientific calculation) or a string of text (as it will be in the automated typesetting of this book). The output from Maya is a sequence of number arrays, each array defining the grid of pixel colors that make up one digital image from your computed 3D animation sequence.

In other instances the process of information change is more transactional, as for example in a video game or in an airline ticket reservation system. In these transactional



The current lingua franca of hard-core higher-level programming, C++ (Bjarne Stroustrup, Bell Labs, 1983 (Figure 02.14)) accommodates both procedural (sequential command) and object-oriented code design.

systems many patterns of information are changing all the time. Each transaction may be thought of as an event in which some part of an overall system-wide information store is updated or changed. Although formally each event has an input and an output, designers of transactional procedures are very concerned with the accurate and efficient orchestration of the overall flow of information among the events. The idea of computation as transactions among automated parcels of computer code is the heart of object-oriented programming (OOP), an influential movement in modern approaches to software design.

The Simula-67 high-level language (Ole-Johan Dahl and Kristen Nygaard, 1967, Norway) for discrete event simulation introduced programmers to key OOP constructs such as the object and the class. Smalltalk (Alan Kay team, Xerox PARC, 1971–1972) is considered the first (and perhaps only) pure OOP language.

A moment's reflection tells us that the OOP viewpoint is also well suited to describing how information is processed in living systems. The organs and tissues of our bodies are built from cells that coordinate their activity by exchanging chemical and electrical signals. Within the cell, the living material is composed of intricate chemical networks whose pathways signal each other. In all these cases, one's description of the biology is based on observable units (cells, modules of reaction pathways) interacting with each other in a transactional manner. High-level languages designed for OOP, like C++, are therefore a popular choice in computational biology. Although MEL is not designed as an OOP language, its command structure gives you immediate access to the huge diversity of software objects and methods comprising Maya. MEL's vector and array data types allow you to map the structure and interactions of cells and biochemical reactions without resorting to OOP conventions. For projects demanding an OOP coding practice, you can assess Maya's C++ API.

It is interesting to speculate that, under the impetus of new applications like those you'll carry out here, future versions of MEL will deploy expressly object-oriented capabilities—let us call this future language for Maya programming MEL++. Time will tell. Meanwhile, current MEL has serious capability for computational modeling in biology. We are going to show you how to harness that power by factoring MEL's command structure against an OOP-inspired breakdown of your biology problem according to its functional transactions and the entities interacting through these signals and messages. The resulting workflow, you will see, is an efficient means of organizing the complex data you must handle in any biology project (Figure 02.15).

As you progress, we will have you take this approach a step further. The transactional paradigm behind OOP lacks a natural vocabulary for situations in which the objects not only signal one another, but move around, explore, and modify their environment while doing so. Such objects behave more like mobile agents—self-directed robots—than like fixed nodes in a communication network. This is of great biological relevance. Important types of cells, such as immune cells, blood cells, and cancer cells, are highly mobile within your body. They are not fixed in one place. Maya and MEL, with their diverse tools for describing and simulating 3D motion, are well suited to computational problems based around mobile agents. Doing the projects in this book will help you develop an agent-oriented approach to assessing and solving problems in computational biology, and will assist you in understanding when an agent-oriented programming (AOP) workflow is to be preferred. As this book goes to press, AOP is one of the hot frontiers in software design and program language development.

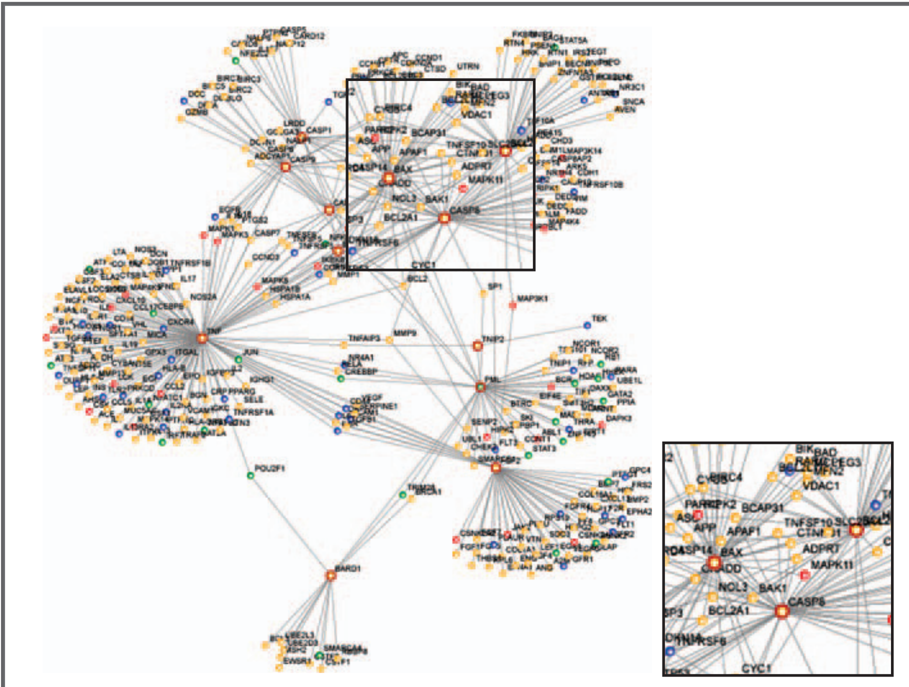


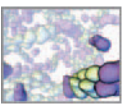
FIGURE 02.15

Some of the protein–protein interactions active in a living cell's molecular program for self-destruction. The self-destruct program can be triggered by excessive cell damage or certain disease states. Compared to von Neumann machines with their deterministic linear chains of instruction fetch-execute steps, cells organize their information transduction actions into dense stochastic clusters carried out in parallel.

These protein networks were set up and visualized using the Human Interactome database and browser toolset public bioinformatics resource (http://www.himap.org) of Rhodes, Chinnaiyan, et al[5] at the University of Michigan.

Summary

Computers are information processing machines, as are living cells. Despite the differences in their size, construction materials, and power sources, computers and cells use some similar strategies to control their information processing activities. But while the development of computers has already advanced to the level of high-level programming languages, our understanding of the cell as an information machine is much more primitive. Cell science provides some glimpses of the cell's "machine code"—the low-level nuts-and-bolts of the genetic code and of signal transduction through individual biochemical pathways and chemical reactions—but so far no hints at all on the mystery of the cell's "high-level programming language", its properties, and even whether one can speak accurately about the design of living matter in this way. These are among the exciting scientific problems to be studied with the aid of the methods you will learn about in this book. In order to step into the Maya user interface and so into the world of MEL programming, we must turn to the basic concepts and vocabulary of 3D animation and computer graphics. Let us now do that and then, via Maya and MEL, see where the scientific path leads (Figures 02.16 and 02.17).



Biological problem

Data

Hypotheses

History

Mining

Analysis

Prediction

Education

Physical principles

Chemical principles

Outcome needs

Mathematical ideas

Visualization ideas

Mathematical model

Visualization model

Numerical methods

Visualization methods

Biophysics algorithm

Visualization procedures

Visual simulation code

Procedural animation libraries

3D models

World model

Lighting and rendering

Render

FIGURE 02.16

The in silico biology workflow.

FIGURE 02.17

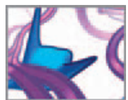
Modern desktop systems are von Neumann computing machines many times more powerful that the early mainframes. Here, the authors (left CJL, middle NW, right JS) discuss MEL code for a Maya-based model of cell motion.

Courtesy and copyright © 2008 Eddy Xuan.

References

1. Campbell-Kelly M: The Airy tape: An early chapter in the history of debugging. *IEEE Annals of the History of Computing* 14: 16–26, 1992.
2. Wilkes MV, Wheeler DJ, Gill S: *The Preparation of Programs for an Electronic Digital Computer: with Special Reference to the EDSAC and the Use of a Library of Subroutines*. Addison-Wesley Press, Cambridge, MA, 1951.
3. Croarken MG: The emergence of computing science research and teaching at Cambridge, 1936–1949. *IEEE Annals of the History of Computing* 14: 10–15, 1992.
4. Bennett JM, Kendrew JC: The computation of Fourier syntheses with a digital electronic calculating machine. *Acta Crystallographica* 5: 109–116, 1952.
5. Rhodes DR, Tomlins SA, Varambally S, Mahavisno V, Barrette T, Kalyana-Sundaram S, Ghosh D, Pandey A, Chinnaiyan AM: Probabilistic model of the human protein–protein interaction network. *Nature Biotechnology* 23: 951–959, 2005.

This page intentionally left blank



3D model of DNA molecules.

03 Animating biology

Introduction

We have been introduced to the need for dynamic visualization in science, considered a basic framework of biological organization, and explored the relationship between biology and computation. How then do we integrate these science-inspired notions with the art of computer animation using Maya? And what is animation, anyway?

To animate is to give life to an otherwise inanimate object.[1] For our purposes in 3D computing, the verb refers to a change over time in a property of a given item, rendered into a succession of still images, or frames. Somehow, our visual system is able to view this succession and create, in our minds, the perception of motion and behavior in the depicted objects. So we will begin this chapter with a brief look at how we see and how we perceive motion in animated images. You'll then meet some of the lexicon and methods of animation, and how and why they might be adapted to scientific visualization. A 3D computer animation workflow will be laid out to give us the roadmap to learning Maya and MEL, placing them in the overall process of 3D animation production.

Since Maya is designed for animation, its effective use follows the workflow you'd find in a professional animation studio. This workflow, which we'll explore in this chapter, applies terminology and conventions that draw (pardon the pun) on modes of practice honed over decades in the ateliers of animation pioneers like Walt Disney and the Fleischer Brothers. These terms and conventions may seem strange, at first, if you are coming to Maya from a science or engineering field. But we hope to show you that there are benefits to the co-option of cinema-oriented animation practices by scientists' intent on understanding biological phenomena.

Experienced animators may encounter some familiar topics in this chapter, but will still benefit from the connections drawn between the animator's and the bioscientist's workflow. By the end of this chapter you should be able to see how animator's techniques can potentially apply to the science discovery process, and have a broadened appreciation for the experimental and expressive power of modern digital animation tools.

Animation and film perception

In film and video, there is no motion, there is only a succession of still images, rapidly displayed. Yet we perceive motion. How is this possible?

Seeing, in brief

The early, "anatomical" stages of vision are fairly well known; we will sketch them below as means of explaining the raw processing power that vision brings to the world. The sensory impulses originating in the retina are ultimately transformed into a skein of neuronal activity in the brain that presents to our consciousness an integrated picture of the world. This latter process—how biochemical fluxes turn into meaning—is the hard part, and a something we can't hope to address here.

Five crucial events make up the initial stages of vision (the structures described below are illustrated in Figure 03.01):

1. Light originating from the sun or some other source scatters through the environment, bouncing off various objects. A small fraction of this light happens to pass through the pupil of your eye (a distensible hole that varies in diameter between 2 and 6 mm and is analogous to a camera's aperture).

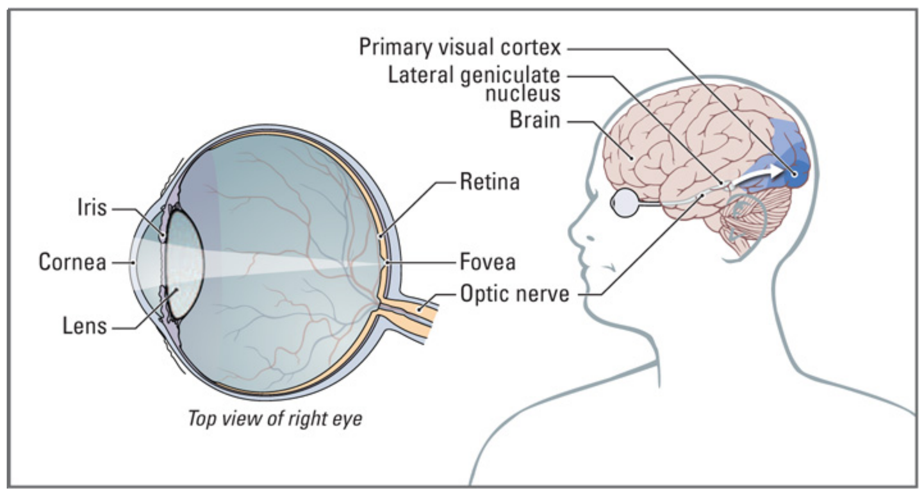


FIGURE 03.01

The anatomy of vision. Light enters the eye and is focused upon the retina, where it is transduced into electrochemical signals by the photoreceptors. The signals travel via the optic nerve, through the lateral geniculate nucleus, to the primary visual cortex in the occipital lobe of the brain. Here the signals are rapidly dispatched in a parallel fashion to various processing centers (V1, V2, etc.) where salient features of the scene are extracted, and ultimately integrated into a coherent internal representation of the scene.

2. The light is refracted by the lens in your eye (as well as the cornea and the refractive gels that fill the eyeball) to focus a high-resolution two-dimensional (2D) image on our retina, a complex nine-layered tissue comprised of photoreceptors, blood vessels, and multiple layers of neurons. These neurons form the earliest stage of visual processing: they take the light pattern detected by the photoreceptors and perform various tasks, such as enhancing the contrast at edges, and suppressing jittery motion.

A **neuron** is a cell type essential to the operation of our nerves and brains. Neurons are electrically excitable, and conduct signals from one part of our body to another.

3. Impulses derived from the photoreceptors, and modulated by those early neurons, are relayed to the optic nerve. The optic nerve passes through the back of the eye (creating the blind spot), and after passing through the optic chiasm (a "neural traffic interchange" where the left and right visual fields from each eye are united and sent to the appropriate hemisphere), the signals are sent to the lateral geniculate nucleus in the thalamus (a deep brain structure). Early stages of color and motion processing occur here.

4. From the lateral geniculate nucleus, the nerve impulses are passed to an area near the base of the occipital lobe known as V1 (visual area 1). The cortical processing of vision occupies about 40% of the gray matter of our brain, and V1 is the first station in that process.

5. From V1, the neural activity is distributed to brain regions V2, V3, V4, V5, and onward. In a feat of massively parallel processing, an astonishing array of features are rapidly extracted from the 2D image received on the retina: edges are detected, objects are separated from the background, depth is assessed through multiple concurrent cues, the direction and magnitude of motion is appraised, faces are recognized, and various salient object features (orientation, size, color, and texture, among others) of the scene are assessed. In addition, remarkable feats of "mental construction" are accomplished, as 3D stereoscopic depth is created from the divergent images originating from each eye, and partially represented (or, in the case of some visual illusions, even non-existent!) objects are built from fragmentary evidence in the scene.

Less than one-fifth of a second has elapsed since the light wave reached the retina at the back of the eye.

In the moments that follow, your brain integrates this initial decoding of the visual scene with other elements of your conscious awareness, calling on the power of your memories, reason, and emotions to interpret it. How astonishing that so much complicated processing occurs so quickly, and with so little deliberate effort!

The fact that vision requires so much active (if unconscious) creation on the part of the viewer helps to explain how we end up imparting so much meaning to cinematic stories; we will see another aspect of the "creative" abilities of visual perception in the next section.

Seeing motion and animation

It may be hard to believe, but the nature of motion perception is still under active investigation.

Many film theory textbooks still claim that the basis for motion perception in film is a phenomenon known as persistence of vision, where the image falling on the retina persists biochemically over some interval until it is replaced by succeeding images; this overlap allows the images to blend, retinally, into "motion". While some biochemical truth lies behind this idea—photoreceptors in the eye do continue to signal for some time after the stimulus has passed—the idea of persistence of vision has largely been replaced by a more comprehensive understanding of mechanisms of motion perception.[2] Newer accounts of motion perception call for a more active engagement from the viewer, and rely on multiple, overlapping mechanisms.

Two of those mechanisms—flicker fusion and short range apparent motion—are worth spending some time on, as they relate rather directly to the standard frame rates that animators use in the production of their films.

Film has a frame rate of 24 frames per second (fps), but each individual frame is actually flashed onto the screen two or three times in succession, leading to a flicker rate of 48–72 Hz (times per second). Why is this? It turns out that in order to present the sensation of viewing a continuously illuminated screen, the projector must flash on-and-off rapidly enough to achieve flicker fusion. This is a phenomenon whereby a flickering source of illumination will, at some frequency, fuse into the perception of continuous illumination. If you watch a strobe light flashing at an increasing rate, at some point the discrete flashes will fuse into what seems like a light that is simply "on". The rate required for flicker fusion varies depending on a number of factors (e.g. whether the flicker is present in central or peripheral vision, the brightness of the illumination, the fatigue of the viewer) but is usually in the range of 50–60 Hz for film and television applications.

This helps explain how frame rates for film and video were determined: if each frame of a 24 fps film were shown only once, the image would "strobe" in a way that would make it very difficult to watch, so each frame is projected more than once. In North American video (NTSC), which is nominally 30 fps (actually 29.97 fps), each frame is composed of two interlaced sub-frames displayed in sequence, leading to a 60 Hz flicker rate.

European (PAL) video is 25 fps, leading to a 50 Hz flicker rate.

Does flicker fusion explain motion perception? No, it simply explains how many projection and display technologies can appear to present a continuous image, rather than one that is strobing. It is worth noting that most newer flat panel display technologies, such as LCDs, do not flicker, since they are continuously illuminated by their back lights. Any digital display, however, has a refresh rate, separate from the potential flicker rate of the display, that is determined by how often the displayed image is updated by the underlying graphics circuitry.

Apparent motion is the term for several distinct phenomena, initially discovered by vision researchers in the early 20th century, in which certain configurations of rapidly displayed still images could precipitate a perception of motion. Short range apparent motion requires fine-grained changes between successive images, as is the case with most film and video.

The perceptual mechanisms underlying short range apparent motion appear to be identical to those active in perceiving real-world motion. Clinical evidence for this comes from the cases of unfortunate individuals who have experienced damage to the part of the brain that allows them to perceive the shape of still objects; these patients cannot recognize, or even see, objects that are not moving. Once an object moves, however, it pops into existence in their preceptual world. Fascinatingly, these individuals can also perceive moving objects in television and film, despite the fact that, of course, they are really seeing a sequence of still images. This demonstrates that, in these subjects, and probably people in general, the mechanisms of film perception are the same as, or very similar to, that of general motion perception.

So, the mystery of film motion perception is starting to yield to scientific study: animated representations of motion are similar enough to real-world stimuli that they engage the same perceptual mechanisms that real-world motion does. Two of those mechanisms—flicker fusion, which in some media makes still images appear as continuous, and short range apparent motion—help to explain why animation is usually crafted as sequences of 24–30 still images-per-second, with relatively small differences between individual frames. Later in this chapter we will look at animation frame rates, and in which contexts it might be advisable (or not) to vary them.

The animator's workflow

Story: The workflow's driving force

The animator's workflow (Figure 03.02) is a time-tested approach to the highly economical depiction of an idea expressed in a finished scene or in the completed film. "Economical" is not used here in a financial sense, but in terms of efficiency: animation is generally so labor intensive—even when powerful computers and software are used—that many benefits accrue to animators who do what they need to, but no more than is necessary; spending weeks diligently animating a scene that is ultimately left out of the film is a painful, costly experience. Thus the necessity of an organized approach to the animation process. This is a process of experimentation and refinement, and it is very much a part of the computer animator's approach as well, where modern digital tools can somewhat speed the iterations shown in Figure 03.02.

Many scientists experience similar cycles as they refine experimental protocols, improving results and explanatory models (Figure 03.03). At the heart of science is humanity's yearning to make sense of the world, in a way we are all free to understand and to evaluate in a logical, testable manner. This notion of making sense, of arranging objects and concepts in plausible causal chains, is also at the heart of storytelling.

The broad range of stories—those limited only by the teller's imagination—encompass a much larger range of possibilities beyond the narratives of reality told by science. Science, of course, is concerned with what is, with actuality and truth. But like the great novels, plays, and myths, important works of fictive cinema—animated and otherwise—also illuminate truth.

The 3D animator's workflow

Preproduction

Idea

Treatment

Script scene/shot breakdown

Revision

Storyboard

Voice/narration

2D animatic/story reel

Production

Layout/3D animatic

Model

Animate

Texture and light

Render (Passes)

intro_232.tga
intro_233.tga
intro_234.tga
intro_235.tga

Postproduction

Composite

Edit

Encode/package for delivery (DVD, webcast, broadcast, etc.)

FIGURE 03.02

A typical workflow for computer animation production.

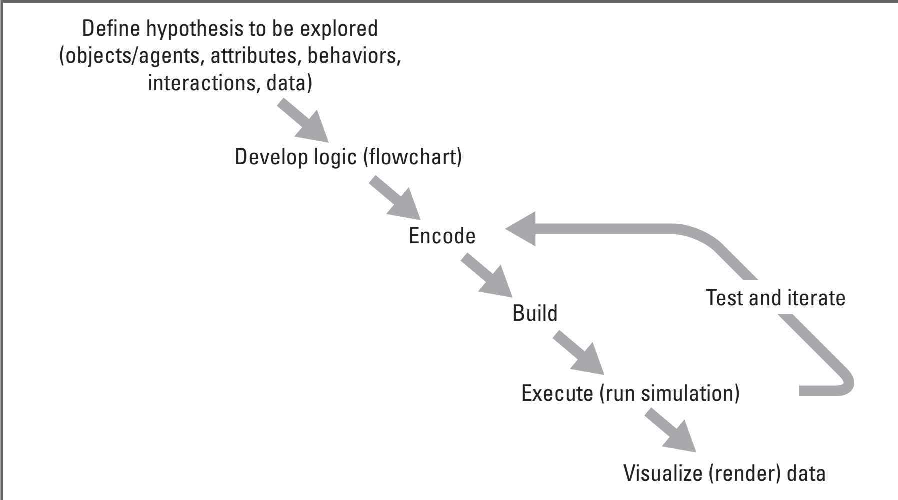


FIGURE 03.03

A typical science animation/simulation workflow. We will be adapting this approach and using it in combination with the traditional animation workflow illustrated in Figure 03.02.

Experiencing fictional worlds helps us make sense of human existence, its glory, and its foibles. Filmmaking of the "Hollywood style", for example, is all about telling "readable" stories—the struggles and conflicts of exciting imaginary characters—written, shot, and edited in ways that allow them to be easily interpreted. This ease of interpretation is partially a result of the set of production heuristics that have evolved over the history of filmmaking.

When discussing storytelling in the Hollywood style, we forego for the moment movements in art and literature, such as Dadaism, that have sought to undermine conventional storytelling requirements.

Since these established animation production workflows support filmmaking economy, as well as film "readability", we can adapt them to support the needs of thoroughly actuality-driven enterprises like science research and scientific communication.

The three-stage workflow

The animator's initial task is to critically formulate the story idea to be communicated and then explore and refine potential approaches to its expression. This stage is known as preproduction. At the end of this stage, the animator has a solid plan for the execution of the project. Preproduction leads to the production stage. Here the animator implements the plan generated in preproduction and creates the resources necessary to complete the film. This is generally the longest and most labor-intensive part of the process. Postproduction follows, where the media developed in production are assembled (edited together) and refined into a form suitable for final delivery. This usually takes the form of a film or film segment intended for theatrical release, for television broadcast, release via the web or podcast, presentation at a scientific meeting, or a video game cut scene.

Let's explore this workflow.

Workflow stage 1: Preproduction

Animation is one of the least direct of the visual arts: the animator works at the drawing table or computer screen, preparing countless images, intending that the

arrangement of those images in time will evoke the sense of life that is sought. Preproduction involves all the project initiation steps, as well as the development of a coherent plan for the completion of the film. Effective preproduction helps assure that suitable animations are produced in the fewest possible steps of execution and revision. The key elements of effective preproduction are the defining your animation's visual style or "look", the treatment and the script, the storyboard, and the 2D **animatic**.

The animation's "look"

A cancer animation that depicts the deadly cells as whimsical, anthropomorphic cartoon characters might work well for an audience of young children. However, it might also leave an audience of investment bankers ready to fund a new cancer drug less than impressed. Such qualities of overall visual appearance are often referred to as your film's "look", a subject demanding careful attention right at the start of your project. The message or point of a film can be misunderstood, or lost entirely, if poor decisions are made about its appearance or look.

Let's consider three of the most popular looks for cell science animation. The list is by no means exhaustive; indeed, one of the benefits of working with a 3D program is the possibility of creating entirely new, never-seen-before representations.

1. **The photorealistic look**. Photorealism is a term from art criticism which refers to pictures which attempt to emulate the qualities of photographs. As a movement in 20th century painting, it has been associated with the work of artists like Chuck Close and Mary Pratt. Photorealism consciously imitates the effects of optical lenses and adheres closely to the rules of vanishing point perspective, creating solid, believable images. In addition to the surface qualities of light and texture familiar to us from our experience of the real world, photorealism often includes artifacts of the photographic process, such as depth-of-field effects, motion blur related to shutter speed, compressed dynamic range, and lens flare. One version of the history of computer graphics research would see it as a progressive march toward the goal of seamless, true photo-like rendering (which has, arguably, recently been achieved with unbiased, light simulator-style rendering engines like Maxwell, from Next-Limit Technologies). A broader look at computer graphics would see computer-generated imagery (CGI) encompassing a number of representational styles.

2. **The micrographic look**. A sub-style of photorealism, devoted to emulating micrography, has emerged over the past several years: this is the micrographic look (the appearance of objects as seen or photographed through microscopes). At the present time in human history, photorealism is a near-universal strategy for depicting the events of our everyday lives in the entertainment and news media. The camera's "eye" is ubiquitous. The conventions of photorealistic depiction have therefore been adapted by artists and scientists to reveal objects and events too large (as in astronomy) or too small (as in cellular medicine) to be seen with the unaided eye. Sometimes, although objects are small, they are still big enough to deflect light rays. Most intact cells are big enough to do this, so with special lenses and other imaging technologies (microscopes) we can magnify their images and take their picture. Other subjects require more exotic preparations and techniques; for instance, researchers have begun to use small fluorescent proteins (such as green fluorescent protein, derived from a jellyfish) to "tag" cellular components they wish to observe. The resulting micrographs are often hauntingly beautiful (Figure 03.04).

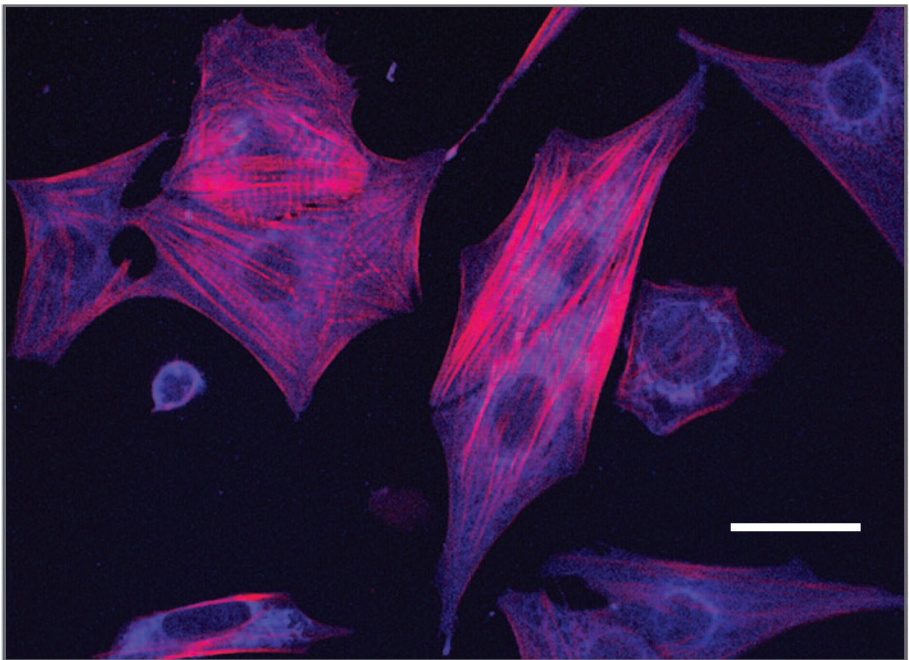


FIGURE 03.04

Cell micrographs can often be appreciated for their intrinsic beauty, quite apart from their obvious utility as scientific objects. This image shows cardiomyocytes (or heart muscle cells) that have been specially stained to make the cellular proteins actin (red) and calreticulin (blue) visible to the microscope under special lighting conditions.

Scale bar ≈ 10 μm

Images courtesy and copyright © 2006 Sylvia Papp, Institute of Medical Science, University of Toronto and Michal Opas, Department of Laboratory Medicine and Pathobiology, University of Toronto. From research supported by the Canadian Institutes of Health Research (CIHR).

A popular approach for animators working at the cellular and molecular level therefore is to render their models in a style of micrographic photorealism. These can include light microscopy, scanning electron microscopy (SEM) (simulated in Figure 03.05), phase-contrast microscopy (simulated in Figure 03.06), confocal microscopy (Figure 03.04), and transmission electron microscopy (TEM). Each of these approaches produces a signature visual texture, which is imitable in a 3D program like Maya.

3. **Non-photorealistic looks**. At the molecular level (see *Chapter 01*), the objects of biological interest are at or below the dimensions of the wavelengths of ordinary visible light; ordinary cameras don't work in this world. Here, our everyday intuitions about the nature of light and form break down. We could try to (and scientists do!) use illuminations of shorter wavelength to diffract from those small structures, as in X-ray crystallography and electron microscopy. The resulting photographs hover at the threshold where our sense of visual comprehension departs from the everyday experience. Once we reach the molecules and atoms of biological structure, we are on the doorstep of atomic physics. This is the quantum realm, where matter seems at once wave like and particle like—traits that do not have anything like a photorealistic depiction. Nevertheless, photorealism has, as we shall see in a later project, been used for depicting certain properties of atoms and molecules. These properties are both essential to biological function and well described in terms of the mathematics of NURBS surfaces. But that is just a start: as a result, animation at the cellular and molecular level is ripe for various kinds of interpretive rendering, which can draw even further on photorealistic effects, or mix them with other non-photorealistic approaches to represent the molecular fabric of living matter for maximum impact and interpretability.

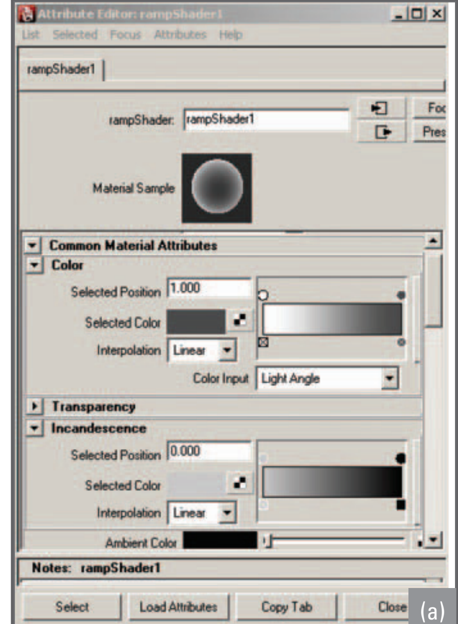


FIGURE 03.05

The appearance of objects in a SEM—with their characteristic bright edges and darkened centers—is often emulated in illustrated depictions of cells and molecules. Such a look can be created in Maya using the Ramp shader (a), as was done for the rendering of bacteria (*C. difficile*) in (b).

Scale bar ≈ 10 μm

(b) Courtesy Shaftesbury Films and AXS Biomedical Animation Studio.

Copyright Shaftesbury ReGenesis III Inc.

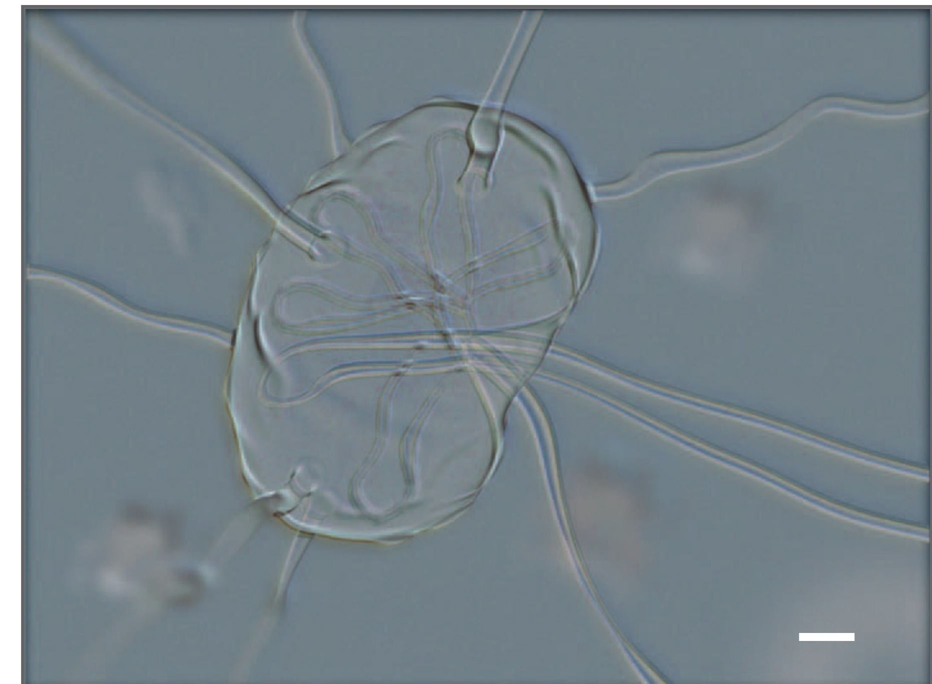


FIGURE 03.06

A Maya rendering of a lymph node, composited in Adobe After Effects to stimulate the appearance of phase-contrast microscopy.

Scale bar ≈ 2 mm

Courtesy and © 2006 Marc Dryer. Used with permission.

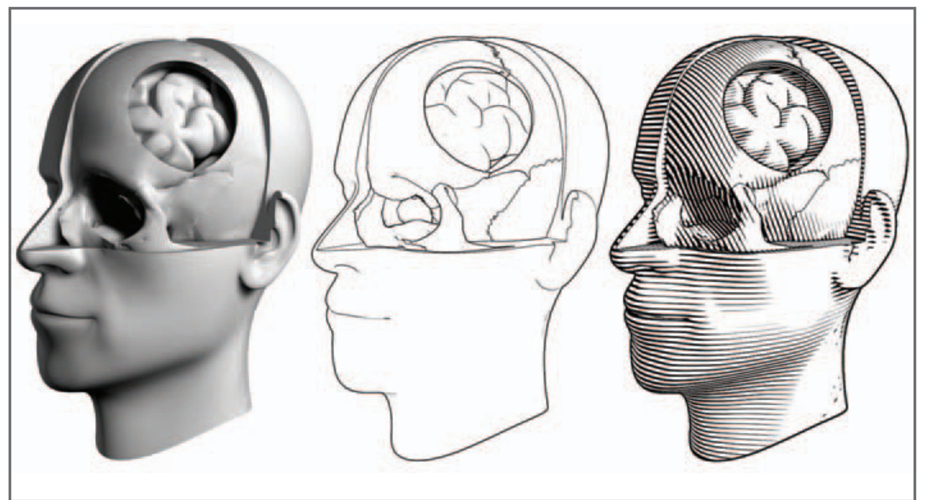


FIGURE 03.07

A 3D model rendered using left: a typical "photorealistic" adaptive scanline render; middle: a pen-and-ink style "non-photorealistic" (**NPR** for short) render; and right: an engraving style NPR render.

The term non-photorealistic rendering (often abbreviated to NPR) refers to computer-generated images that either emulate traditional artistic styles (hand-drawn, painted, engraved, etc.), or otherwise represent images in a non-photoreal way (see Figure 03.07). As noted above, research in computer graphics, for its first few decades, was consumed primarily with the goal of creating photorealistic images. Before that goal was even accomplished, many questioned why photorealism should be the default end goal of rendering systems[3,4]—after all, artists have, for millennia, made compelling and informative images without cameras (or computers), and there must be something of use in the variety of visual styles toward which they have gravitated. In response, over the last decade or so numerous graphics researchers have explored stylized depiction; the result has been a discipline named (unfortunately) for what it is not, rather than what it is.[5]

Why choose an NPR style to define the look of your animation or film? There are several possible reasons:

- A number of studies[6,7] have shown that non-photorealistic representation (especially well-constructed line drawings) is often easier for people to interpret than photographs or continuous tone images. The reason for this has not been fully elucidated, but it may have to do with the necessary simplification of line drawings, their elimination of extraneous detail, and the pre-segmented nature of the objects in a line drawing. It is worth noting that, despite the ease of acquiring photographs, line drawings are still very common in technical documentation.
- Photorealistic rendering approaches can be convincing enough that viewers of an animation might mistake what they are looking at as empirical imagery, rather than a simulation, reconstruction, or interpretation. In some cases, this misattribution of veridicality to animated sequences could be problematic. One could argue that NPR-rendered sequences will usually be understood by viewers as interpretations, and would be far less likely to be mistaken for "reality" as captured by a camera.
- NPR approaches tend to communicate the "provisional" or contingent nature of what is being represented, and therefore may be more appropriate when the

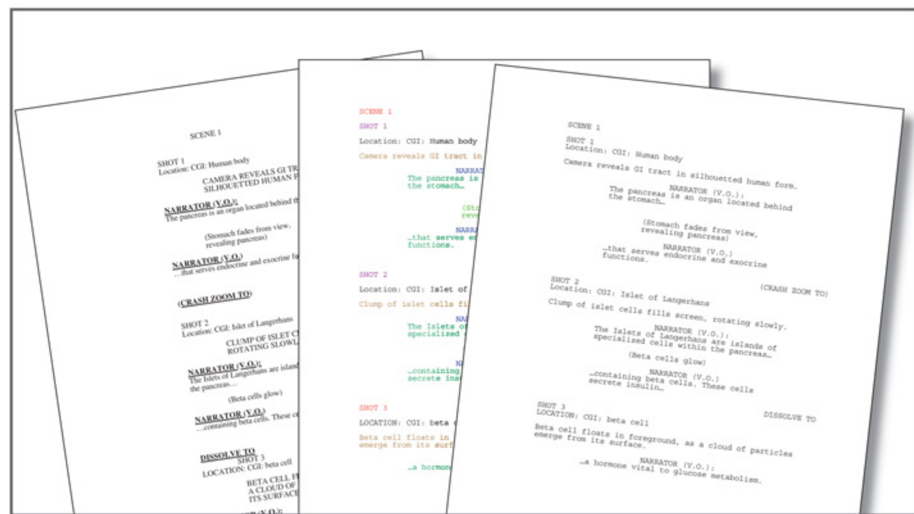


FIGURE 03.08

Examples of script formatting styles: from left to right: BBC, color-coded, and tombstone.

animation is highly simplified, or when the particular structures or processes being represented are not fully characterized.

- NPR can be used for purely aesthetic reasons. Also, digital pen-and-ink is far less messy than its real-world counterpart, and potentially easier to learn.

There has been an explosion of interest during the last few years in stylized depiction, with attendant research groups, conferences, and commercial development. Interestingly, the difficulty of deriving a good line-drawn representation of a 3D model, for instance, has emphasized how little science grasps about the psychology of picture perception. In that sense, many NPR researchers may be helping, from the algorithm upwards, to build models of human visual perception.

As you work through the book, we will guide through projects that create a distinctive look for each final animation produced by Maya and your MEL programming. You will also see many images that showcase the looks chosen for other projects. We think this will help build your experience in creating visual styles—"looks"—well suited to the needs of your future endeavors. But regardless which visual style you select, keep in mind that look of your film is no more subjective than any other facet of scientific communication; it is the result of a series of reasoned decisions about how an animation is meant to be interpreted, and is thus of crucial importance.

The treatment and the script

A film treatment is a short, narrative description of what the viewer of the proposed film would see. It is less about the "back story" of the animation and more about what the experience of watching the film would be like. Treatments are often the first step in the filmmaking process and are often used to gain initial approval and financing for a project.

A script is also a written document, but a far more detailed one which formalizes the proposed film in terms of sequences, scenes, shots, dialog or narration, sound effects, and production notes. There are several standard formats for scripts, which use special text formatting to distinguish these different elements (see Figure 03.08).

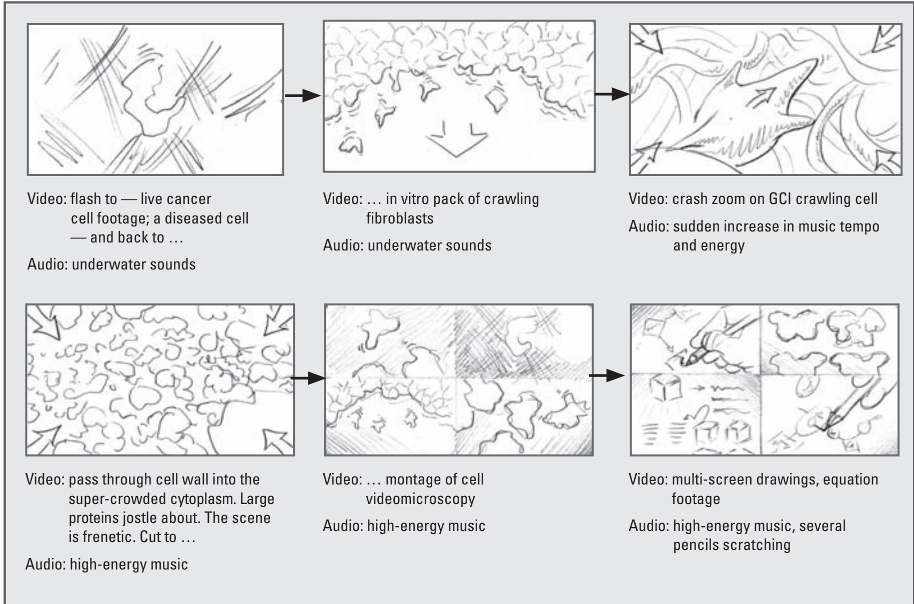


FIGURE 03.09

Plans for animation, storytelling, and science unite in this storyboard for a film about cancer cell migration. A storyboard is an essential tool in the computer animation workflow.

If the film is a live action one, the script is often the final required preproduction document created before shooting can begin. Animation, however, requires a more intensive preproduction phase, and two more elements are usually required before production starts.

The storyboard

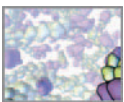
A storyboard (Figure 03.09) takes the production script and breaks it down, shot for shot, into visual form. Salient frames from each shot are rendered in thumbnail form and assembled in sequence. Narration or other scripted elements are often included below or to the side of the rendered frames. Storyboards are a presentation medium and are the focal point of the "pitch", where team members are lead through a proposed film sequence.

The storyboard images reflect what the camera would see. Graphic devices, such as arrows and superimposed rectangles, are used to indicate intended camera motion or change in focal length. The rendering approach used to make storyboard frames should not be "careful" or "finished" since a storyboard is meant to be a working document that, in a successful project, will see numerous revisions as sequences are reworked, new camera angles and movements tested, and shots added or eliminated.

In classical (i.e. Disney-style) animation preproduction, the storyboard can be the principal arena for the definition of the overall narrative, supplanting the script as the source of the story.

The 2D animatic

The 2D animatic is a more recent innovation. It usually involves the transformation of the static storyboard into a piece of motion media, complete with test soundtrack.



Storyboard frames are scanned and assembled in an editing, compositing, or motion graphics software package like Adobe After Effects. These programs allow for the animation of 2D elements over time; thus object and camera movement can be simulated and synchronized with audio. The storyboard images are sometimes separated into foreground, midground, and background elements, to better facilitate the creation of object motion and **parallax** effects. The animatic allows the film director to test story flow and timing; since the result is a rudimentary film, it is the first working version of the project. In some studios, an animatic is called a **story reel**. In the earliest years of hand-drawn animation, story reels were known as **pencil tests** since the animators' penciled drawings were put on a camera stand and photographed, one by one, as film frames to be projected for review and criticism of the animation.

Workflow stage 2: Production

In the production stage, the plan developed in the preproduction phase is implemented. Ideally, at this point the story is well defined, and no further narrative changes are anticipated. In most studio settings, animated scenes and films are the creative work of teams of artists and technical specialists who work together in the preproduction, production, or postproduction stages of the workflow. One artist would not normally undertake all of the numerous steps and stages alone. One advantage of this book is that you will gain experience in all of these principal roles and functions. We believe this will strengthen your ability to undertake, from start to finish, self-assigned projects on your own, and to function well as a member of these large, diverse teams. Animation production in Maya and similar top-tier products for 3D computer animation will require your attention to the following elements of the production workflow.

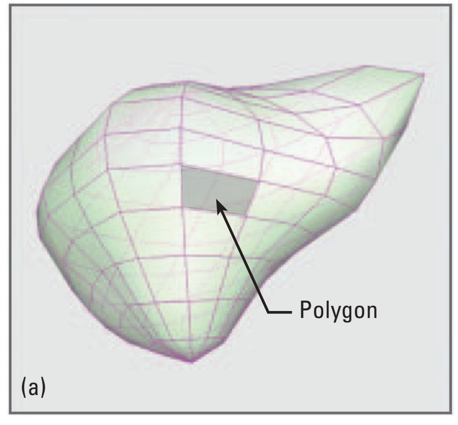
The 3D scene: Your digital stage

The term scene has two meanings for animators. Traditionally, it refers to a sequence of events that comprise a distinct element in a story. In Maya, on the other hand, a scene is the 3D environment, including models and animation, contained in one computer file. It is essentially a stage for digital action. Several Maya scenes may be developed to create a single traditional one, or one Maya scene may contain the models, action, cameras, and lights needed to create an entire story comprised of many traditional scenes. In this book we'll use the Maya definition of a scene: one 3D environment embodied in a file computer—Maya scene—file.

For a scientist, this notion of the word scene might seem a hazy or foreign concept. Perhaps the best way to think about it is as a model world, or, more specifically, a specialized apparatus for running an experiment. Just as the cell biologist might have a bank of Petri dishes, each growing a different variety of bacteria, the Maya-using researcher might have a range of scene files, comprising various projects, tests, and iterations of particular experimental approaches.

Geometry modeling

The creation of objects and environments in digital 3D space is called modeling. Objects in computer animation are typically modeled as shells with no solid, or volumetric, form to them. There are two main surface types that make up these shells. These are polygonal surfaces, comprised of many interconnected flat polygons and spline surfaces,



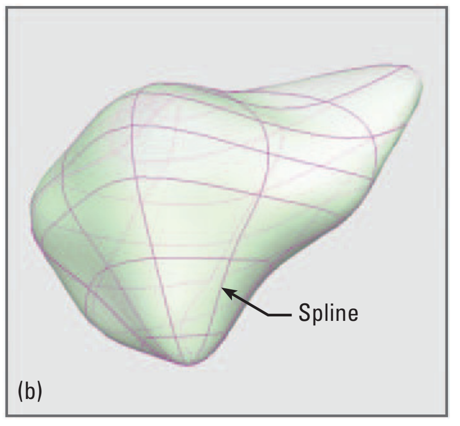


FIGURE 03.10

Common surface model types:

(a) Polygonal surface: composed of interconnected three- or four-sided polygons.

(b) NURBS surface: described by parametric curves known as **splines**.

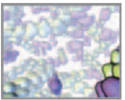
also called NURBS, that are described by mathematical curves (Figure 03.10). NURBS modeling generally produces smoother surfaces with geometries limited by the curve properties, whereas polygonal models, comprised of many small facets, can appear coarser but can be built in any conceivable shape, unencumbered by topological limitations. The choice of model type depends on the purpose and desired qualities of the finished model, but often comes down to personal preference.

Modeling and animation applications typically offer a suite of tools for model creation and manipulation in addition to a collection of primitives—ready-made models like spheres, cubes, cylinders, cones, and tori that are often the starting point for more complex geometries. Maya has a range of polygon and spline primitives as well as tools for working with both types of model. We will discuss Maya's model-making capabilities in some detail in the second part of the book. Once you understand how models are created and manipulated using the standard tools, you can tackle procedural modeling, using a computer program to automate the modeling for you and to simulate the dynamics of their interaction.

As you work on your geometry models in Maya, you may find it necessary to adjust the fidelity of the display to the source geometry (Figure 03.11). Several preset levels of detail and shading are available, such as smooth shaded, wireframe, point mode, and box. These trade off visual quality for display speed. When models and scenes become complex, it is also handy to be able to selectively hide or reduce detail on specific objects, so your workspace becomes less cluttered.

Volumetric modeling

Mention should be made of volumetric models, which use an approach very different from most commercial 3D modeling applications. Surface models, such as those discussed above, are shells possessing no inherent solidity. Volumetric models, on the other hand, are composed of arrays of cubes inhabiting 3D space called voxels (a word derived from **vo**lume pi**xels**), or densely sampled point clouds. Voxels encode for some spatially distributed variable, such as luminance, color, temperature, or density. Volumetric models can be derived from serial imaging technologies, like CT, MRI, and confocal microscopy, or produced by computational models of dynamic systems (as in modeling of storm systems). One famous example of volumetric modeling



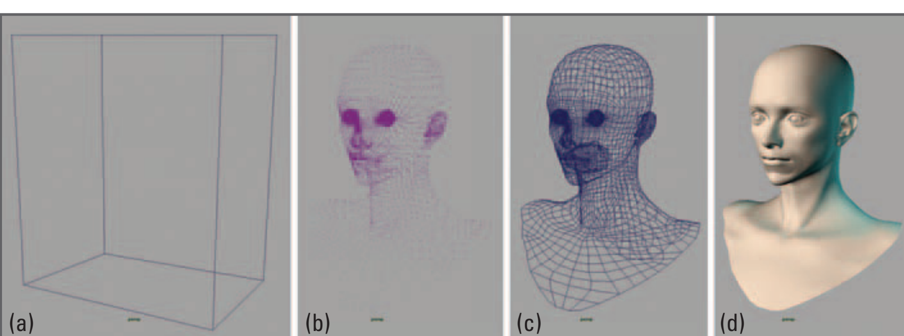


FIGURE 03.11

3D scenes can often be visualized using different display modes. Shown here from least to most computer intensive are: (a) box; (b) points; (c) wireframe; and (d) smooth shaded.

is the Visible Human Project,[8] a freely available database of sectional anatomy. In this project, sponsored by the US National Library of Medicine, researchers froze and thin-sliced male and female cadavers taking detailed photographs of the end blocks. This huge serial image database (along with calibrated MRI and CT scans done before the slicing started) can be reconstructed into a highly detailed volumetric model, with the voxels deriving their color from the pixels in the 2D images. This volumetric model is then amenable to various representational approaches (arbitrary slices, selective transparency) that allow for an unprecedented look into the human body.

While voxels are generally arranged in a rectilinear grid, point clouds can be freely arranged, with more densely packed points concentrated at points of detail or interest. Like voxels, point clouds can have color or some other property at each sample location, and can build up interestingly representative displays that are less computationally demanding than voxels.

Maya contains some volumetric tools, in the form of Maya Fluid Effects, and, to some extent, particle tools and Maya Paint Effects. But generally speaking, volumetric modeling comes at some computational cost, and the tools available for the manipulation and de novo creation of volumetric models have yet to approach the sophistication of those available for surface models. For this reason, we will concentrate primarily on surface models in this book. As tools and algorithms evolve, volumetric animation and simulation approaches will surely move into the mainstream, and therefore developments in the field bear watching.

Procedural modeling

Procedural modeling is the generation of geometry in a 3D program by algorithmic means. Many natural structures, such as plants, landscapes, and circulatory trees, exhibit qualities that are tedious to model by hand but which are amenable scripted or programmed modeling approaches. Some of these qualities include randomness, high detail, and self-similarity at a number of scales. The key benefit of this approach is that, with a small amount of input (simple equations or formulas, initial parameters), a huge amount of output can be derived (complex models of forests, coastlines, mountains, arteries, and veins).

There are numerous procedural modeling approaches, several of which are built in to Maya. You will be exploring the use of procedural modeling when you build a protein model in *Chapter 14* and a tissue matrix in *Chapter 17*.

The frame rate

It's important to note that, when using 3D software for a simulation, the work doesn't necessarily end when the simulation has run its course. The stunning imagery that makes a program like Maya so attractive to use must be rendered out. Once produced, the final images are played back at a specified rate of display, in **fps**. For example, a 30-frame animation will produce one second of motion when played back at 30 fps. The frame rate determines the quality of perceived motion and varies depending on the requirements of the viewing medium. The slower the frame rate, the less convincing the illusion becomes. Nonetheless, there are practical considerations that may warrant a slower frame rate. Rendering finished frames is a time-intensive, and therefore expensive, endeavor. A slower frame rate is therefore often a cost-saving measure used by animation studios resulting in a trade-off between quality and efficiency. In the case of animations produced for Internet viewing, a slower frame rate may be used to conform to limited data transfer rates, resulting in uninterrupted viewing but with relatively poor visual persistence.

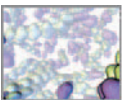
The intended rate of display should be determined *before* you begin animating items in a scene. A walking figure animated for 30 fps playback will appear in slight slow motion if projected at 24 fps. When using a 3D application for simulation, the frame rate acquires an additional meaning; it becomes the rate at which simulation events are evaluated. In the cell migration project in *Chapter 18*, we equate one Maya frame with approximately 200 seconds of "real" cell time. Therefore, when you play a rendering of the simulation at the typical NTSC 30 fps, one second of playback equates to roughly 100 minutes of cell movement! Were you to set the frame rate to 15 fps, one second of playback would represent only 50 minutes of migration. This distinction is important when presenting simulation results to your audience.

Animation

Animation, we've seen, is the art of taking otherwise static images and objects and imparting a sense of motion, and life, to them. The property being animated (called an **attribute** in Maya) could be position, scale, shape, or color among others. For example, consider depicting a deflating balloon with a sequence of still images (see Figure 03.12). As air rushes out, we make the balloon careen in all directions and gradually shrink, so it winds up looking flatter, less bright and more opaque frame by frame. To achieve this, we have animated, over time, the balloon's position (or translation in Maya), scale, shape, and its color and transparency.

Animation software like Maya generally takes two different approaches to creating these changes: **key-framed animation** and **procedural animation**. The concept of animation is inseparable from that of time. The smallest unit of time in film animation is the frame (many physically based calculations within a 3D application can rely on arbitrarily divided sub-frames for precision, however). Animators use a timeline, a linear scale divided into equal measures of seconds or frames, to locate key moments in the action. Frames containing these key moments are called **key frames**, for which values are assigned by the animator or by a script to the attribute(s) being animated.

Disney-style animation exemplifies the key frame approach taken to a highly refined stage of drawing and action timing: the term key frame comes from hand-drawn animation, where a senior, or "lead", animator would draw only those frames considered "key" to the action being represented—the most dramatically intense moments of



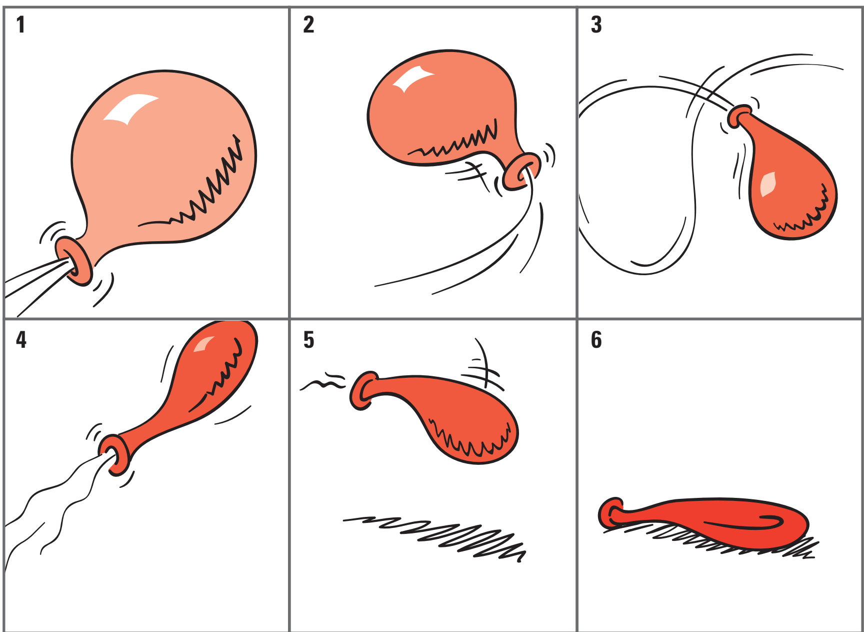


FIGURE 03.12

A deflating balloon represents the concept of animated attributes. Not only do the balloon's position and rotation change over time, but so do its size (scale), and surface appearance (opacity).

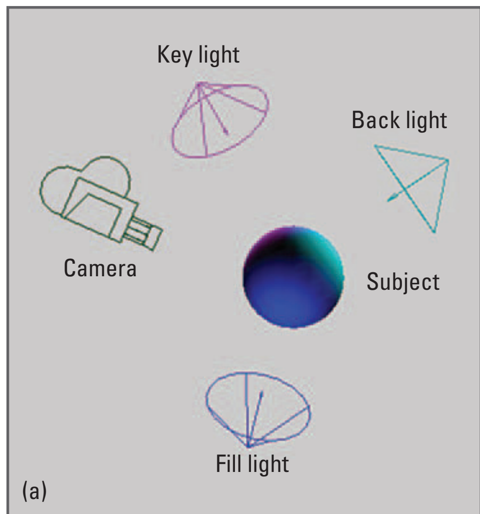
the action. Less experienced animators known as animation assistants and (below them) "in-betweeners" would fill in the frames between those key frames.

In the digital era, the computer acts as our in-betweener, interpolating object qualities from key frame to key frame. The interpolation is represented as an **animation curve**, a 2D plot of the attribute in question versus time. In the case of translation through space, the animation curve is a velocity graph. Some applications, including Maya, give animators complete control over the shape of these curves and, therefore, over the nature of the in-between action. The velocity curve, for example, can be manipulated to give a desired acceleration throughout a translation.

Procedural animation is a term for animation that is driven algorithmically, not unlike its cousin procedural modeling. It is far more similar to simulation than animation per se; the animator sets the initial parameters for the objects and then watches to see how the animation evolves over time. In a procedural animation, the animation curves are produced by your computer model. We will examine animation curves, both key framed and simulation generated, in detail later in the book.

Dynamics

Many 3D applications have dynamic simulation capabilities that utilize a built-in **physics engine**. In this case natural laws of motion can be applied to a model, which has assigned physical properties, to emulate the effects of various forces acting on it. A simple example is that of a bouncing ball. A gravitational force applied to a ball on a sloped surface makes it roll, drop off the edge, and bounce when it strikes the ground below. Attributes such as mass, elasticity, and friction are input by the user and the physics engine does the rest. A useful time saver for animators, the robust



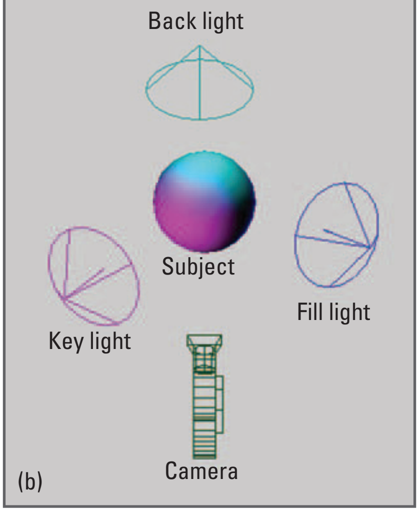


FIGURE 03.13

A standard three-point lighting rig involves a **key**, a **fill**, and a **back** light.
(a) Side view.
(b) Top view.

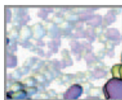
onboard dynamics capabilities of a program like Maya can prove useful in the type of predictive scientific modeling that we're interested in exploring in this book. Where the modeling requirements go beyond the capabilities of the onboard engines, Maya's programming tools let you develop the model to meet your needs.

Lights

Like a real stage, a digital stage is dark until you add lights. Most 3D applications offer a suite of available lights that mimic those found on a movie set or in a photographer's studio. These include spotlights, area lights, point lights, and infinite lights among others, but vary in actual name from application to application. For instance, a light bulb is a **point light** in Maya but an **omni light** in Cinema 4D (another robust 3D animation package developed by MAXON Computer GmbH of Germany). There is usually a default light to provide general illumination before you begin adding lights to a scene. Once lights are added, the default is switched off. You may add as many or as few lights as you wish, as you will see, and place them throughout the 3D space to best illuminate your models to be seen from a given viewpoint. Lights can be colored and assigned a number of attributes that produce special effects such as dappling or a visible beam of light.

Shadows cast by lighted objects in a scene can be a very useful device for conveying realism and for emphasizing spatial relationships within a scene. CGI shadows may have hard or soft edges (as do real-world shadows) and are typically set and adjusted within the controls for a given light.

Good lighting is an art in itself, but a beginner can achieve a reasonably good effect with the standard three-point lighting setup pictured in Figure 03.13. This configuration, often a starting point for photographers, involves a key light for primary illumination, a fill light to "fill in" the potentially harsh shadows created by the **key light**, and a **back light** to add a highlit rim to the upper edges of objects and emphasize their contours. Later on in the book you will set up and apply a three-point scheme in a protein modeling project.



Cameras

In everyday life we use cameras to view and record models and action in our 3D environment. Maya, like many other 3D animation tools, implements the same idea to help you plan how you will depict the events of your 3D virtual world. When you create a new scene in Maya, a default camera provides the view that you see. As you maneuver in the space to get a desired view of the scene, you are actually translating, rotating, and perhaps zooming the Maya virtual camera through which you're looking. 3D cameras provide orthogonal and perspective views and have many of the attributes of real cameras, such as exposure settings, lens angle, and focal length. These attributes, along with the camera's translation and rotation, can be animated and keyed for narrative purposes.

Shading

In 3D CGI, shading refers to the combined effects of lighting, surface color, surface texture, and geometry, determining the final rendered appearance of your models. When a Maya object is first created it is assigned a default shader (also called a material in Maya) with appearance attributes including color, opacity, and surface characteristics like texture or a geometric pattern. Shaders can be created and applied to objects in a number of ways to emulate real-world surfaces and, in some cases, volumes such as glass or fog. Some approaches to shading focus on non-realistic appearances, such as a pen and ink or cartoon (or toon) style (Figure 03.07).

In recent years computer graphics cards have improved to the point where, in certain cases, their output is sufficient for final quality renders. Programs like Maya have embraced this possibility with the option to render scenes using the hardware renderer. Enabling this option uses the power of the hardware in modern graphics cards (sometimes called GPUs, for Graphics Processing Units) to create the final images, often in a fraction of the time a software-based render would take.

Rendering

The production of images from a 3D scene is called rendering, a complex subject which combines the effects of lights, cameras, and shading. The images are saved as individual picture files or as a group in one movie file and can then be displayed in succession using a viewing application or passed along for postproduction work. Collectively, rendered images are often referred to as footage, borrowing from film terminology. The image format and pixel resolution of the footage are assigned in the render settings of the 3D application, having been determined by the end purpose of the animation. For example, you would usually require a different format and resolution for a small movie destined to be viewed in an Internet browser, compared to a feature film on a large screen. It is important to know the requirements prior to setting up cameras and rendering, particularly if you're creating an animation for an established format such as NTSC or an existing web page. 3D applications provide a range of standard formats and resolutions to choose from as well as custom settings.

Render engines support a number of **photorealism** effects that may be of use in developing a look for your animation projects, including:

- **Sub-surface scattering**, in which light penetrates a surface, scatters, and re-emerges (as in real-world translucent materials such as skin and wax). This can create the impression of translucent, gel-like substances.
- **Ambient occlusion**, which models the decrease of ambient light where surfaces come close together. This is a computationally inexpensive way to add a sense of real-world light interaction and solidity to an object.
- **Global illumination**, which is a computationally expensive way to model real-world illumination, where light bounces diffusely around a scene and the color of one

object can "bleed" onto another one nearby. One global illumination algorithm is **radiosity**.

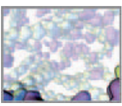
A simple approach to creating a photorealistic look of SEM in Maya, for example, is to apply a material called a Ramp shader, that is controlled by the camera direction. This technique was used to create the image in Figure 03.05b. You will meet the Ramp shader, as well as shaders for the other effects discussed here, in *Chapter 08*.

There are currently numerous rendering algorithms available to assist you in creating non-photorealistic looks for your animations. These NPR tools are available commercial options (built in to production renderers like mental ray; see *Chapter 11*) and as do-it-yourself shader techniques. Many NPR algorithms are designed to emulate traditional cel-based animation (which consisted of pen drawings on transparent acetate "cels", with flat or simply shaded color painted on the back) and are therefore referred to as toon shader (from cartoon) or cel shader techniques.

Whatever approach you choose in terms of the look of your film, you will need an efficient strategy for producing your final renders. This strategy should have two components:

1. **Compositing plan**. A computer animation scene, like the ones you see in films and on television, is rarely rendered as a single entity. Usually an individual frame is composed of layers, numbering anywhere from two to tens (or even hundreds) of separately rendered images. Sometimes the passes are composed of different image planes (e.g. foreground, midground, and background), sometimes they are of different "characters" (e.g. interacting proteins), and sometimes they are of individual image components (e.g. texture color, shadow, and highlight passes). There are very practical reasons for this: some effects are too difficult or compute intensive to render directly (e.g. depth-of-field effects) and relatively easy to add at the compositing stage; and changes are easier to make when only one component of a scene needs be re-rendered rather than the whole scene. Also, by rendering elements like lights and shadows in separate passes, they can be easily tweaked for maximum effect. While affording flexibility, rendering multiple passes can be more time consuming than rendering just one. The choice, therefore, will depend on available time, and the end use.
2. **Data management plan**. Given the huge number of render files generated by the typical animation project and the general practicality of rendering in multiple passes (which can multiply the number of render files many times), it is essential to maintain a sane data management strategy. This has a number of components:

- Project directory hierarchies are used to organize files by type. For example, Maya will, by default, save rendered image files to the Images directory within your current Maya Project folder (more on this in the next chapter).
- File naming conventions help you keep track of your work and are important for tracking file versions (e.g. myScene_001, myScene_002, and so on). Naming conventions are especially helpful on larger projects where multiple users are sharing files.
- Multiple backups of essential files: the most important files in the production phase are your Maya animation scene files and the files on which they depend (textures, ASCII data, embedded reference files); these should be redundantly backed up, preferably with an off-site option. Final renderings are also important, but in a crunch, they can be re-rendered from the scene files. In the postproduction phase,



your editing/compositing application project files are most important. Losing your footage (render) files means re-rendering lost scenes; annoying and time consuming, but not tragic. But losing your Maya scene and editing/compositing project files would mean recreating the project from scratch.

Much of our rendering work to date has been done on PCs equipped with Pentium 4 or AMD Athlon XP2700 processors and typically 1 GB of RAM.

Rendering can tax your computer system enormously, with a single frame taking from as little as a few seconds to as long as 30 minutes or more to produce. The time taken is a function of scene complexity, image resolution, available memory (**RAM**), and processor speed among other factors. Commercial animation studios typically employ an array of computers, called a **render farm**, to produce renderings more efficiently. Imagine a 90 minute animated feature created at 30 fps, with each frame taking an average of 10 minutes to render. This translates to 27,000 hours, or 1,125 days on a single computer! That's over 3 years of non-stop computing, assuming there are no errors and, therefore, a need to re-render some portion. It's easy to see why RAM and processors are at a premium when it comes to producing animated footage. Don't be discouraged, however. We routinely produce high-end rendered animations on modestly powerful desktop PCs and Macs. Throughout the book we will explore different rendering modalities that span a range of aesthetic and time-efficient possibilities.

The 3D animatic or layout

In some 3D animation production workflows, a further refinement of the 2D animatic is completed as an early stage of production. This is called the layout stage, or the 3D animatic. Draft versions of key object geometry and sets are constructed in 3D and camera movement and simple object motions are choreographed. Draft quality renderings are set to a "scratch" (draft) soundtrack. The result offers another opportunity to confirm the choices made in the earlier stages, or to refine the narrative flow further. An added benefit to the animator is the knowledge of exactly where cameras are to be placed in each scene and the economy that can be realized by only building and refining things that will be seen by those cameras. There is no use building a whole street when you are only going to shoot one side of it.

The 3D animatic is considered part of production since much of the work, especially the camera positioning and animation, will survive in the final version, even though the sets and objects are usually substantially refined or completely replaced.

Workflow stage 3: Postproduction

It is rare that an animation is in final form when rendered from a program like Maya. More often lengths of footage are produced and combined in an editing application where other elements like sound and titles are added. It is here, in the postprocessing (or just post) stage, that special effects usually are produced. In our workflow, for example, we regularly use Adobe After Effects to composite and enhance the appearance of our footage, and to add special effects, titles, narration, and music. Compositing applications like Adobe After Effects, Discreet's Combustion, and Apple's Shake are well suited to compositing and special effects work for short films and for individual shots within longer films. While also a competent compositor, an application like Apple's Final Cut Pro is more oriented to editing and is well suited to longer films. It is not uncommon to use After Effects to produce segments of effects-heavy footage and then composite all footage in an editor like Final Cut Pro to create the assembled film.

At the time this book was published, Apple Shake was available for Mac OS X and Linux systems only and Adobe After Effects supported Mac OS X and Windows systems.

Moreover, it is also common to render components of 3D animated scenes, such as the background and foreground elements, in separate passes to be composited in post. By rendering elements like lights and shadows in separate passes, they can be easily tweaked for maximum effect. If, however, they are rendered together in one pass, they can only be adjusted within the 3D application and then re-rendered. While flexible, rendering multiple passes is more time consuming than just one, and requires competent file organization and management. The choice, therefore, will depend on available time, and the end use.

Regardless of the approach taken, your final animation must be output from the editing application as a sequence of image files or as a self-contained movie file. Since there is considerably less computer processing involved at this stage, the output or "final render" from the editing or compositing stage (not to be confused with the 3D animation rendering discussed above!) takes far less time than an average render from a 3D application.

Putting it all together

Now that you have a sense of the computer animation workflow, it's time to start Maya and have a closer look at how crucial workflow steps like modeling, animation, and rendering are tackled. The next part of the book will introduce you to Maya. We'll then have you writing MEL code and rendering your own animations. Onward!

References

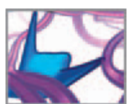
1. Kerlow IV: *The Art of 3D: Computer Animation and Effects*, 3rd ed. John Wiley, Hoboken, NJ, 2003.
2. Anderson J, Anderson B: The myth of persistence of vision revisited. *Journal of Film and Video* 45: 3–12, 1993.
3. Winkenbach G, Salesin DH: Computer-generated pen-and-ink illustration. *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*. In Computer Graphics; Annual Conference Series, 28: 91–100, 1994.
4. Saito T, Takahashi T: Comprehensible rendering of 3-D shapes. *ACM SIGGRAPH Computer Graphics Archive* 24: 197–206, 1990.
5. Gooch B, Gooch A: *Non-photorealistic Rendering*. AK Peters, Natick, 2001.
6. Ryan TA, Schwartz CB: Speed of perception as a function of mode of representation. *American Journal of Psychology* 69(60), 1956.
7. Newman RM, Bussard N, Richards CJ: Integrating interactive 3-D diagrams into hypermedia documentation. *Proceedings of the 20th Annual International Conference on Computer Documentation (SIGDOC 2002)*, ACM Press, 2002, 122–126.
8. The Visible Human Project (website): http://www.nlm.nih.gov/research/visible/visible_human.html, accessed October 14, 2007.

This page intentionally left blank

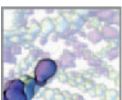
Part 2

A foundation in Maya

This page intentionally left blank



04 Maya basics



We wanted this book to be self-contained without just repeating what others have communicated so well. The material in this part of this book is therefore intentionally brief. There are many excellent resources available for learning Maya, including the Help library that comes with the software. Also, we encourage you to take advantage of the resources listed in the *Further reading* section under the heading, *Learning Maya*.

Getting started

This chapter will get you started in Maya. We'll begin with a quick description of the different Maya packages, followed by where to look for help and a listing of system requirements, and then a description of Maya scene files and projects. Next we provide a brief discussion of how Maya works behind the scenes. With these basics out of the way, you'll be ready to explore the user interface (**UI**), which we introduce throughout the rest of the chapter.

Maya Complete

Maya Complete is the name given to the software package which contains the basic modeling, animation, dynamics, and rendering functionality. It is also programmable via the Maya Embedded Language (**MEL**) and Python scripting interface and the C++ developer application programming interface (**API**).

Maya Unlimited

Maya Unlimited includes all of the Maya Complete functionality, with the addition of software modules that enhance specific areas of the computer animation workflow. With the release of Maya 2008, the following modules were included: hair; fluid; fur; live; and nCloth. Descriptions of these are available on Autodesk's website and within Maya Help.

Maya Personal Learning Edition

Autodesk, the makers of Maya, provide a limited version of the software free-of-charge for non-commercial use. Maya Personal Learning Edition (**PLE**) can be downloaded from the Autodesk website at the following URL (current February 2008):

http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=7639525

With this version of Maya you can be up and running in a matter of minutes. For a list of limitations and restrictions to Maya PLE, follow the *Questions & Answers* link on the above Web page. Among these limitations, the following should be noted with regards to the material presented in this book:

- Maya PLE uses a different file type (.mp) than the commercial version of Maya (.ma and .mb).
- Maya PLE files cannot be opened in the commercial version of Maya.
- Images rendered from Maya PLE bear a watermark.
- Vector image formats cannot be rendered from Maya PLE.
- Certain MEL commands cannot be run in Maya PLE. Those worth noting here are fopen and fwrite.
- Certain file translators (for importing and exporting non-Maya file formats) are not supported in Maya PLE.

System requirements

Check the documentation accompanying the version of Maya you're using for system requirements. Autodesk provides information on their website regarding Maya-qualified hardware:

http://www.autodesk.com/qual-charts

The mouse

We recommend using a 3-button mouse and have written instructions throughout this book accordingly. In the text, we use the following abbreviations:

LMB = left mouse button

MMB = middle mouse button

RMB = right mouse button

The terms **click** and **Double-click** apply to the **LMB**.

Monitors

If your budget and desk space allow, dual monitors are a good idea. Editing windows can be placed on one screen, allowing you to maximize the view of your scene in the main window on the other.

Consult the Qualified Hardware lists on the Autodesk Maya support website if you are considering using Maya with a small display. Some laptop and tablet displays are too small (smaller than 1280 × 1024) for Maya and limit its usability.

Help and instructions

As you learn to use the software, Maya's Help Library (Figure 04.01) will be your single greatest resource. It provides information, and often examples, for most of the program's features and functions. Frequently in the text, we will set down a Maya Help reference where you can pick up more information on the current topic, with arrow characters (→) separating links or titles as follows:

Maya Help → Using Maya → Tools, Menus, and Nodes → Main Window

To launch Maya Help:

Choose Help → Maya Help

or

press F1

You may also find the **Popup Help** feature useful as you feel your way around the UI; it is enabled by default the first time you start Maya and will display a short description of each tool and button that you pause over with your mouse.

To enable or disable ***Popup Tooltips***, choose ***Help*** → ***Popup Help***. Popup Tooltips work only on the active window. You make a window (or editor) active by clicking on it.

Through this and the subsequent chapters, we have written step-by-step instructions as follows:

1. **Choose Create → Polygon Primitives → Cube ❐.**

Figure 04.02 shows the menu selection corresponding to the above instruction. The result is the creation of a polygonal cube model.

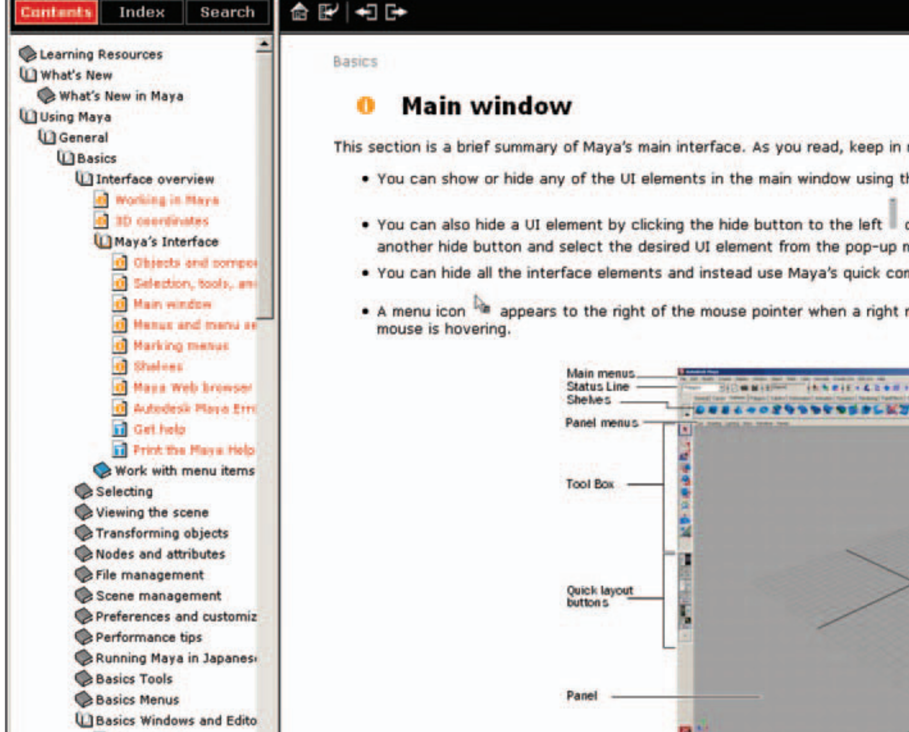


FIGURE 04.01

Maya Help can be accessed using the F1 hotkey. References throughout the text of this book indicate the links to click on in the Contents section. Keyword searches return links in two categories: Information and Tutorial.

Release notes

You may encounter strange limitations in Maya's performance, including UI oddities that are peculiar to a specific operating system. The Maya developers at Autodesk often are aware of these problems before the software package is published and include a list of known limitations along with suggested solutions (or work-arounds):

Release Notes

Maya Help → Using Maya → General → Release Notes

Hotkeys

A **hotkey**, also known as a **keyboard shortcut**, performs a task with a single keystroke or combination of keystrokes. This saves the time otherwise spent locating an item with your mouse pointer. Useful hotkeys will be mentioned where appropriate. You can create a custom hotkey for just about any operation in Maya, including the execution of MEL commands and whole scripts.

In Windows, Linux, and IRIX, key combinations involve either the Ctrl or Alt key. The Mac OS equivalents are the Command (or Apple) key and Option key, respectively. The Shift key is common to all systems. For efficiency, we use the Windows key notation throughout this book.

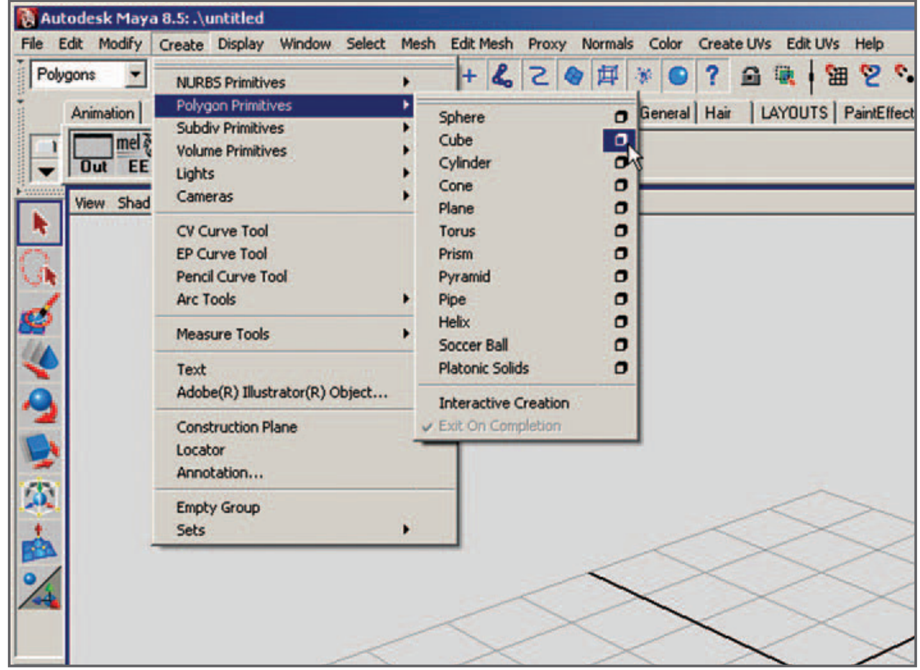


FIGURE 04.02

Menu selections in Maya are written as follows in the text:

Choose Create → Polygon Primitives → Cube ❐.

Hotkeys

Maya Help → Using Maya → General → Basics → Basic Tools → Hotkeys

User profiles

Maya, when installed, automatically creates a profile for each user account on the computer. Figure 04.03 shows the directories and several of the files that make up a user profile, where Maya, by default, stores and retrieves files, including user preferences. The advantage to this setup is that users don't require system administrator's access to use Maya and organize their files, and each user can have their own settings for Maya.

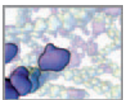
Start Maya

To start Maya, do one of the following:

Double-click the Maya desktop icon or the Maya application icon in your program (or applications) directory.

or **Type maya at a command prompt.**

or **In Windows, choose: Start → All Programs → Autodesk → Maya (version #) → Maya (Complete, Unlimited, or PLE, and version #).**



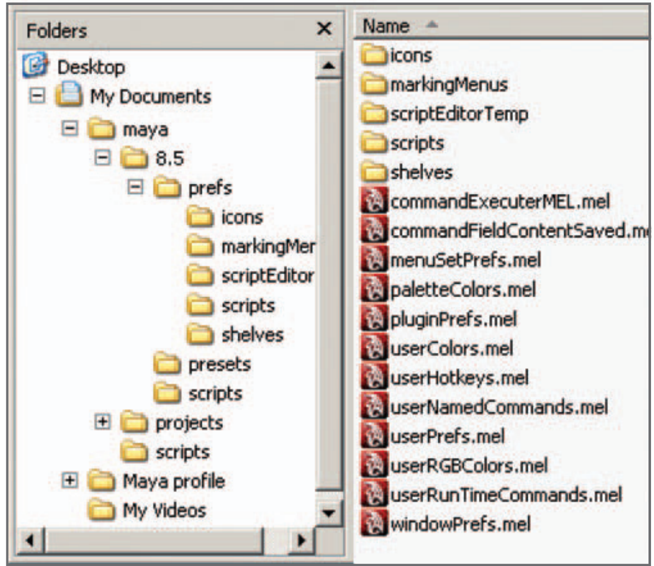


FIGURE 04.03

A user profile consists of directories and files specific to Maya for a given user.

The scene file

In Maya, a scene is the 3D environment, including models, animation, lights, and cameras, contained in one file. When you open a file in Maya, you open a scene. Scene files are of two types:

We recommend saving frequent incremental backup versions of a scene file as you work on it. If the current version becomes corrupted or you made a change to the scene that you wish to undo but can't, you can simply open the previous version of the file and continue working.

1. **Maya ASCII**, denoted by the file extension **.ma** as in *myNewScene.ma*. A Maya ASCII file, which is written in MEL script, can be opened and edited in a text editing application. This comes in handy if a Scene file becomes corrupted and will not open in Maya; it is often possible to track down and delete or correct the offending bit of code in a text editor, then resave the file and attempt to open it again in Maya.
2. **Maya Binary**, denoted by the extension, **.mb**, as in *myNewScene.mb*. Binary files are written in computer machine code and therefore cannot be easily edited in a text application, the way ASCII files can. However, .mb files are generally smaller than .ma files, so they take up less storage space on your hard drive, and take less time to open, save, and render than .ma files do. We usually work with Maya binary files but save backups in Maya ASCII format because of the security the latter provides against corrupt files.

Figure 04.04 shows excerpts from .ma and .mb versions of the same file.

The Maya project

Maya saves scene and related files in a *projects* directory on your hard disc. It is the directory that Maya defaults to when saving or retrieving files. Multiple projects directories can exist, but Maya will refer only to the one that has been specified, either by default the first time you launch Maya, or by you as described below. Within the projects directory, you have the option to create subdirectories to organize different file types that may be associated with your main Scene file. To create a new Project:

1. **Choose File → Project → New. This will open the Project window.**
2. **Enter a name for your Project in the Name field.**

(a)

```
//Maya ASCII 8.5 scene
//Name: mayaCodeSample.ma
//Last modified: Sat, Nov 10, 2007 12:
//Codeset: 1252
requires maya "8.5";
currentUnit -l centimeter -a degree -t
fileInfo "application" "maya";
fileInfo "product" "Maya Unlimited 8.5
fileInfo "version" "8.5";
fileInfo "cutIdentifier" "200612162224-
fileInfo "osv" "Microsoft Windows XP S
createNode transform -s -n "persp";
    setAttr ".v" no;
    setAttr ".t" -type "double3" 239 179
```

(b)

FOR4[•šÈMayaFOR4[][¤HEADVERS[][
ef[][MADE[][]undef[][CHNG[][]Wed,[A
:11:00[PM[][ICON[][]undef[][INFO[][]u
undef[][INCL[][]undef([]LUNI[][]cm[]T
[][]deg[FINF[][]application[maya[][]FI
t[Maya[Unlimited[7.0[]FINF[][version
utIdentifi er[200507192211-654274[]
icrosoft[Windows[XP[Service[Pack[2|
n[]PLUG[][]Mayatomr[7.0.1.14m[-[3.4
4[][`XFRMCREA[][]persp[]DBL3[][]t[]¿
_Mv1èB¿âß²l–…[]DBL3[][]r[]¿ã[Ì[è[@[
è[][DBL3[][]rp[]<Ð[][][]<[][][][<À[][][]D
†½¤ò{½[;,îó=½)fQ'ëö[][FOR4[][HDCA
spShape[persp[][FLGS[][]v[][DBLE[][]r

(c)

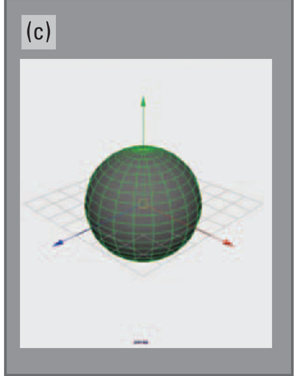


FIGURE 04.04

Excerpt from a Maya scene viewed in a text editor:

(a) Maya ASCII format,

(b) Maya Binary format, and

(c) as seen in Maya.

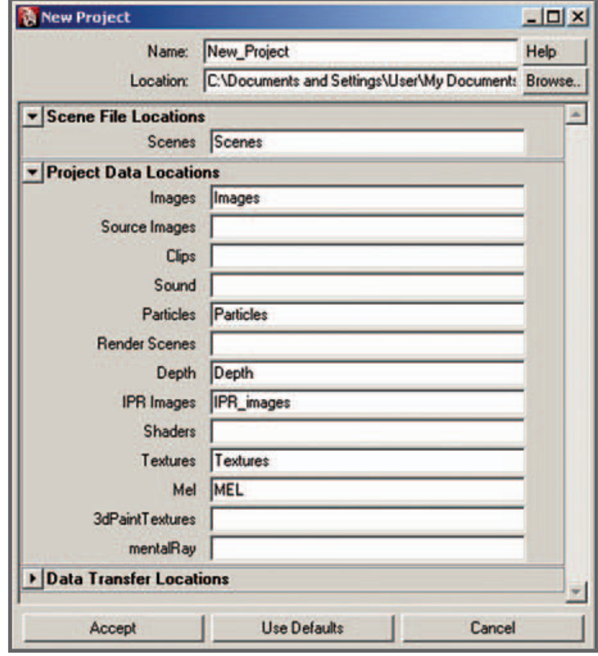


FIGURE 04.05

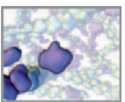
The New Project window. The names in the text fields will be used to create the directories in which Maya writes and looks up files.

3. **Click Use Defaults to create default subdirectories**

or **Enter names only in the fields for which you want subdirectories created.**

For the projects and tutorials in this book, the configuration shown in Figure 04.05 **will suffice. You won't require directories for the fields that have been left blank because you won't generate or read in the types of data typically stored in those fields—at least with respect to the exercises in this book.**

The new project is created in the default Maya projects directory unless you navigate to a preferred location using the Browse button before hitting Accept. By creating a



project, you have already begun exploring Maya's UI. The next section puts the UI in the context of how Maya works behind the scenes.

How Maya works (briefly)

Though deep, Maya is designed to be remarkably transparent; its inner workings are exposed for those who wish to explore beyond the basic UI tools. Our discussion of the program architecture will lay the foundation for such an exploration and help you understand what's actually happening—with the models, cameras, lights, animation, and so on—when you start pressing Maya's buttons. While much of what follows here may seem rather abstract to the Maya beginner, this is nonetheless the place to discuss it since it concerns the foundation on which everything you will do in Maya is based. You may wish to skim this section at first, then come back to it after covering the rest of *Part 02*.

The Maya program architecture

The underlying architecture of Maya is what sets it apart from other high-end 3D computer graphics applications. It is arguably infinitely flexible and expandable, which is a primary reason for choosing Maya as a platform for in silico biology. However, it is possible to use Maya extensively without ever being aware of what's going on beneath the surface. This is a testament to Maya's ease of use. Nonetheless, a basic knowledge of the underlying structure will make your experience with Maya more meaningful, and can pave the way to more advanced work with the program, including the development of custom tools called **plug-ins.**

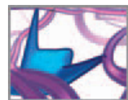
The Dependency Graph and DG nodes

In Maya, scene elements are represented by nodes connected to one another. The complete network of nodes and their connections is called the **Dependency Graph** (**DG**, for short), and the nodes themselves, **DG nodes**. The term *dependency* refers to the interdependency of elements in the Maya scene. For example, the location of a moving cube *depends* on an input connection from an animation node which calculates its position. The animation node in turn depends on a Time node in order to calculate its value(s).

The DG essentially *is* the Maya scene. Most users interact with the DG using the windows, menus, and tools of the UI. Figure 04.06 is a schematic illustration of the DG and UI working together. When you perform an action through the UI, it is relayed as a MEL command to the DG, where either a DG node is created and/or connected to another.

A DG node stores, sends, and receives information (or data) about an item in a Maya scene. Data is stored in **attributes** which can be connected to the attributes of other nodes in order to send or receive information. There are many types of attributes, each storing specific information, such as the color of an object or the brightness of a light. In many cases, a node performs calculations on the data it receives (through input attributes) to produce the data it sends (through output attributes). Figure 04.07 is a schematic representation of a simple DG node. Maya allows great flexibility for viewing and interacting with the DG nodes and their attributes. Figure 04.08 shows three UI windows with different views of the same node and its attributes.

From the software engineer's point of view, working in Maya might be said to boil down to creating nodes, then setting and interconnecting their attributes. For example, Figure 04.09 shows the DG nodes that are created and connected when you



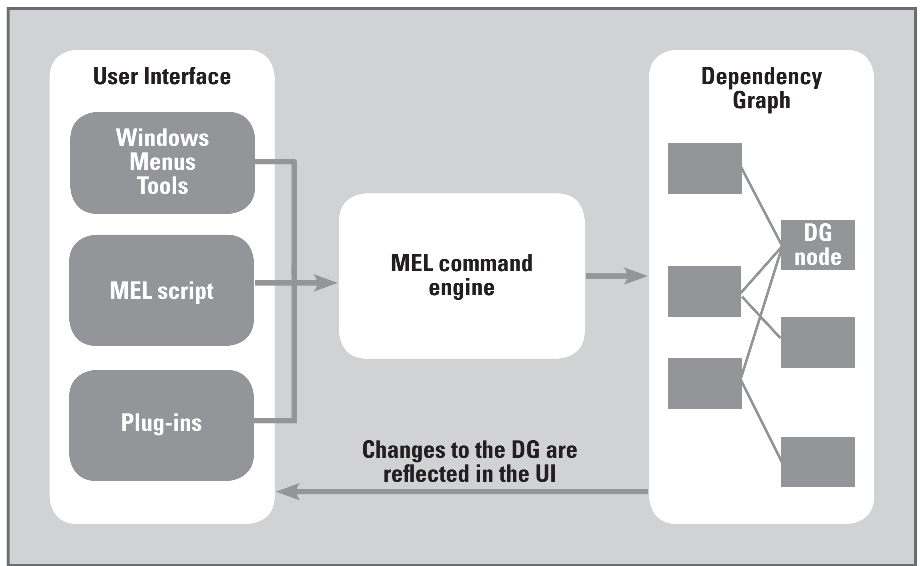


FIGURE 04.06

The Maya UI translates user input through MEL commands to the DG. Updates to the DG are in turn reflected in the UI.

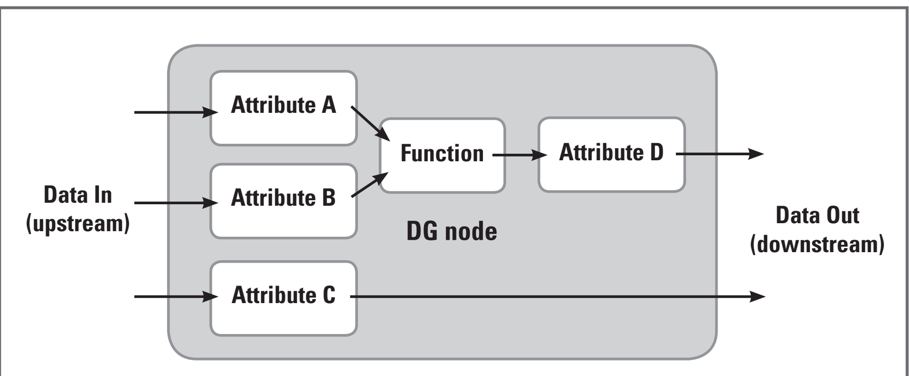


FIGURE 04.07

Schematic representation of a simple DG node. Attributes store data. Functions calculate new data. Data coming into a node is commonly referred to as "upstream". Data leaving is "downstream".

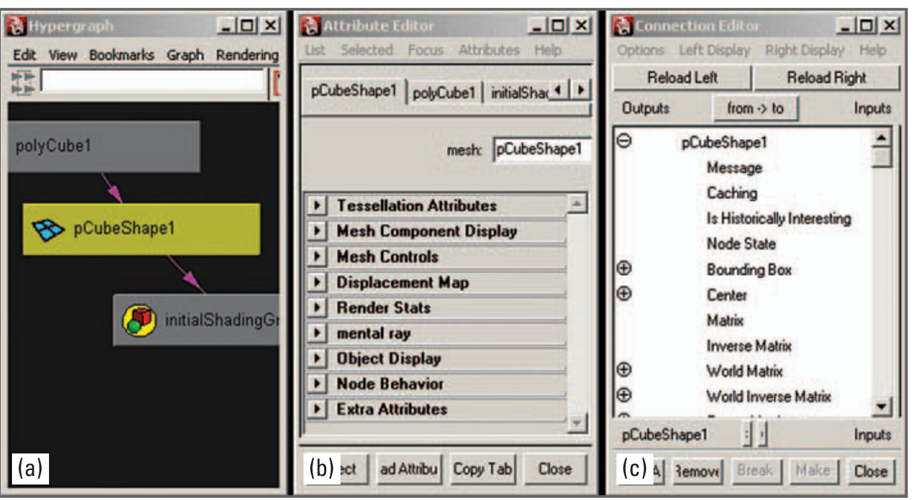


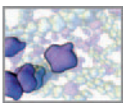
FIGURE 04.08

One node, as represented by:

(a) the Hypergraph

(b) the Attribute Editor

(c) the Connection Editor.



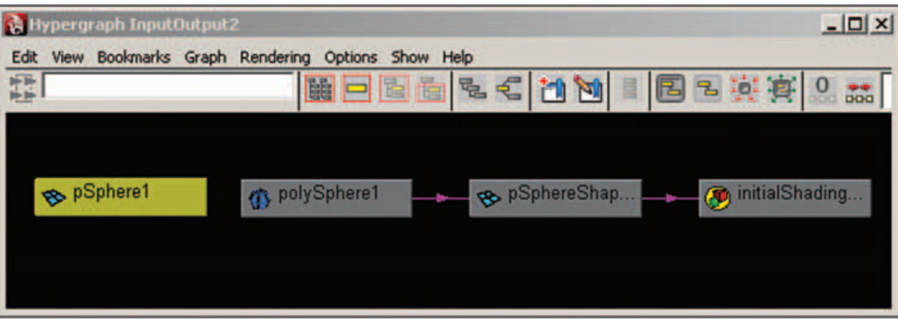
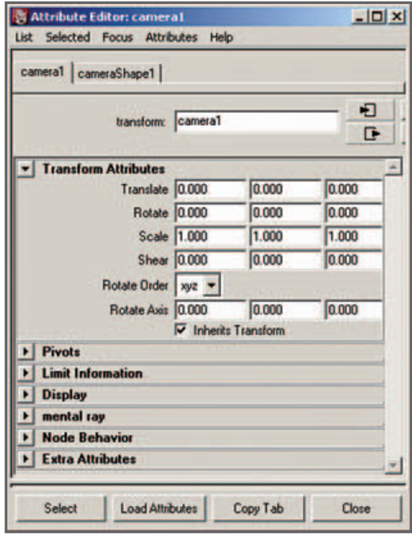


FIGURE 04.09

The DG nodes that represent a polygon sphere are displayed in the Hypergraph as boxes. When selected, a node turns yellow, as is the case for pSphere1.



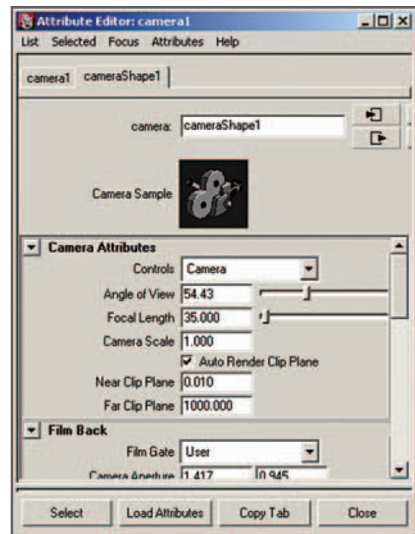


FIGURE 04.10

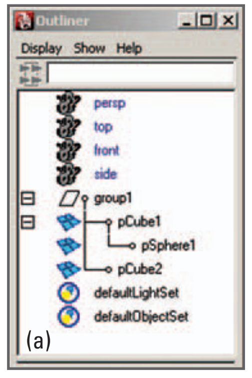
The Transform node (left) and Shape node (right) of a camera, as displayed in the Attribute Editor.

make a sphere model in Maya. Most items in a scene, including the sphere, are represented by at least two nodes, called the **transform** and **shape** nodes. There is only one type of transform node, which is used by all objects, cameras, and lights in a scene. Transform node attributes store the position (or translation), rotation, scale, and visibility of an item. In contrast, there are many types of shape node, each of which stores attributes peculiar to the entity it represents. Figure 04.10 shows the transform and shape nodes of a camera, as represented in the Attribute Editor, a tool you will use a lot for working with nodes in Maya. Note the translation, rotation, scale, and visibility attributes of the transform node. This shape node contains attributes unique to a camera, such as Focal Length and Camera Aperture.

The Attribute Editor is one of many editors that allow interaction with the elements of a Maya scene. You'll see more of it in the next section.

Construction history

When combined, any number of nodes used to create an object—the sphere from Figure 04.09, for example—make up that object's construction history. As long as this network of history nodes remains intact, their attributes can be edited to change some feature of the sphere. The node called polySphere1 contains creation attributes—those that you set when you made the sphere. As long as polySphere1



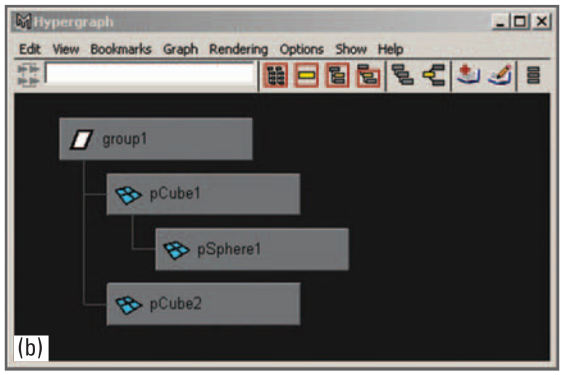


FIGURE 04.11

Scene hierarchy (parent/child and group relationships) is displayed graphically in both the

(a) Outliner, and

(b) as the DAG in the Hypergraph.

remains connected to the shape node, pSphereShape1, you can change the radius of the sphere, for example. As soon as that connection is broken, by deleting history for the sphere, you can no longer edit the radius. Construction history is a powerful feature of Maya's architecture: you can alter any node in the history network, and the object updates automatically.

As you work in Maya, you will become familiar with nodes and how to set and connect their attributes. You can work with nodes through the standard UI tools, through MEL scripting, or directly in the Hypergraph. We will explore the DG a little further in *Chapter 05*, using a graphical representation called the **Hypergraph** (Figure 04.09).

The DG

Maya Help → Using Maya → General → Basics → Nodes and attributes → Dependency Graph

Scene hierarchy and the DAG

In addition to the DG, which maps the interdependency of nodes and their attributes, Maya uses a system of connections among transform nodes called the **scene hierarchy**. There are two types of relationship in the hierarchy. These are parent/child and group relationships, and they affect how items in your scene relate spatially to one another. An example of these hierarchical relationships is shown in Figure 04.11. pSphere1 (short for polygon sphere 1) is a child of pCube1, and pCube1 is the parent of pSphere1. Both pCube1 and pCube2 are "grouped" together under group1. It can also be said that the cubes are children of group1. The scene hierarchy is represented graphically by the **Directed Acyclic Graph** (**DAG**), which is viewed through the Hypergraph (Figure 04.11).

In 3D computer graphics, the term ***parenting*** refers to making one item the child of another.

When a parent is transformed (translated, rotated, or scaled), so are its children. A child may also be transformed relative to the **transform** of its parent. These features of parent/child relationships are used extensively in computer animation, enabling the build up of complex relative motion. One everyday example of parent/child hierarchy concerns a wristwatch. The hands, which are children of the watch move relative to it. The watch in turn is a child of the wrist of its wearer, which is a child of the wearer's body. As the wearer walks down the street, the motion of the hands relative to the ground is considerably complex—a formidable kinematics problem—but

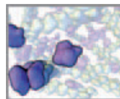


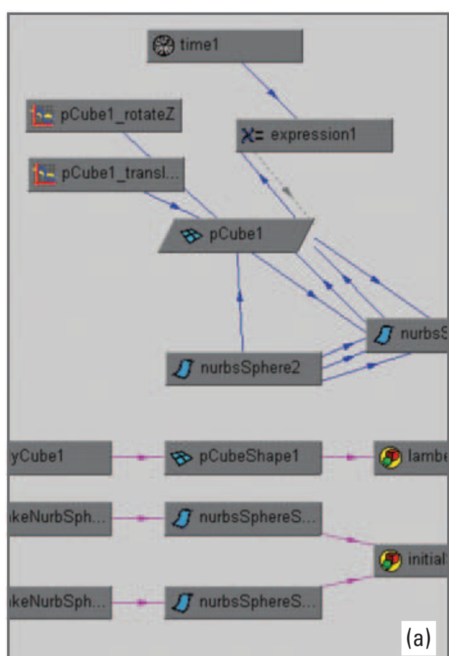
FIGURE 04.12

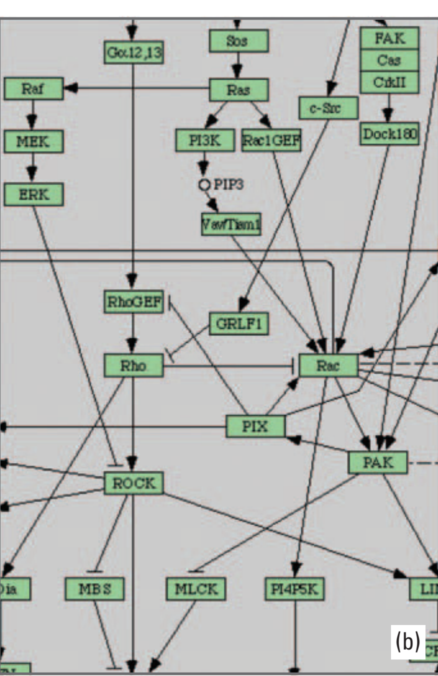
Molecular interaction and reaction pathway diagrams share visual and conceptual similarities with the underlying organization of a scene in Maya.

(a) A scene represented in the Hypergraph.

(b) Molecular interaction pathways involved in the regulation of the actin cytoskeleton. Detail from "Regulation of actin cytoskeleton" KEGG Pathway Database: http://www.genome.jp/kegg/pathway/hsa/hsa04810.html. Accessed January 20, 2008. Kanehisa Laboratories in the Bioinformatics Center of Kyoto University and the Human Genome Center of the University of Tokyo. Used with permission.

© Copyright and courtesy Bioinformatics Center, Institute for Chemical Research, Kyoto University.





this motion was brought about by simple hierarchical relationships. We will exploit parenting in the upcoming tutorials and projects.

Scene hierarchy

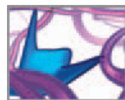
Maya Help → Using Maya → General → Basics → Nodes and attributes → Scene hierarchy

The DG, DAG, and biology

It has not escaped our notice that the DG bears striking similarities to biochemical reaction pathway diagrams familiar to biochemists, both visibly and conceptually (Figure 04.12). In the latter, nodes represent specific molecules and the connecting lines, potential reactions between molecules. Similarly, processes in cell development and interaction, can be mapped as interconnected nodes. Parallels can also be drawn between Maya's transform hierarchy and hierarchical organization in biology. For example, the compartmentalization of functional groups of molecules within cells and the arrangement of cells into tissues bears resemblance to the parent/child and group relationships in a Maya scene (Figure 04.11). It is with these ideas in mind that we are exploring the application of Maya's programmable architecture to problems in cell biology.

Maya's UI

A logical way to start exploring Maya is with a tour of the graphical UI. Its main elements are labeled in Figure 04.13. In addition to a view of the 3D digital stage, which Maya calls the workspace, you have access to many menus, tools, and controls. The items that are labeled will be introduced below, while others will be described only as



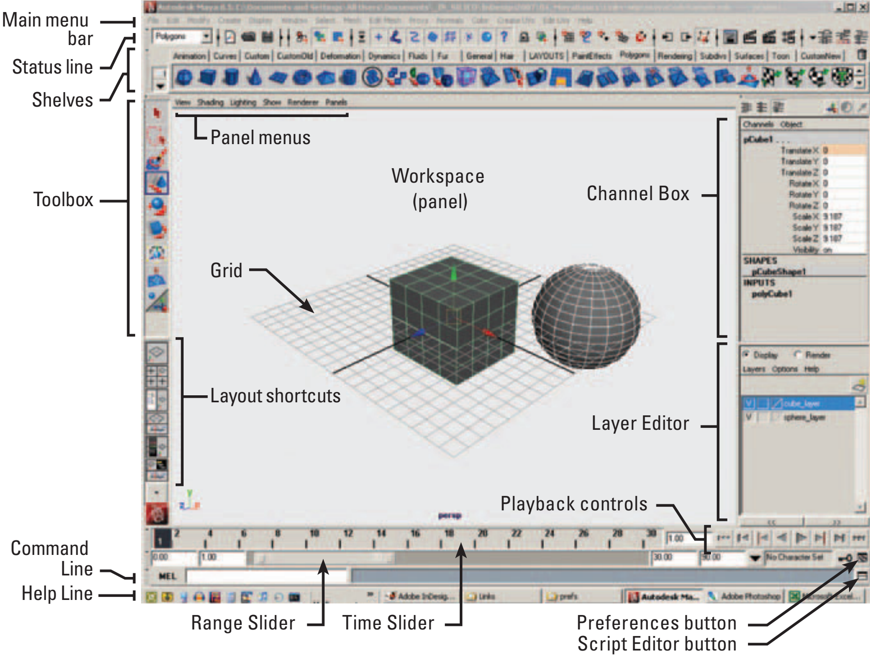


FIGURE 04.13 The Maya UI.

needed as they arise in the tutorials and projects. Furthermore, the UI may be customized in a number of ways to suit specific requirements. UI customization is discussed briefly in the Hotbox, Toolbox, and Preferences sections.

If a UI Element in Figure 04.13 was not visible after you started Maya, it may have been hidden the last time Maya was used. You can hide and display an element in the following way:

1. **In the Main menu bar, choose Display → UI Elements.**
2. **Select the item you wish to hide or display. A check mark appears next to ones that are already displayed.**

Alternately, you can hide or show all UI Elements or restore the default UI display by selecting Hide UI Elements, Show UI Elements, or Restore UI Elements, respectively.

Title bar

The file name and the path as well as the name of the currently selected item are displayed in this space.

Main menu bar

As you work in Maya, you will encounter many **tools** and **editors**. Tools let you transform and otherwise manipulate items in a scene, while editors give you access to tool settings, item attributes, and software functionality. All of Maya's tools and editors can be accessed through the UI's pull-down menus. The icons arranged around the UI simply offer quicker access to many of the tools and actions represented in the menus. There

are four main menu sets (more if you purchased extra modules or the Unlimited version of Maya) which are displayed, one at a time, along the Main Menu bar at the top of the interface. They are Animation, Polygons, Surfaces, Dynamics, and Rendering, and each corresponds to a specific software module, or more generally, to a collection of like tasks.

There are seven pull-down menus (File, Edit, Modify, Create, Display, Window, and Help) that are common to all menu sets, while the remaining pull-down menus contain items specific to each set. To switch between menu sets, say from Animation to Modeling, use the menu at the far left of the Status Line.

Alternately you can use a hotkey to bring up each set:

F2 for Animation

F3 for Polygons

F4 for Surfaces

F5 for Dynamics

(The Rendering menu set does not have a default hotkey)

Hotkey Editor

Maya Help → Using Maya → General → Basics → Basic Windows and Editors → Hotkey Editor

Each time you launch Maya, it loads all of the licensed software modules. Each module takes up memory (RAM) on your computer. You can disable any of these modules (e.g. Dynamics, Hair, or Fur) so that it doesn't load automatically on Maya startup, and therefore saves memory. To disable a software module:

1. **Choose Window → Settings/Preferences → Preferences.**
2. **Under Categories → Modules → Load on Startup, uncheck the box next to the module you wish to disable. To enable a module, check its box.**

Working with menus

Menus are critical to how tools and actions are accessed in Maya. The following descriptions will help you take advantage of built features that enhance menu usability.

Tear Off menus

Many individual menus in Maya can be "torn off" so that they remain open, with their contents easily accessible, and can be positioned anywhere on the screen. The Tear Off option is indicated at the top of the menu by a double line in Windows and a dotted line in Mac OS.

Marking menus

These are customizable menus that can be accessed from anywhere in the interface. In addition to accessing Maya's built-in tools and actions, you can tailor the Marking menus to execute custom scripts. This can be useful in a modeling application in which you use a script to execute a series of repetitive tasks. With a custom Marking menu you could, for example, execute the script easily from wherever your mouse pointer happened to be.

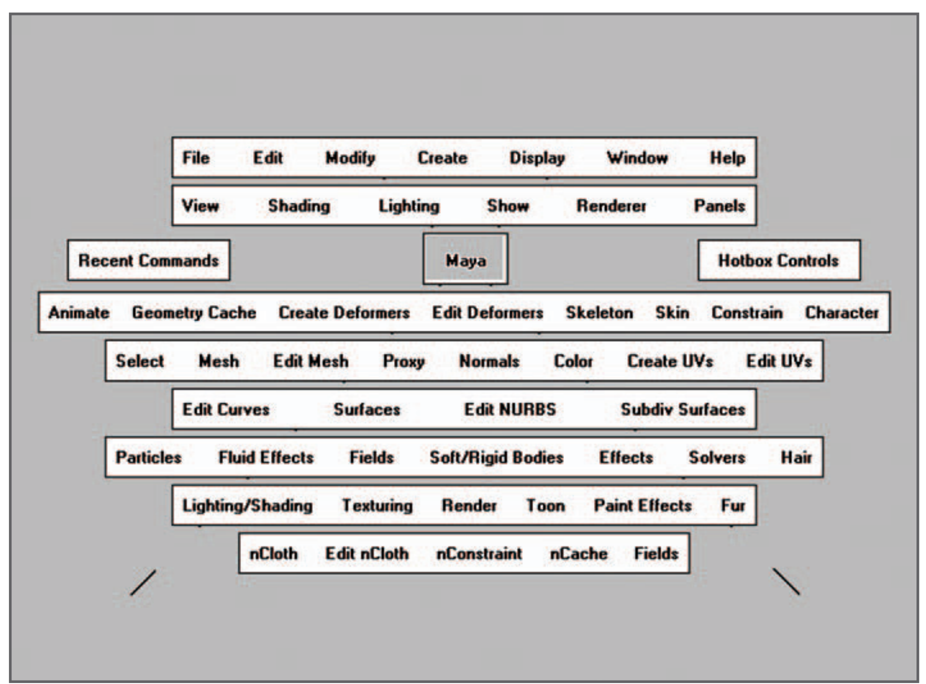


FIGURE 04.14
The Hotbox provides access to all menus available through the UI. It is activated by pressing the space bar and appears wherever the cursor is.

Marking menus

Maya Help → Using Maya → General → Basics → Interface overview → Maya's Interface → Marking menus

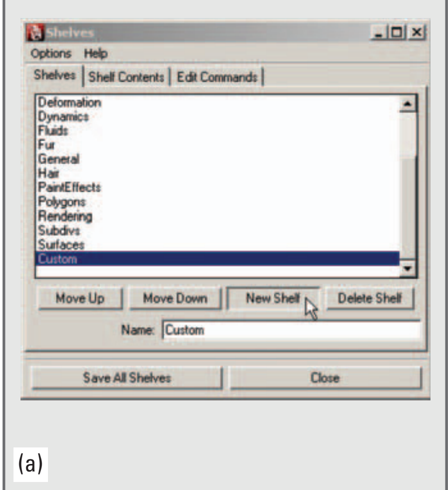
Hotbox

The Hotbox (Figure 04.14) can display all of Maya's menus at once, giving you access to every tool and editor. To access it, press and hold the space bar. It will pop up wherever your mouse cursor happens to be. You can load the Hotbox with as many or as few menus, standard and custom, as you like. When you're comfortable with it, you can hide most of the standard UI Elements (such as tool and menu bars) to reduce screen clutter and increase your visible workspace, using only the Hotbox to access the menus. To hide or show UI Elements:

1. **Choose Display → UI Elements.**
2. **Elements that are currently displayed have check marks beside them. Choosing an element changes its status from displayed to hidden or vice versa.**

To customize the Hotbox contents:

1. **Press and hold the space bar to bring up the Hotbox.**
2. **Move your mouse pointer over Hotbox Controls.**
3. **Choose from the many display/hide options.**



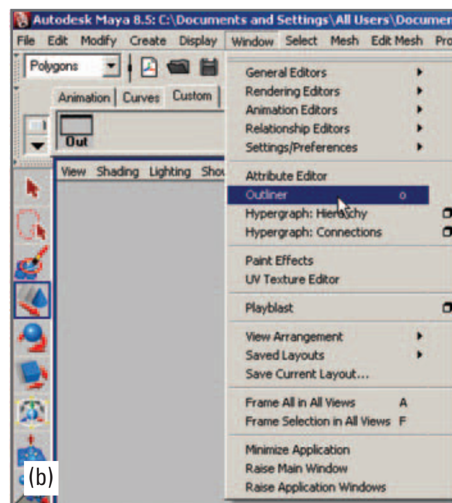


FIGURE 04.15
(a) Creating a custom shelf and (b) adding an Outliner button to it.

To customize the Hotbox

Maya Help → Using Maya → General → Basics → Preferences and customization → Customize Marking menus and the Hotbox → Customize the Hotbox

Option boxes

Many menu items are followed by the option box symbol, ❐. When chosen, this launches a window containing options for the selected tool or command. When it is not chosen, Maya applies the options that were most recently set—often the default values.

Status Line

The Status Line provides easy access to a variety of controls for the interface, for managing your scene, and for modeling and rendering. These items, as with most in the UI, can also be accessed through the pull-down menus and the Hotbox. In the Status Line, they are arranged in groups that can be collapsed and expanded by clicking on the vertical bars. At the far left of the Status Line is the menu with which you choose a software module: Animation, Dynamics, Polygons, Surfaces, or Rendering.

Shelves

Shelves are a means of organizing tools and actions for quick access. Maya comes with a number of prestocked shelves to which you can add more items. For example, the Surfaces shelf displays buttons used to create each of the NURBS geometry primitives—sphere, cube cylinder, cone, plane, and torus—along with tools to edit and manipulate these surfaces. You can also create your own custom shelves for easy access to menus, palettes, tools, and actions that you use frequently. In the following example you will create a custom shelf to which you'll add the Outliner, a window that you will use often in Maya.

1. **By default, Shelves will be displayed in the UI (**Figure 04.13**). If they are not visible, choose Display → UI Elements → Shelf.**
2. **Choose Window → Settings/Preferences → Shelf Editor.**

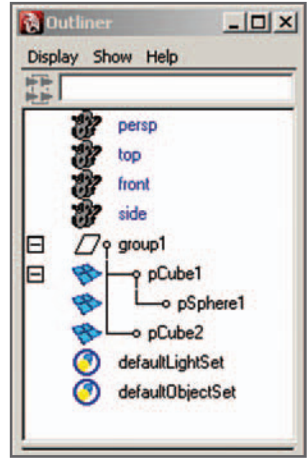


FIGURE 04.16

The Outliner lists the objects, lights and cameras in a scene.

3. **In the Shelves editor, select the Shelves tab. Click on New Shelf (**Figure 04.15a**) and type *Custom* in the Name field. A new empty shelf called Custom should appear alongside the existing ones. Notice that you can rename or delete any Shelf using this palette.**
4. **Click on Save All Shelves.**
5. **Click on the tab for the new Custom Shelf in the main window to make it the active shelf.**
6. **Press and hold Ctrl+Shift while you choose Window → Outliner (**Figure 04.15b**). Release the mouse button, then the keys.**

   This adds the Outliner to your Shelf—a new icon will appear.

   Follow the same procedure to add any menu item to a Shelf. In *Chapter 12*, you'll create a shelf button to execute specific a MEL.
7. **Click on the icon at any time to bring up the Outliner palette.**

Working with Shelves

Maya Help → Using Maya → General → Basics → Preferences and customization → Customize Marking menus and the Hotbox → Customize shelves

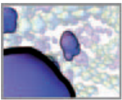
Outliner

The Outliner is one the most commonly used windows in the Maya UI. It lists the objects, cameras, and lights in your Maya scene and enables you to easily select and organize them into hierarchies. While not a default UI Element, you will benefit from having the Outliner (Figure 04.16) close at hand—which is why you added it to your custom shelf. To open it, do one of the following:

Click on the icon as described above in your Custom shelf.

or **From the Main menu bar, choose Window → Outliner.**

Unless you've added any objects to your new scene, the Outliner will only contain the default cameras, light set, and object set. We will use the Outliner and explain hierarchies in one of the upcoming tutorials.



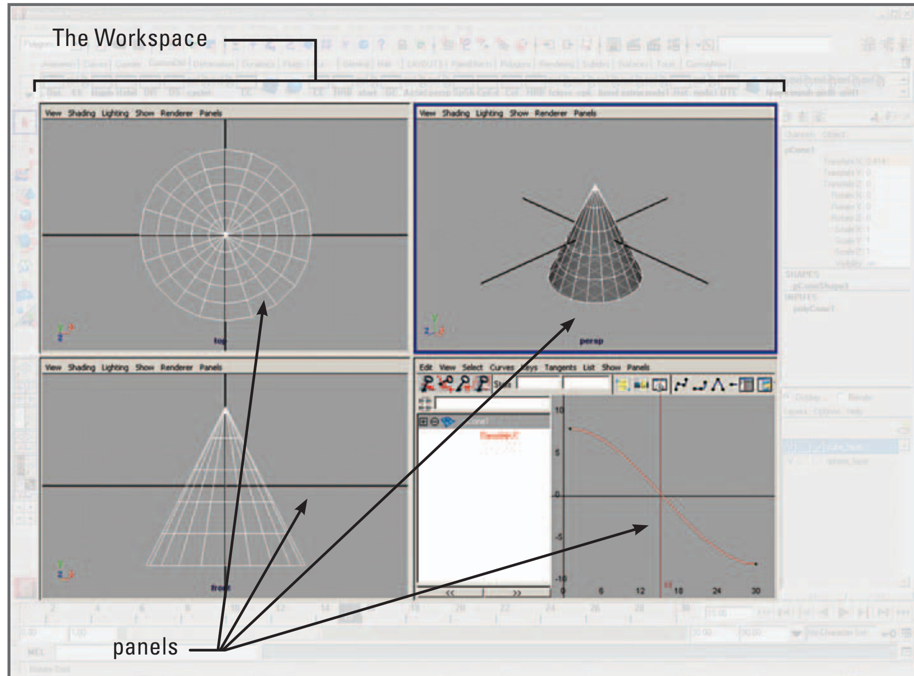


FIGURE 04.17

The workspace with four panels displayed. A blue outline indicates the active panel (upper right).

Outliner

Maya Help → Using Maya → General → Basics → Basics Windows and Editors → Outliner

Workspace and the Panel menus

The workspace is the part of the UI that displays the 3D scene (Figure 04.17). Within it you view and manipulate objects, cameras, lights, and other elements of a scene. It contains one or more *panels*. A panel can contain a view of your scene through a camera—which we call a *scene view*—or any one of Maya's many windows and editors.

The active panel

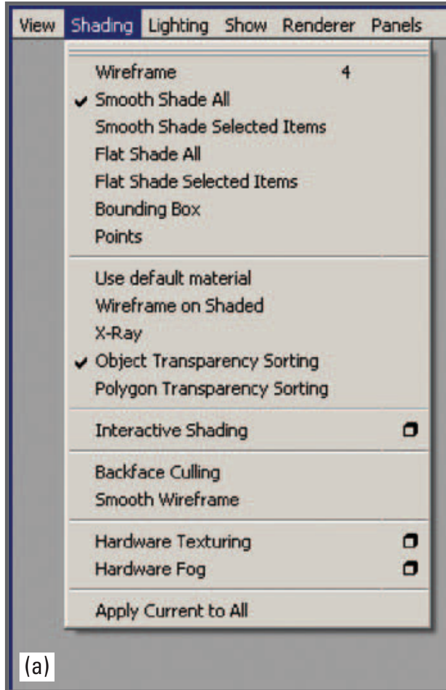
Hotkeys will only work if a panel is active.

Within the workspace, the active panel is the one with a blue box around it (e.g. top-right panel in Figure 04.17). Only the active panel responds when you move the scene view by tumbling the camera. You make a panel active by clicking anywhere inside it, or on its menu bar. You can toggle between a multi-panel arrangement, like that in Figure 04.17, and a full-workspace view of a single panel. To toggle between views:

> **Move your mouse pointer over top of the view you want to toggle and hit the Space bar.**

Panel menus

These menus, at the top-left of the workspace, contain a number of settings that determine what is shown in the workspace, including options for cameras and



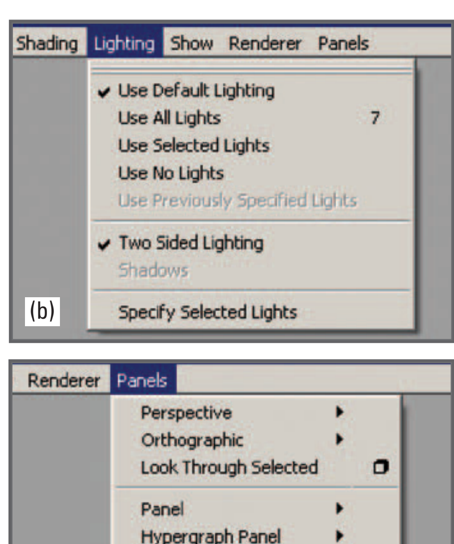


FIGURE 04.18

The Panel menus:

(a) Shading menu,

(b) Lighting menu,

(c) Panels menu.

lights, and for the way that Maya displays various items. Below we describe the more commonly used items from the Shading, Lighting, and View menus. We will discuss the remaining functions and menus only as they are needed to assist you with the book's tutorials and projects.

Panel menus

Maya Help → Using Maya → General → Basics → Basics menus → Panel menus

Shading menu

This menu (Figure 04.18a) is used to set the type of interactive shading used; in other words, how objects will appear as you interact with them in the workspace. Your selection here has no effect on how objects will appear when *rendered*, but can impact the speed with which you work as it relates to the video refresh limitations of your computer. In general, it takes more graphics computing power to display smooth shading and color than it does to display wireframe, bounding boxes, or points. It is common practice to switch between modes regularly as your needs change during a work session. The first eight Shading menu items are the most relevant presently and are explained below. The remaining items are explained in the Help Library:

Wireframe mode hotkey: ***4***

Wireframe displays polygon edges for polygonal objects and **isoparametric** curves (or **isoparms**) for NURBS objects. For navigating in complex scenes, this mode allows considerably faster interaction than smooth shading.

FIGURE 04.19

A polygon torus as displayed in the eight interactive shading modes:

(a) Wireframe

(b) Smooth Shade

(c) Smooth Shade with High Quality Rendering enabled

(d) Flat Shade

(e) Bounding Box

(f) Points

(g) Hardware Texturing with Use Default Lighting enabled. The previous modes didn't display the assigned texture, a checkered pattern.

(h) Hardware Texturing with Use All Lights enabled.

Smooth Shade All shows objects closer to how they'll appear when rendered, with surface color and tone.

Smooth shade mode hotkey: ***5***

Smooth Shade Selected Items same as above for selected items only.

Flat Shade All displays surface color but lacks the smooth surface appearance of smooth shading and therefore is faster to refresh than smooth shading.

Flat Shade Selected Items same as above for selected items only.

Bounding Box represents each item in a scene with a wireframe box defined by the item's bounding volume. It is very quick to redraw and therefore useful when navigating in complex scenes.

Points diplays the surface vertices of an object. This mode is second only to Bounding Box mode for speed.

Hardware texturing hotkey: ***6***

Hardware Texturing displays textures applied to objects, that would otherwise appear only when the scene is rendered. This is useful for orienting the placement of a texture on a surface in the scene view.

Figure 04.19 shows a model of a torus, viewed with each of the eight interactive shading modes.

Lighting menu

There is one item on this menu (Figure 04.18b) that we use regularly:

The hotkey ***7*** toggles between Use Default Lighting and Use All Lights.

Use All Lights shows the effect, on objects, of lights that you have added to a scene. It can be used in conjunction with Smooth Shade, Flat Shade, and Hardware Texturing from the Shading menu.

Panels menu

The first three items on this menu (Figure 04.18c) let you select a camera through which to view your scene.

Perspective cameras display your scene with a visual distortion similar to that which we see through a camera or the naked eye. The degree of distortion is determined by the focal length of the camera specified in the camera settings. The default **Perspective** camera is called ***persp***.

Orthographic cameras show no distortion due to perspective. We often use multiple orthographic views to position objects precisely in 3D space. The default orthographic cameras are called ***front, side***, and ***top***. The default side view is called ***Right Side***, with the camera pointed in the negative X-direction.

To select other orthographic views, such as Left Side and Bottom, or to reset a view to its default settings, choose *View* → *Predefined Bookmarks* →, then make your selection.

Look Through Selected creates a perspective view along the **Z-axis** (described below) of the currently selected item (not necessarily a camera). This feature is handy when you need one item to point at another, such as a spot light to shine on a model; by viewing the scene through the light, you can center it on the model.

The remaining Panel menu items allow you to set panel contents to an item other than a camera view, and to configure panel layouts.

Panel lets you choose an editor or window to display in the current panel.

Hypergraph panel displays the Hypergraph—showing either DG relationships or scene hierarchy.

Layouts lists the possible arrangements of **panels** (called **panes** in the menu). The layout in Figure 04.17 has four panels: a perspective camera (top right); a top-view orthogonal camera; a front-view orthographic camera; and the Graph Editor, which will be described subsequently.

While you're getting used to Maya, we recommend using the default four panes layout. As we go along, we'll show you our favorite layouts for biomolecular and cellular work.

Saved Layouts contains preset arrangements of camera views, editors, and other windows. Several of these layouts can also be accessed using the **Layout shortcut buttons** in the **toolbox (**Figure 04.13**)**.

Tear Off detaches the scene panel from the rest of the Maya interface.

Tear Off Copy creates a free-floating copy of the scene panel. The panel that was copied remains intergrated with the rest of the UI.

Panel Editor allows you to create new panels and customize layouts.

The **space bar** is a hotkey for toggling between single- and multi-panel layouts. With a multi-panel view displayed, placing your cursor over a single panel and quickly striking the space bar will enlarge just that panel. Striking it again will return the multi-panel layout. This is a useful way to quickly enlarge the display to get a closer look at an object in a particular view.

The XYZ coordinate system and vectors

In Maya, every object is located in 3D space according to its distance along the **X-**, **Y-**, and **Z-axis**, with its position described by a vector, using the notation, (X, Y, Z). The vector (0, 0, 0) is called the **world origin**, which is represented in the **workspace** by the centre of the grid in your scene (Figure 04.20). Three-dimensional vectors are used to describe translation, rotation, and scale of objects, as well as RGB (for the Red, Green,

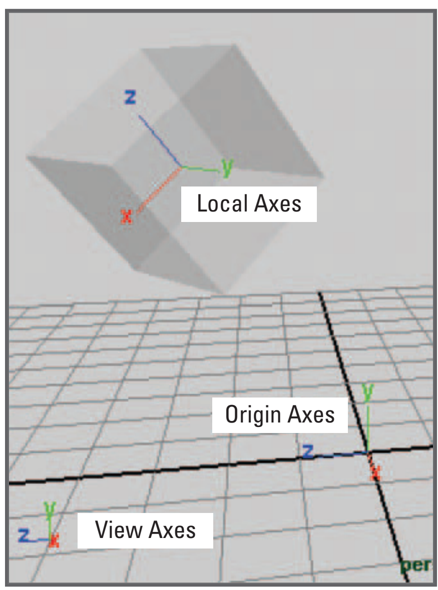


FIGURE 04.20

Local and world coordinates in Maya are represented by axis indicators. Red, green, and blue are used throughout Maya for the X-, Y-, and Z-axis, respectively.

and Blue color system) colors in Maya. Much of what Maya does behind the scenes to transform objects relative to one another is performed by vector mathematics.

There are two coordinate systems in Maya, called **world coordinates** and **local coordinates**. There is only one set of **world coordinates**, centered at the **world origin**, whereas every item in a scene has its own **local coordinate** system, centered at the origin of the item in question—like the hands of the wristwatch in our example on page 81. When an item, such as a sphere, is translated or rotated, so to are its local coordinates; they move with it. The **world coordinates** remain fixed to the **origin** of the scene but appear to move as you move the view around. This is because you are moving the camera, through which the scene is viewed, relative to the **world coordinates**.

The Up axis

Maya creates new objects as if they were Y Up, regardless of the Up Axis settings. In a Z Up world, a new objects is essentially rotated so that its Y-axis is horizontal.

Coordinates in Maya can use either a **Y** or a **Z Up axis**, indicating which axis corresponds to the vertical direction. In 3D modeling programs, animators typically use Y Up, whereas those involved in industrial and engineering design, and medical imaging generally work with Z Up . When using Y Up, the Z-axis represents depth. Conversely, when using Z Up, the Y-axis represents depth. In both cases, the X-axis corresponds to the horizontal direction. To change between the two, choose:

1. **Window → Settings/Preferences → Preferences → Settings.**

2. **Under World Coordinate System, select Y or Z.**

Axis indicators

World coordinates are represented by global axes indicators. These can be displayed at the bottom left corner of the scene view, where they're called the **View Axis**, or at the world origin, where they're called the **Origin Axis**. Local coordinates are represented

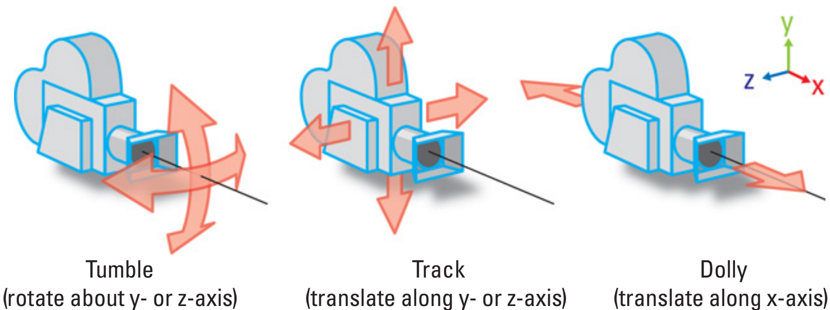


FIGURE 04.21
3D camera movements.

TABLE 04.01
Keyboard/mouse combinations used to view a scene through a camera.

Hold	Drag	Camera move
Alt	**LMB**	**Tumble** (will not work for orthographic views unless you uncheck *Locked* in the Tumble Tool settings)
Alt	**MMB**	**Track**
Alt	**RMB**	**Dolly**
Alt	**LMB+MMB**	**Dolly**
Ctrl+Alt	**LMB**	**Bounding box dolly** draws a marquee (rectangular lasso) around the area in your scene that you wish to dolly in to or out from. This has a similar effect to "zoom" tools in other graphics software.

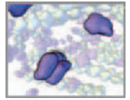
by the **Local Axis**. These indicators will help you stay oriented as you move objects and navigate with your camera through 3D space.

UV coordinates

Maya uses another coordinate system for locating textures and materials on surfaces. This is the 2D **UV** system, which maps out an object's surface. In character modeling for the entertainment industry, an understanding of UVs is important. However, the in silico workflow requires a minimal working knowledge of them. UVs will be introduced in *Chapter 11 Rendering*.

Navigation: Viewing the scene through a camera

When you start Maya, you're shown a view of the scene through the default perspective (or **persp**) camera. In order to look around the scene, you move the camera. You can move a camera in several ways. The most natural and spontaneous is to **tumble**, **track**, and **dolly** the camera using your mouse (Figure 04.21). To tumble is to rotate about the vertical and horizontal axes. To track is to move the camera up and down and side to side. Dollying moves the camera toward or away from its subject, along its Z-axis (or Y-axis if you are using **Z Up** coordinates). Table 04.01 lists the keyboard/mouse combinations used for these camera movements.



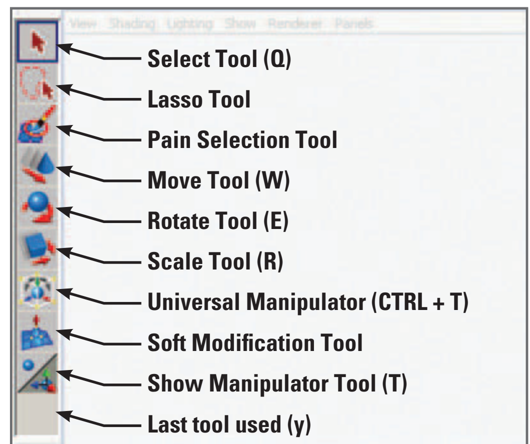


FIGURE 04.22

Transform tools are available in the Toolbox. Hotkeys are indicated in parentheses.

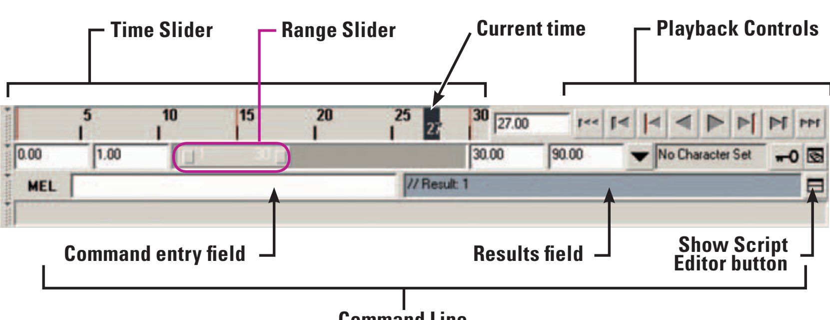


FIGURE 04.23

Time Slider, Range Slider, Playback controls, and Command Line. The current time is indicated by the black bar in the Time Slider. Vertical red lines indicate keyframes. The Command Line has three parts to it. The left field is for entering MEL commands, the right is where Maya reports results and messages, and the button on the far launches the Script Editor.

Toolbox

In addition to the Quick Layout buttons mentioned above, the toolbox (Figure 04.22) contains tools commonly used for selecting and transforming items in your scene. You will use each of the tools in *Tutorial 1*.

Time Slider, Range Slider, and Playback controls

These items are used to control the timing and playback of animation in your scene. The Time Slider (Figure 04.23) displays the timeline, measured in frames, the current time, and the playback (or transport) controls. You can manually move along the timeline in one of the following ways:

Enter the desired frame number it in the current time box.

or **LMB+click anywhere along the timeline—the current time indicator will jump to that spot.**

or **LMB+drag or MMB+drag the current time indicator.**

Button	Function
⏮ ⏭	Jump to the start or end of the playback range
⏪ ⏩	Step backward or forward one frame at a time
⏮ ⏭	Step backward or forward one keyframe
◀ ▶	Play forward or backward. **ESC** key will stop the playback
■	Stop playback (replaces play button during playback)

TABLE 04.02
Playback controls.

The Range Slider is used to set the playback range that is displayed in the Time Slider. It is with the Range Slider that you set the animation start time and end time and the playback range that is displayed in the Time Slider. For example, when creating a three second animation at 30 fps (i.e. 90 frames), you would input 1 for the animation start time and 90 for the end time. To focus on only the first 30 frames, input 1 for the playback start time and 30 for the playback end time, or just drag the ends of the Range Slider bar to these values. Now when you scrub the timeline (see side-bar, this page) or use the playback controls you will be confined to the first 30 frames of your animation. To widen your playback range, simply drag the Range Slider bar ends or input a larger playback end time.

Dragging the ***current time indicator*** along the ***timeline*** is commonly referred to as ***scrubbing***. You can ***scrub the timeline*** to see how an animation looks in the scene view.

The playback controls (Table 04.02) are for playing and for stepping through an animation, and work much like the controls on a home DVD player. They are active within the range set in the **Range Slider**.

Unlike a clock which starts at zero seconds, animations typically start at ***frame 1***. A project that began at ***frame 0*** and ended at ***frame 90***, would actually contain ***91 frames***—more than the three seconds of animation you intended to create.

The Command Line and the Script Editor

You can enter single lines of MEL code in the left half of the Command Line (Figure 04.23). The right half displays system responses, warnings, and error messages. To scroll through previously entered lines, use the up and down arrow keys.

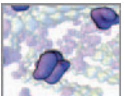
A more effective way to deal with multiple lines of MEL code is through the Script Editor. It can be launched with the icon at the far right of the Command Line, or by choosing:

Window → General editors → Script Editor

We will cover the Script Editor in some detail in *Chapter 12*.

Preferences

Maya stores UI and general application settings in a file called userPrefs.mel. The application reads this file each time you start the software and rewrites it when you click Save in the Preferences Window. Certain preference settings allow you to customize the UI to suit your requirements. These include the option to turn UI Elements on and off and to specify how certain ones are displayed. For example, we find it preferable to have Maya open the Attribute Editor and Tool Settings (both of which are described subsequently) in separate windows rather than embedding them in the main Maya Window, which is the default action. An embedded window causes



undesirable resizing of the workspace each time it is launched. To set the Attribute Editor and Tools Settings to open in windows separate from the main one:

1. **Choose Window → Settings/Preferences → Preferences.**

or **LMB+click the icon at the bottom right of the interface.**

2. **Under Categories, choose Interface.**
3. **For Open Attribute Editor, click the radio button for In Separate Window.**
4. **Do the same for Open Tool Settings.**

Before clicking Save or Cancel, take a look at the other options available under UI Elements. They enable you to show or hide certain elements in the UI. Notice that Attribute Editor and Tool Settings are turned off and Channel Box/Layer Editor is on by default. Maya will let you select only one of these three windows. This just means that the other two won't be displayed until called upon, which is a way to reduce desktop clutter. UI Elements can also be displayed or hidden as follows:

1. **Select UI elements.**
2. **Check or uncheck the box next to the item you want to show or hide, respectively.**

Under UI elements, you'll also find Panel Configurations. These settings determine how the workspace panels are laid out when you start Maya and open a new file.

Under Display, you'll see Performance and View options, including ones that determine how specific items are displayed. Under the heading, Settings, you will set one option for now:

1. **Choose Undo from the Categories list.**
2. **Beside Undo, click the On button (if it's not already selected).**
3. **Beside Queue, click the Infinite button. Infinite undos can potentially use a lot of memory. In most cases this won't be problem and the extra undos can come in handy. However, you may want to set the Queue back to Finite when running a memory intensive animation.**

Finally, under Modules you can tell Maya what to load on startup. Loading modules ties up RAM and can increase startup time. Disabling ones that you don't plan to use alleviates this.

When you are satisfied with the Preferences settings, hit Save. You can, at any time, open **Preferences** and change the settings. We will cover additional preferences as they are needed throughout the book. For more information, refer to:

Using Preferences

Maya Help → Using Maya → General → Basics → Basic Windows and Editors → Preferences

Layer editors

Layers provide a way of organizing items that is independent of the scene hierarchy. Once a group of objects is added to a layer, they can be hidden by turning it off. There

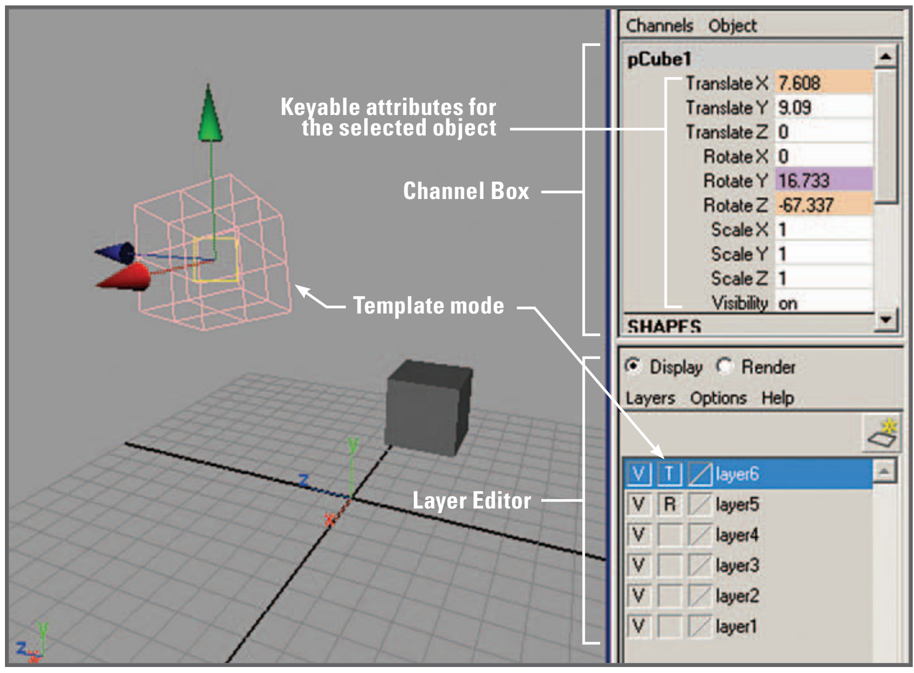


FIGURE 04.24

The Channel Box and the Layer Editor. The colored fields in the Channel Box indicate attributes that have connections to other nodes.

are two types of layer: *Display* and *Render*. Display layers can be viewed in normal, template (T), or reference (R) mode. Template mode displays objects using wireframe shading and protects them from being selected or modified in the scene view. Reference also protects objects from being modified, but displays them in the regular scene view shading mode. Figure 04.24 shows the result of setting a Display layer to Template mode.

Render layers are used to organize lights, cameras, objects, and shaders into separate rendering passes, for later assembly in a compositing program.

Display Layer Editor

Maya Help → Using Maya → General → Basics → Basic Windows and Editors → Display Layer Editor

Render Layer Editor

Maya Help → Using Maya → Rendering and Render Setup → Rendering → Rendering Windows and Editors → Render Layer Editor

Channel Box

In animation, a channel is an attribute that can be animated. The Channel Box (Figure 04.24) displays many of the keyable attributes for a selected item (object, camera, light, etc.). The attributes in the Channel Box are listed under the nodes to which they belong. You can use the Channel Box to easily set attribute values and keyframes. By default, when you create an item in Maya some attributes are set to be keyable and

some, non-keyable. The former will be visible in the Channel Box while the latter will not be listed.

Channel Control editor

The Channel Control editor determines what is and isn't visible in the Channel Box. It is used to make non-keyable attributes keyable, and vice versa, and to lock and unlock attributes in order to prevent or permit their adjustment. For example, once you have a camera placed to your satisfaction, you may wish to lock its attributes to prevent it from accidentally being moved. To open the Channel Control editor:

1. **In the Channel Box, choose Channels menu → Channel Control**

or 2. **In the main window, choose Window → General Editors → Channel Control**

Channel Control

Maya Help → Using Maya → Animation, Character Setup, and Deformers → Animation → Animation Windows and Editors → Editors → Channel Control editor

Attribute Editor

While the Attribute Editor (Figure 04.25) is not a default UI Element, it is used extensively in the Maya workflow, and therefore deserves special mention here. It enables you to view, set, create, and delete attributes, which are arranged by DG node. To open the Attribute Editor:

Choose Window → Attribute Editor.

or **Press Ctrl+A in the workspace.**

Unless an item (object, camera, light, group, etc.) is selected, nothing will appear in the Attribute Editor when you open it.

or **Click the Show or hide the Attribute Editor button on the Status Line.**

or **RMB-click the object in the scene view, or its node in the Hypergraph, and select its name from the Marking menu.**

or **Select Display → UI Elements → Attribute Editor. The Attribute Editor displays to the right of the modeling view.**

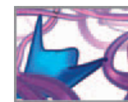
or **In the Hypergraph, select the object or node. From the Hypergraph menu bar, choose Edit → Attributes.**

or **Double-click an object or node icon in the Hypershade, Visor, Multilister, or Outliner.**

Selecting one of the node tabs along the top displays all of the attributes for that node—not just the keyable ones. Attributes that are grayed-out and can't be selected or edited are either non-keyable or locked. Those that are tinted orange have keys set. Right-clicking an attribute name brings up a Marking menu with options such as set key, lock, and create expression, which have to do with animation. You can also rename and delete existing attributes and add your own custom ones using the Attribute menu.

Plug-ins

Plug-ins are software files that exist separately from the Maya application. When activated, or "loaded", using the Plug-in Manager, a plug-in provides additional



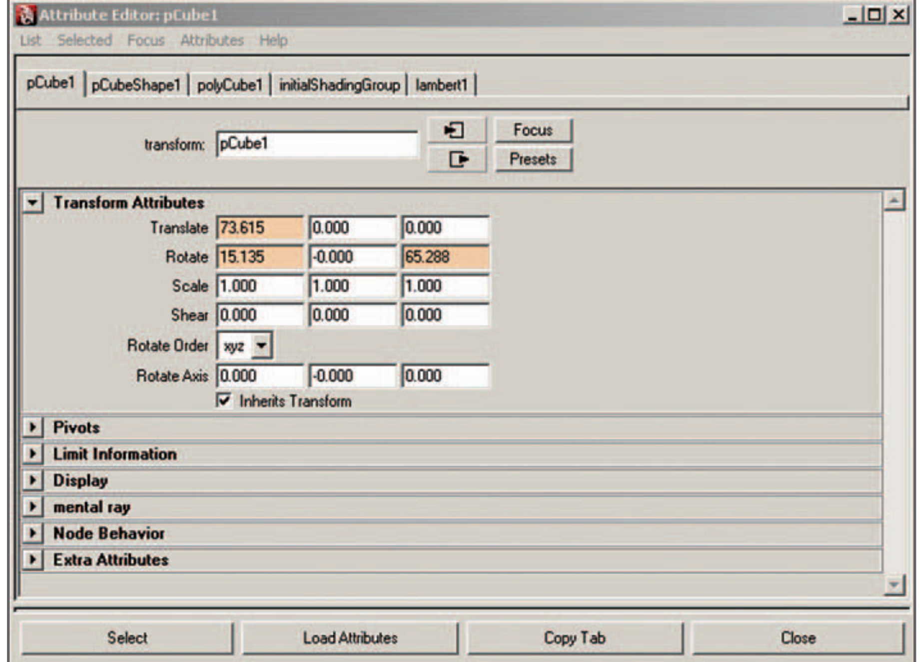


FIGURE 04.25

The Attribute Editor displaying DG nodes and their attributes for a polygon cube.

functionality to Maya. For example, loading the plug-in, objExport.lib, allows you to export a Maya scene as an .obj (or Wavefront) file. Maya comes bundled with a number of plug-ins, and independent developers create their own, many of which can be downloaded for free or purchased on the Internet. For experienced programmers, the Maya Developer's Toolkit includes an API which allows them to create such custom plug-ins using the C++ programming language.

Wavefront file format, denoted by the extension, .obj, is an ASCII 3D scene file format created by Alias/Wavefront.

Loading Plug-ins

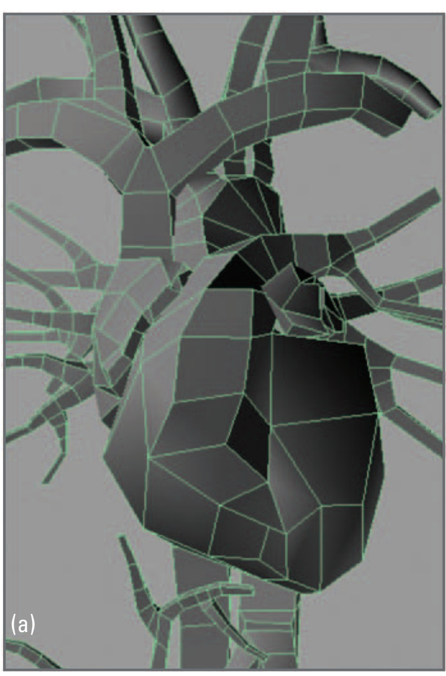
Maya Help → Developer Resources → API Guide → Maya API introduction → Loading a Plug-in

Summary

This chapter is a quick introduction to a deep and rich program. Maya Help references were listed along the way for the reader wishing more information right away. We've covered how to set up a Maya Project and start the program. A brief discussion of Maya's program architecture followed, highlighting its resemblance to biological organization. We then covered the primary elements of the Maya UI to provide a general orientation. In the coming chapters, we will explore much more of the UI through specific examples in modeling, animating, rendering, and dynamics. Nor have we left the DG or Scene Hierarchy behind. These will continue to be of relevance in the projects to follow.

This page intentionally left blank

05 Modeling geometry



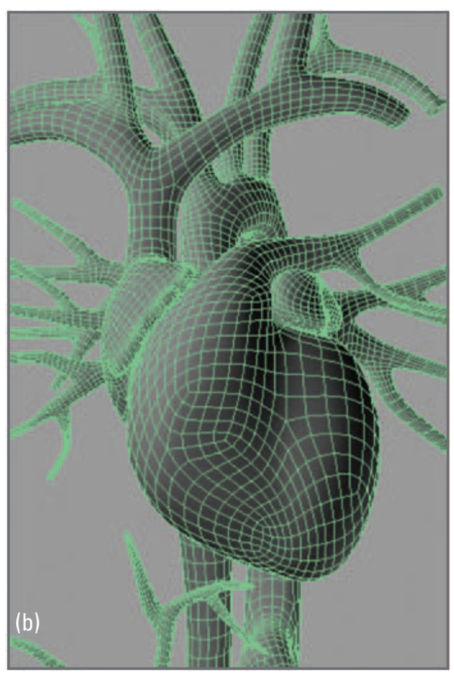


FIGURE 05.01

A simple primitive, such as a polygon cube, can be subdivided and manipulated with modeling tools to create complex topologies like this heart model.

(a) A low polygon-count surface is easy to work with in a scene.

(b) The same model can be smoothed for better rendering results. Smoothing can readily be switched on and off.

Courtesy and copyright © 2006 AXS Biomedical Animation Studio.

Introduction

In this chapter "modeling" means the creation of surfaces and curves in Maya.

Now that we've covered the basic user interface (**UI**), it's time to start using Maya. In this chapter you will learn the difference between NURBS, polygon, and subdivision surface (**sub-D**) objects, and why you might choose one type over another. You will then learn how to create and manipulate **geometric primitives**. A primitive object—a sphere, cone, cylinder, cone, and so on—often forms the basis for a more complex model such as the heart in Figure 05.01. Like a lump of clay, the primitive can be sculpted and deformed, added to, cut apart, and put back together.

Moreover, certain primitive geometric shapes have considerable merit on their own for in silico biology, without being significantly altered from their basics shapes. Spheres and cylinders are design paradigms in nature. In other words, there are many structures in the natural world that are topologically equivalent to these primitive geometric shapes. Notable examples are shown in Figure 05.02. The projects in *Part 3* of this book all make extensive use of geometric primitives for approximating biological structures (atoms, protein fibers, and cell bodies).

For those who wish to explore building more complex models, plentiful tutorials exist in Maya Help, in print, and online, about advanced, high-end modeling in Maya. We have highlighted several of our favorites, under the heading *Learning Maya* in the *Further reading* section.

The Create menu

Part of the permanent menu set, this menu is used to create geometric primitives and NURBS curves, along with lights, cameras, and locators. Selecting the option

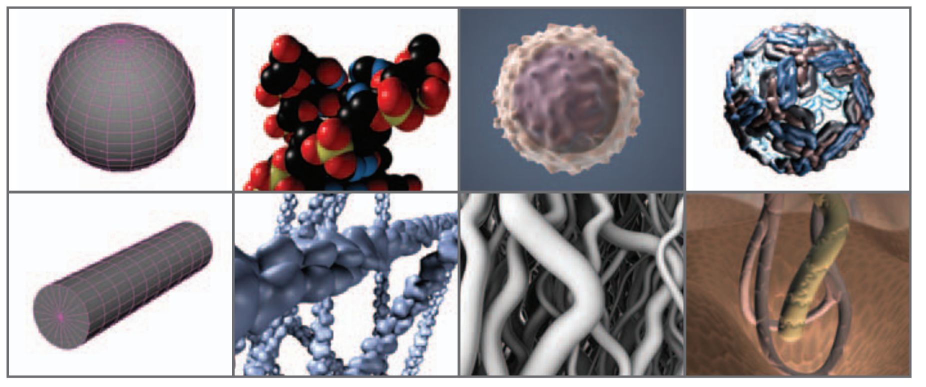


FIGURE 05.02

Geometric primitives can be used as a starting point to represent various entities in nature:

Top row
NURBS sphere, atoms, cells, virus capsid;

Bottom row
NURBS cylinder, biopolymers, protein fibers, blood and lymphatic vessels.

Cell and blood vessel images courtesy and copyright © 2006 AXS Biomedical Animation Studio. The virus capsid was created from Protein Data Bank file 1K4R[1] using UCSF Chimera software (http://www.cgl.ucsf.edu/chimera/).[2]

box next to a menu item brings up the Create options window, allowing you to alter the settings. Many of the primitives can also be created by clicking the appropriate buttons on the Surfaces (NURBS), Polygons, and Subdivs Shelves.

Creation options

In the following menus

Create → NURBS Primitives

Create → Polygon Primitives

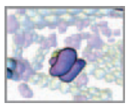
you have options called Interactive Creation and Exit on Completion. They will be either checked or unchecked in the menu, indicating whether they're on or off. Interactive Creation lets you choose the initial location of the created object and scale it interactively. When this option is turned off, Maya places newly created objects at the world origin by default, with a scale value of 1. You can subsequently move and scale an object after you've created it. With Exit on Completion turned on, Maya exits the Create tool after you've created one object. If the option is turned off, the Create tool remains active until you select another tool (e.g. the Select tool).

NURBS modeling

NURBS is an acronym for Non-Uniform Rational B-Splines and describes a class of mathematically defined curves (or **splines**) and surfaces computer graphics. Creating objects using these curves and surfaces in Maya is called NURBS modeling. NURBS modeling makes sense for smooth, organic shapes in film and for industrial design. Also, because NURBS surfaces can be created curves, they have particular utility in modeling fibrous structures, as you'll see later on in the chapter and again in *Chapter 17* in *Part 3* of this book.

The Surfaces menu set

Everything you need to make and work with NURBS is available through the Surfaces menu set. To activate the set, use the pull-down menu at the far left of the Status Line (Figure 05.03). Alternately you can use the hotkey, **F4**. Some of the tools housed in the menu set will come into play in the projects in *Part 3*, although you will access them using MEL (Maya Embedded Language) commands, and not through the UI.



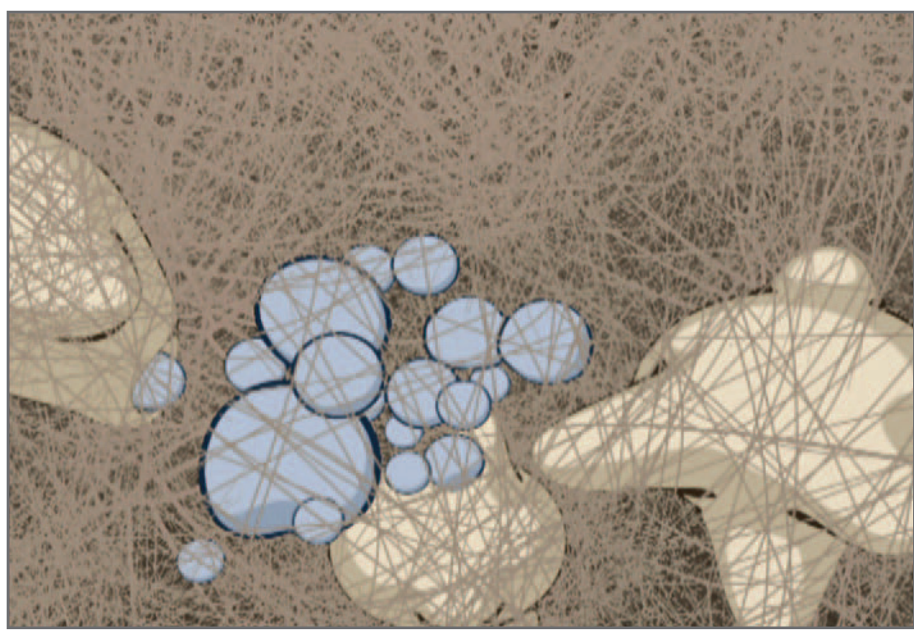


FIGURE 05.03

Spline curves were traditionally used in 3D modeling and animation to build NURBS surfaces and to provide directional pathways for animation. More recently, splines have become models in their own right, being used to recreate hair, fur, and plants. In the example shown here, medical animator Jen Platt used splines to create a visualization of the metamatrix, the fibrous network that extends through cell nuclei, cytoplasms, and extracellular space within living tissues.

Courtesy and © 2006 Jennifer A. Platt.

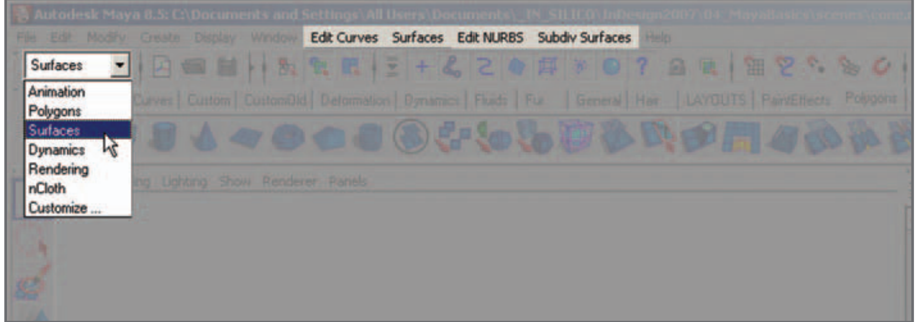


FIGURE 05.04

The Surfaces menu set.

NURBS curves (splines)

The term spline originated in shipbuilding where it described a thin piece of wood used to define hull shape. The spline was bent into a smooth curve by metal weights—the control points.

A spline is essentially a line (straight or curved) composed of segments. Its path is defined by points called **control vertices** (**CV**s for short). In Maya, splines are most often used as motion paths to guide animation and to build NURBS surfaces, which is the subject of a tutorial in this chapter. However, when endowed with color and thickness using Maya Paint Effects, splines themselves can become models. The fibrous environment shown in Figure 05.04 was created almost entirely using splines and Paint Effects.

Spline components

In Maya, the constituent parts of a curve or surface are called components. CVs and Edit Points (**EPs**) are the components of a spline. Depending on the mathematical

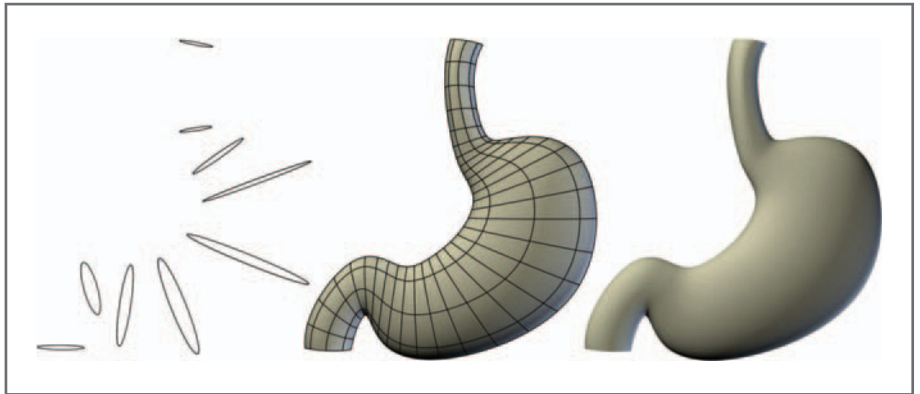


FIGURE 05.05

NURBS modeling is well suited to smooth, curvilinear surfaces. This stomach model was created using the Loft tool on a group of curves (or splines). Lofting interpolates a curved surface between curves.

nature of a curve, it may or not pass through the CVs. EPs, on the other hand, always lie on the curve. You can change the shape of a curve by moving either its CVs or EPs.

NURBS surfaces

NURBS surfaces are mathematically described shapes connecting splines in 3D space. Figure 05.05 shows a stomach model created by **lofting** between a series of splines. NURBS surfaces are used widely in automotive and architectural design, and in other disciplines where smooth, curvilinear surfaces are required. Organic objects like organs such as lungs, kidneys, bowels, and blood vessels are well suited to NURBS modeling. Maya has an extensive suite of tools for building and working with splines and NURBS surfaces. These can be found in the Edit Curves, Surfaces, and Edit NURBS menus. You will explore some of them further along when you create a fiber from two splines.

The smoothness of a NURBS surface is determined in part by its divisions attributes—sections and spans—which are set at creation time and can be accessed through its history node. Figure 05.06a shows two spheres, each with a different number of divisions. When Maya renders NURBS, it converts them to polygons through a process called **tesselation** (Figure 05.06b). Together, the tesselation settings for object and its subdivisions determine how smooth it appears when rendered.

With NURBS, smooth 3D surfaces can be made quickly from relatively few curves. This also means you can deform such a surface by simply altering one of its constituent curves. However, despite their advantages, NURBS can prove difficult for creating complex topologies. Likewise, because a NURBS surface is dependent on its constituent curves, multiple surfaces cannot be combined into one continuous surface, as can be done with polygons. This can result in an undesirable seam between two pieces of geometry.

Hiding seams, so they are not visible to the camera, is one of the tricks to good NURBS modeling.

NURBS surface components

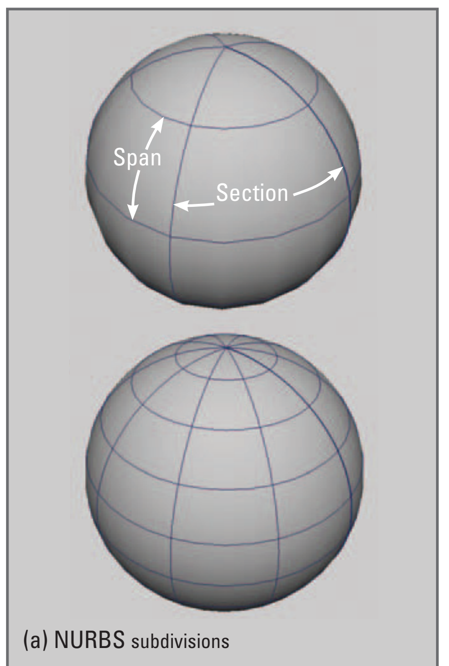
Because a NURBS surface is built from splines, it too has CVs that can be used to manipulate its shape. A NURBS surface (Figure 05.07a) is defined by curves called isoparms, which can be added to or deleted from a surface. Hulls are straight lines connecting CVs that can lie on or off of the surface. Hulls are handy ways to quickly deform a NURBS object.

FIGURE 05.06

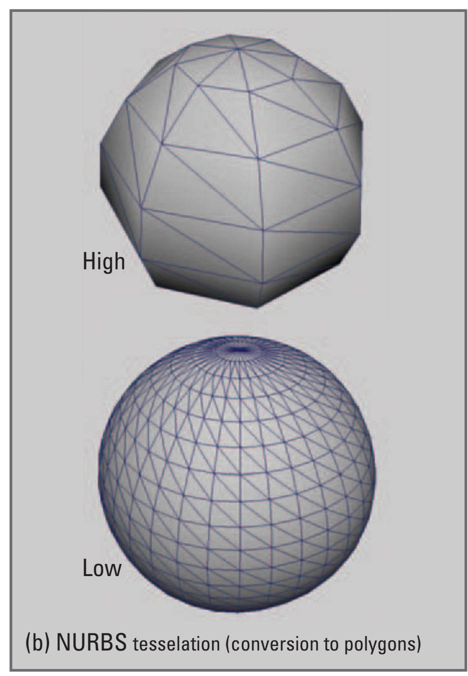
(a) NURBS subdivisions (measured in **sections** and **spans**) affect surface smoothness and the number of control points available for manipulating a surface. The top sphere has fewer subdivisions than the one below it.

(b) To a large degree, **tesselation** (the conversion of a NURBS surface to polygons at render time) determines the surface smoothness of an object when it's rendered. The top sphere has lower tesselation settings, and will therefore be coarser when rendered, than the one below it. The surface lines in this image demark polygons resulting from tesselation of NURBS spheres.

Together, subdivisions and tesselation settings give you control over the appearance of an object when it's rendered.



(a) NURBS subdivisions



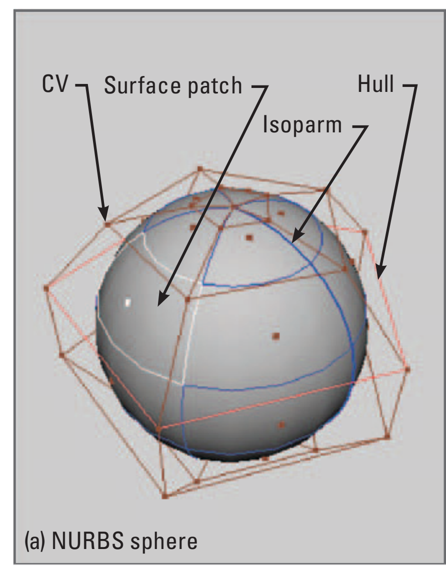
(b) NURBS tesselation (conversion to polygons)

FIGURE 05.07

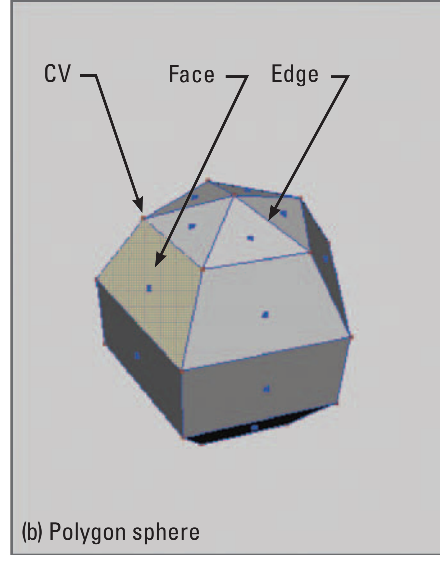
A 5 × 5-division primitive sphere:

(a) The components of a NURBS model.

(b) The components of a polygonal model.



(a) NURBS sphere



(b) Polygon sphere

NURBS surface normals

Normals are lines perpendicular to a surface (Figure 05.08a). Generally speaking, normals are said to point outward from an object's surface, thereby indicating its *direction*. Reversing a surface's direction makes the normals point in the opposite

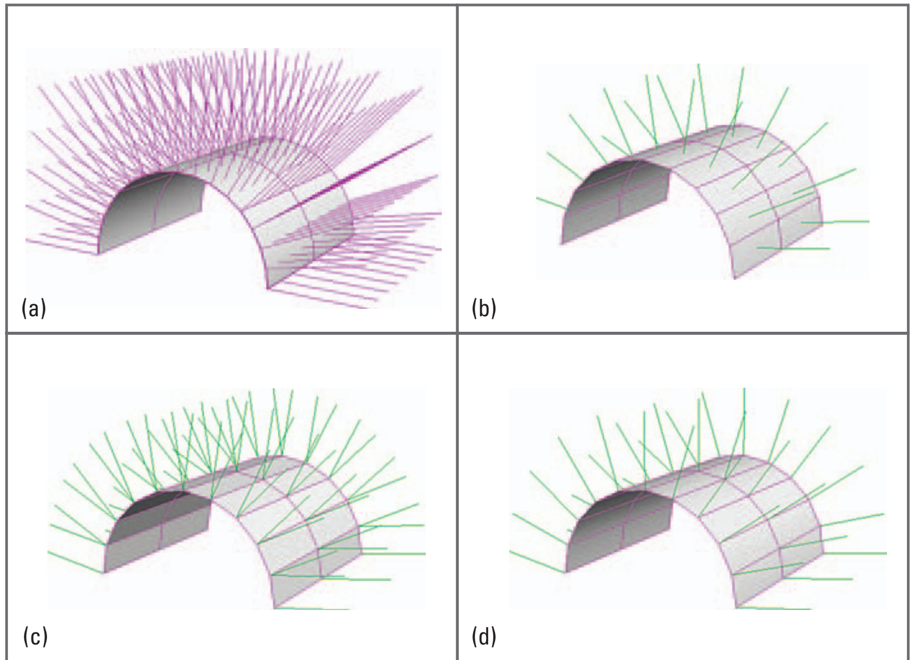


FIGURE 05.08

Normals are a way of indicating the direction of a surface.

(a) NURBS surface normals.
(b) Polygon face normals.
(c) Polygon hard vertex normals.
(d) Polygon soft vertex normals.

direction, or "inward". One consequence of surface direction is the way in which Maya Dynamics detects collisions between two or more objects. More on this in *Chapter 07*.

NURBS modeling

Maya Help → Using Maya → Modeling → NURBS modeling

Polygonal modeling

A polygon surface is a continuous mesh comprised of many individual polygons (facets), which can be pushed, pulled, rotated, or extruded to create any conceivable topology. Polygon models are used widely in computer animation for entertainment and video game development. In these applications, polygon surfaces are preferable to NURBS for the detailed topology required for characters' faces, hands, and the like. Similar forms can be created with NURBS, but often require multiple surfaces to be joined, resulting in visible seams between them, which show up as unwanted lines in renderings. For example, a model of a hand may require a separate NURBS object for each finger, each joined at its base to the hand with a visible seam. In contrast, polygon objects can be joined to create a seamless, continuous mesh.

Polygons are usually three- or four-sided, called **triangles** and **quads** (short for quadrangles), respectively.

In a film animation workflow, where Maya is rendering images from a scene, NURBS surfaces can be slower to render than their polygon equivalents. This is due to the computation required for conversion of NURBS to polygons before they can be processed by Maya's rendering engine.

Point for point, polygonal surfaces tend to be computationally lighter than NURBS, which can speed the interactive display of geometry. This is one reason why polygons are preferred for video game development—they allow for quick video refresh.

Unlike a NURBS model, the smoothness of a polygon surface is constrained by the number of polygons that comprise it—called its **poly count**. A low poly count object is easier to work with but produces a coarser rendered surface than an equivalent shape with a high poly count. Maya's smooth command, available in the Polygon menu,

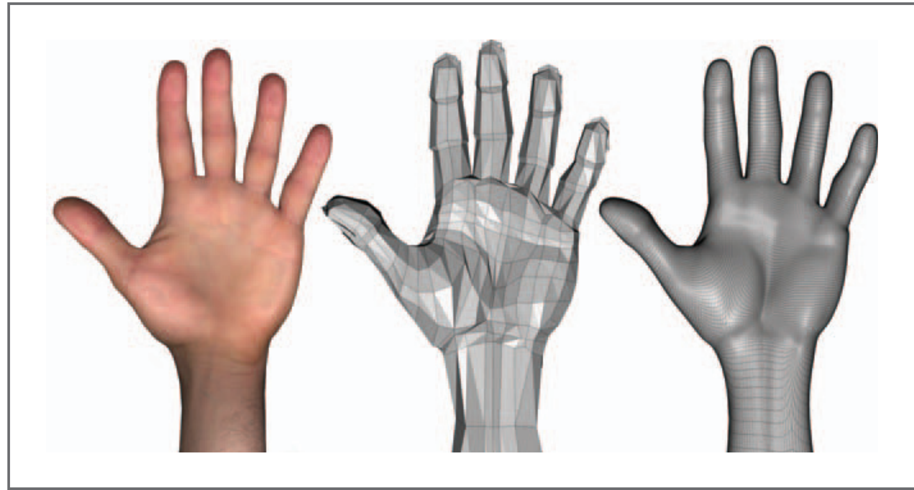


FIGURE 05.09

Polygon modeling is used extensively in 3D character animation. Models can literally be drawn in 3D from 2D templates, the way this 3D hand model (center) was drawn from reference photographs of a real hand (left). The rough polygonal model (center) was smoothed (right) by subdividing its surface into smaller polygons.

allows you to work with a low poly count model and then subdivide the surface into more polygons to make it smoother for rendering. This was the workflow used with the heart model shown in Figure 05.01 and the hand in Figure 05.09. You will use it to smooth the polygon model in *Tutorial 05.03: Make and deform a polygon sphere* coming up.

The Polygons menu set

Hit the the hotkey, **F3**, to activate this menu set, or use the pull-down menu at the far left of the Status Line. This menu set contains the menus and tools you'll use for building and editing polygonal surfaces.

Polygon components

Like a NURBS surface, a polygon object can be described in terms of its components (Figure 05.07b): CVs (or points), faces, and edges. Each has its utility in modeling.

Polygon face normals

Like NURBS surface normals, polygon face normals indicate the direction of a surface (what's considered *outside* versus *inside*). Face normals project from, and run perpendicular to, the center of each polygon face (Figure 05.08b). Reversing the direction of a face makes its normal point in the opposite direction.

Polygon vertex normals

Vertex normals project from polygon vertices. They indicate the rendering smoothness of a polygonal surface. When vertex normals are *hard*, they run perpendicular to their associated faces. Consequently, Maya renders a hard edge between those faces (Figure 05.08c). Soft vertex normals indicate an average perpendicular direction for their associated face. Maya renders a smooth edge between these faces (Figure 05.08d).

Polygonal modeling

Maya Help → Using Maya → Modeling → Polygonal modeling

Subdivision surfaces

sub-Ds are similar to polygon surfaces, but allow you to vary the level of detail in different regions of the mesh. This way you can put in detail where you need it, while keeping the mesh coarse where you don't. Our tutorials and projects don't require this capability, so sub-D modeling won't be discussed further. The Maya Help menu provides basic information and instruction for sub-D modeling.

Sub-D modeling

Maya Help → Using Maya → Modeling → Subdivision Surface modeling

So which model type should I use?

When choosing a modeling approach, consider the relative merits and limitations mentioned above. Most important is the nature of the object you wish to model. Is it smooth with relatively little surface detail or does it have detailed topology? Is it curvilinear or jagged? Is it composed of multiple parts. Curvilinear shapes, where seams between objects can be hidden or are an integral part of the design, are well suited to NURBS modeling—hence its popularity with industrial designers. Intricate topologies like those of a hand, a heart, or a face are best handled with polygons. In the world of tissues, cells, and molecules, which is our focus, the same principles apply. A polygon that can be sculpted into the shape of a crawling cell makes sense for the project in *Chapter 16*. *In Chapter 17*, NURBS modeling is well suited to the creation of the meandering collagen fibers of the **extracellular matrix** (**ECM**).

One thing to bear in mind is that a NURBS object can be converted to polygons. Therefore, you can begin sculpting a NURBS surface, then change it into a polygons when you're ready to add more detailed topology, or to connect several objects into one (join fingers to a hand, for example). Polygons, on the other hand, cannot be converted directly to a NURBS surface. For the time being, let's start with a NURBS primitive and see how to manipulate it in your Maya scene.

Tutorial 05.01: NURBS primitive modeling

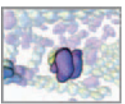
In this tutorial you will create a NURBS sphere primitive and manipulate it by changing its attributes in the Channel Box and using the transform tools. A sphere is an appropriate shape to begin for its utility in representing the atoms of biomolecules, and the form of many cells and viruses.

Start Maya and set up a Project

To begin, start Maya and create a new project. When you start Maya it opens to a new, untitled scene file.

1. **Double-click the Maya desktop icon or the Maya application icon in your Applications directory.**

2. **Choose File → Project → New to open the Project window. The current project name and directory path are displayed.**



3. **Browse to the directory in which the project will reside. For starters, you can simply use Maya's default project folder. In Windows this would be something like:**

   C:\Documents and Settings\User\My Documents\maya\projects

4. **Name the project Learning_Maya.**

5. **For the tutorials in this chapter, you will only need directories for image and scene files. Type images and scenes in the appropriate fields.**

   This is where Maya will write your render files. You can always edit these fields later by choosing File → Project → Edit Current.

6. **Click on Accept or Cancel.**

When you create a new project, Maya sets it as the current project. If you wish to switch to another, existing project, do the following:

1. **Choose File → Project → Set.**
2. **Browse to find the appropriate project directory.**
3. **Click OK.**

Chances are, when you started Maya it opened with a perspective panel view. If it didn't, in the Panel menu set:

Choose Panels → Saved Layouts → Single Perspective View.

Activate History

Make sure History is active: the History icon, in the Status Bar must be depressed. This will ensure that attribute you set when you create a model will remain editable as you continue to work with it.

Create the sphere

1. **Choose Create → NURBS Primitives.**
2. **Turn off Interactive Creation.**

Releasing your mouse over the **option box** ❐ symbol in any menu launches an **Options** window.

3. **Choose Create → NURBS Primitives again, but this time, select Sphere ❐. This launches the NURBS Sphere Options box.**
4. **In the Options box, choose Edit → Reset Settings.**
5. **Customize the settings to match the ones in** Figure 05.10**.**

   The key attributes to note here are the Radius and the Number of Sections and Number of Spans.

The settings you specify in an **option box** will be applied each time you use that tool until you change the settings again using the **Options window**.

6. **Click Apply to create the sphere and keep the Options window open (useful for creating multiple objects with different dimensions).**

or **Click Create to create the sphere and close the Options window.**

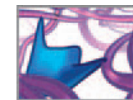


FIGURE 05.10

The Create settings for a NURBS sphere primitive.

This creates a new object called nurbsSphere1. Take a moment to tumble, track, and dolly the scene view to get a comfortable view of the object. If you can't see the sphere at first, press the hotkey, **A**, to frame all objects in the scene.

The Reset Settings or Reset Tool steps are precautions in case the settings had previously be changed from their defaults by you or another user. By starting with the defaults, you're more likely to get the expected results.

Select the sphere

When you create an item in Maya, it is automatically selected. By default, the most recently selected item is indicated with a green wireframe mesh. If multiple items are selected, all will be indicated by white wireframe outlines, except for the most recent selection, which will be green. If you click in the workspace away from the sphere, you will deselect it, and the wireframe mesh will disappear. To reselect it do one of the following:

LMB+click on any one of the tools in the Tool Box (Figure 05.11**), then LMB+click on the sphere in the workspace.**

or **LMB+click on any one of the tools in the Tool Box, then LMB+drag to draw a bounding box selection around all or part of the sphere, then release the LMB. Any object lying partly within the box when you release the mouse will be selected.**

or **LMB+click on the Lasso Tool in the Tool Box, then LMB+drag to draw a lasso selection around the sphere or any part of it.**

or **LMB+click on the sphere's name in the Outliner.**

When selected, the name sphere1 (or whichever name you gave the object) appears at the top of the Channel Box (Figure 05.12). This is the sphere's transform node, below which are its attributes that can be animated: Translate X, Translate Y, and so on (much more on animating attributes in the next chapter!). Below these are two headings,

NURBS cubes are different from other NURBS primitives in that they are composed of six individual planes, grouped together. A sphere, in contrast, is one contiguous piece of geometry.

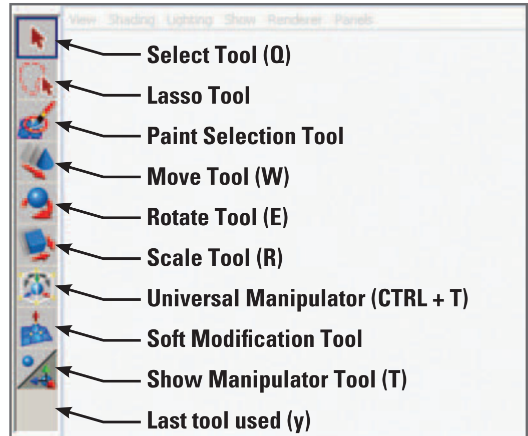


FIGURE 05.11

Transform tools are available in The Tool Box. Hotkeys are indicated in parentheses.

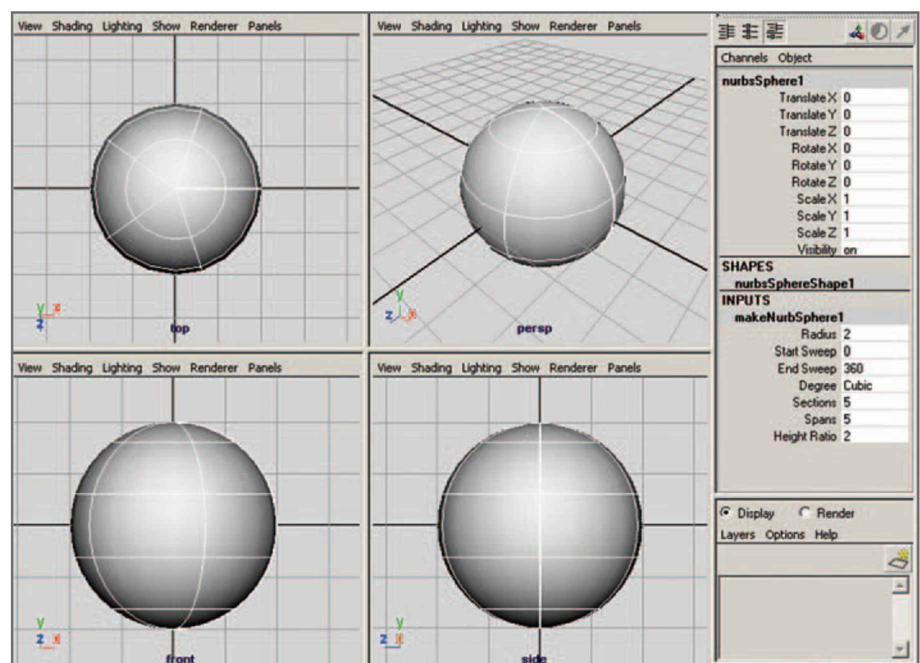


FIGURE 05.12

The transform node attributes of the NURBS sphere appear in the Channel Box (on the right). This panel layout, known as the Four panel view displays three orthogonal and one perspective camera.

Shape and Inputs, under which are the names sphere1Shape and NURBSsphere1. These are the sphere's shape and history nodes, respectively. Clicking on the history node, NURBSsphere1, reveals its attributes, which are the same ones you had access to when you created the object using the NURBS sphere Options window: Radius, Sections, and Spans.

Rename the sphere

As with most actions in Maya, there are several ways to rename an object. Here are two:

Click on the name nurbsSphere1 in the Channel Box and type a different name such as sphere1.

or **Double-click nurbsSphere1 in the Outliner and type sphere1, then hit Enter.**

Names in Maya cannot contain spaces. The convention for multi-word names is to capitalize the first letter of all but the first word, as in **mySphere**, or to use underscore characters, as in **my_sphere**.

View the sphere with different interactive shading modes

Use the Shading menu (in the Panels menu) to try different shading modes. When you're done:

1. **Choose Shading → Smooth Shade All.**
2. **Choose Wireframe on Shaded**

The wireframe indicates the location of isoparms, which will enable us to see when the sphere is rotated about its axes.

Channel Box Tips

For an object to show up in the Channel Box, it must be selected.

You can LMB+drag select multiple fields at once in the Channel Box. This is handy for assigning more than one field the same value.

Translate the sphere

There are many ways to transform objects in Maya. To do so interactively, use the **Transform Tools** (shown in the Tool Box in Figure 05.11). With the sphere selected, activate the **Move Tool** in one of these three ways:

Choose Modify → Transformation Tools → Move Tool.

or **LMB+click the Move Tool icon in the Tool Box.**

or **Hit its hotkey, "w".**

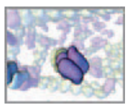
The red, green, and blue Move Tool Manipulator handles should appear, arising from the center of the sphere; if they don't, make sure the sphere is selected then try one of the above steps again. As with axis indicators in Maya, red, green, and blue correspond to the X, Y, and Z directions for Manipulator handles as well. When you change tools you'll see that the shape of the handles changes too. Figure 05.13 shows the handles corresponding to the Move (translate), Rotate, and Scale tools.

Now practice translating the sphere freely in any direction you wish. With the sphere selected:

LMB+drag the yellow box at the center of the sphere.

or **MMB+drag the mouse pointer anywhere in the workspace. Because the sphere is selected, it will move as you move the mouse.**

The manipulator handles are aligned with the local axes of the sphere and travel with it as it moves. If you watch the Translate attributes in the Channel Box as you move the sphere around, you will see them changing simultaneously because you are moving the sphere in all three dimensions at once while you are engaged in real



The hotkeys + and – increase and decrease the size of the manipulator handles, respectively.

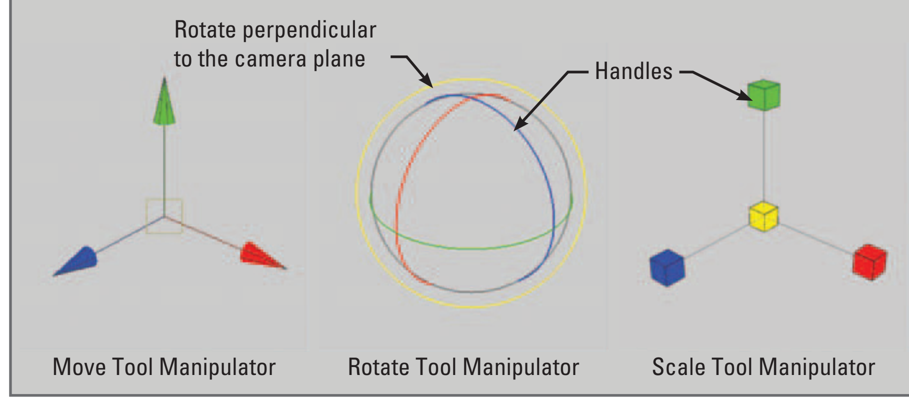


FIGURE 05.13

The Manipulator handles for (from left to right): the Move Tool; the Rotate Tool; the Scale Tool.

project work. Rarely, however, is this desirable in animation; it is much easier to keep track of the spatial relationships of items in your scene if you apply a transformation in one dimension at a time. You do this by clicking and dragging one manipulator handle (the X, Y, or Z) at a time. When active, a manipulator handle turns yellow. To transform the sphere in one dimension do one of the following:

When referring to transformations, we use the terms "dimension" and "axis" interchangeably in this book. For instance, a translation in the X-dimension is the same as a translation parallel to the X-axis.

1. **Select the sphere and hit "w" to activate the Move Tool.**
2. **LMB+click on one of the three manipulator handles to make it active.**
3. **LMB+drag on the active handle.**

or **MMB+drag anywhere in the workspace.**

The Four panel view

You may have noticed that distance is difficult to gauge when translating the sphere in the perspective view. This is due to the visual distortion applied by this camera. The orthographic views show space without this perspective distortion and make it easier to see where objects "really are", relative to one another in Maya's 3D space. We find the Four panel view (shown in Figure 05.12) an effective layout for visualizing a scene, including multiple biological molecules or cells moving in complex environments. The Four panel view enables you to tumble around your models in perspective view in order to see them from all angles, while using the front, side, and top orthographic views to accurately gauge 3D spatial relationships. To set the workspace to the Four panel view:

In the Panels menu, choose Panel → Saved Layouts → Four panel view

or **Press the Four panel view icon, , in the Layout shortcuts panel on the lower-left side of the main window.**

Rotate the sphere

You can activate the Rotate Tool in the same way you did the Move Tool. The rotate manipulator handles are red, green, and blue circles corresponding to each of the X,

Y, and Z axes. In addition, there is an outer yellow circle which allows you to rotate the sphere in a plane perpendicular to the camera axis. To rotate the sphere freely about all axes:

LMB+drag the yellow box at the center of the sphere.

or **MMB+drag the mouse pointer anywhere in the workspace. Because the sphere is selected, it will rotate as you move the mouse.**

Note the changing Rotate X, Y, and Z attribute values in the Channel Box. To transform the sphere in one dimension do one of the following:

1. **Select the sphere and hit the "E" hotkey to activate the Rotate Tool.**
2. **LMB+click on one of the three manipulator handles to make it active.**
3. **LMB+drag on the active handle.**

or **MMB+drag anywhere in the workspace.**

The manipulator axes indicate the sphere's local coordinates. When you translated the sphere, the Move Tool handles traveled with it, but remained oriented to the world coordinates (global X, Y, and Z axes). When you rotated the sphere, its local coordinate system (axes) rotated with it, as indicated by the Rotate Tool handles, which changed orientation relative to world coordinates.

Scale the sphere

The Scale Tool works in much the same way as the Move and Rotate Tools. However, when you scale an object freely, it scales uniformly in all three dimensions. To scale the sphere freely:

Entering 0 for its Scale X, Y, and Z attributes will cause an object to disappear in the scene view. You can bring it back simply by entering positive values in the Scale attribute fields.

1. **Activate the Scale Tool by hitting the "R" hotkey.**
2. **LMB+drag the yellow box at the center of the sphere left to shrink the sphere and right to enlarge it.**

or **MMB+drag the mouse pointer anywhere in the workspace, left to shrink the sphere and right to enlarge it. Because the sphere is selected, it will scale as you move the mouse.**

Note the changing Scale X, Y, and Z attribute values in the Channel Box. To scale the sphere in one dimension do one of the following:

1. **Select the sphere and hit the "R" hotkey to activate the Scale Tool.**
2. **LMB+click on one of the three manipulator handles to make it active.**
3. **LMB+drag the active handle.**

or **MMB+drag anywhere in the workspace.**

Like the Move Tool Manipulator handles, those for the Scale Tool travel with the sphere and remain oriented to world coordinates.

Change the Move Tool Settings

Many of the tools in Maya, including the Move, Rotate, and Scale Tools, have settings that you can change. As an example, you'll change a Move Tool setting and observe its effect. To open the Move Tool settings palette:

1. **Double-click the Move Tool icon in the toolbox.**

   or **With the Move Tool active (i.e. selected), choose Display → UI Elements → Tool Settings.**

2. **In the Move Tool Settings window, select Object.**

   With this setting, the Move Tool will translate objects relative to their local (*object*) rather than world coordinate systems.

   You don't need to close the palette to apply the change. Changes are applied instantly in Tool Settings.

Now when you translate the sphere with the Move Tool, you do so along its local rather than world axes. If the sphere has been rotated, its local axes are no longer aligned with the world axes. In this case, dragging just one Move Manipulator handle with change all of the Translate attributes in the Channel Box. This is because the Channel Box attributes are in world, not local coordinates. There are some situations where it's desirable to set the Move Tool to translate an object along its local coordinate system. For the most part, however, we recommend using World or Global settings in the Tools Settings palette. To reset the Move Tool Settings:

When open, the Tool Settings palette will display settings for the current active tool. If no tool is active, the palette remains blank.

1. **With the Move Tool active, open the Tool Settings window.**
2. **Select Use Defaults.**
3. **Close the window.**

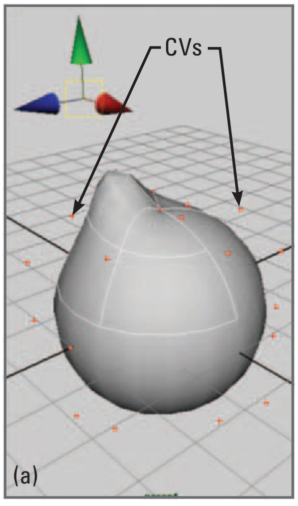
Transform the sphere using the Channel Box

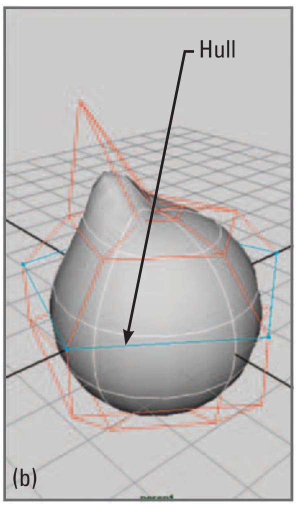
Here, you'll use the Channel Box to reset nurbsSphere1 to its initial position and scale, then hide it from view.

The Visibility attribute is of the **Boolean** data type, unlike Translate, Rotate, and Scale, which are **floating point** (or decimal number) attributes. Boolean data can have one of two values: "on" or "off". Maya accepts the integers 1 and 0 for on and off Boolean values, respectively. We will discuss Maya data types in some detail in *Chapter 12*.

1. **With the sphere selected, LMB+drag from the Translate X field down to the Rotate Z field. This selects multiple fields at once in the Channel Box, allowing you to assigning more than one field the same value at one time.**
2. **Enter 0 in one of the selected fields. Your sphere should return to the world origin with no rotation.**
3. **LMB+drag from the Scale X field down to the Scale Z field.**
4. **Enter 1. This returns the sphere to its original size.**
5. **To hide the sphere, enter 0 or "off" in the Visibility field**

   or **Use the hotkey, Ctrl+H.**





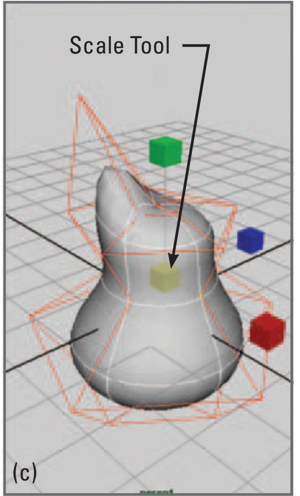


FIGURE 05.14

A NURBS surface can be reshaped by moving and scaling its components.

(a) Moving a CV changes its curve (isoparm) and deforms the surface.

(b) A hull is a line that connects CVs.

(c) Scaling a hull moves has the effect of moving its constituent CVs uniformly; in this case, toward the center of the sphere.

The Visibility attribute is one of several ways you can hide an object in Maya. To show the sphere again:

1. **Enter 1 or on for Visibility in the Channel Box.**

or **Use the hotkey, Ctrl+Shift+H. This shows the object that was last hidden.**

Save your scene

Click in any view panel, so that hotkeys will work, then hit Ctrl+S

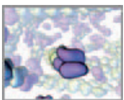
or

1. **Choose File → Save Scene.**
2. **Enter a name for your scene file and hit Save.**

When you save a scene, Maya defaults to the directory specified in the Project Settings.

Tutorial 05.02: Deform the sphere using components

Before leaving your NURBS sphere behind, let's change its shape by transforming its components. Figure 05.14 shows the transformations you'll apply to each type of component. You will do manually what you might normally employ a Maya deformer, such as the **Lattice Deformer**, to do. However, through moving components by hand, you will get a sense of how a deformer works to change the shape of an object. Furthermore, surface deformations of the kind you're about to make are analogous to the kinds of shape changes cells undergo. It's not too much of a stretch then to imagine how you might model a deformable cell in Maya.



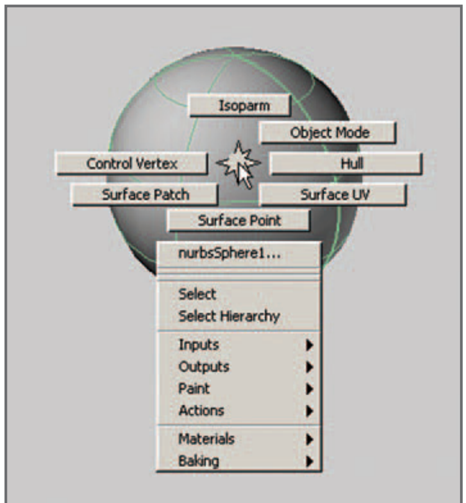


FIGURE 05.15

The NURBS component marking menu is revealed by an RMB + click over the object.

Move a CV

1. **RMB+click on the sphere in the scene view to activate the Component Marking menu (**Figure 05.15**).**

2. **In the Marking menu, move the cursor over CV and release the RMB.**

3. **Hit the "w" hotkey to activate the Move Tool.**

4. **LMB+click to select one of the CVs.**

5. **Create a spike on the sphere's surface by LMB+dragging the CV away from the sphere's center (**Figure 05.14a**).**

or **LMB+dragging one of the Move Tool Manipulator handles.**

Imagine the spike you just made is a cellular protrusion—a pseudopod—extending through the ECM of some tissue in the body! Next you will contract the middle of the sphere using a component called a **hull**.

Scale a hull

A hull is a line connecting CVs that correspond to one isoparm and forming a closed loop (Figure 05.14b). Hulls are a convenient tool for deforming surfaces quickly.

1. **RMB+click on the sphere to bring up the Marking menu.**

2. **In the Marking menu, move the cursor over "Hulls" and release the RMB.**
   A Hull appears as a line connecting CVs in a loop around the sphere.

3. **Hit the "R" hotkey to activate the Scale Tool.**

4. **Click on a Hull. A color change will indicate that it's active.**

5. **MMB + drag left on the center of the Scale Tool Manipulator to shrink the Hull and cause a narrowing in the sphere, as shown in** Figure 05.14b**.**

Selection modes and masks

If you find yourself modeling extensively with components, you will want to learn how to use selection modes and selection masks. They allow you to tailor the kinds of objects or components that you are able to select with your mouse. For instance, if you're going to be selecting hulls for a while, you can set the selection mode to *select by component type* and the component mask to *hulls*. This makes hulls visible for the objects in your scene and means you can select only them, and not other components or object transforms. These settings are made using the selection modes and masks buttons in the Status Line at the top of the main window.

Help → Learning Resources → Getting Started with Maya → Maya Basics → Lesson 3 Viewing the Maya 3D scene → Selection modes and masks

Before moving on to the next tutorial, save your file:

1. **Choose File → Save**

or **Use the key combination, Ctrl+S.**

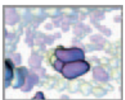
Tutorial 05.03: Make and deform a polygon primitive

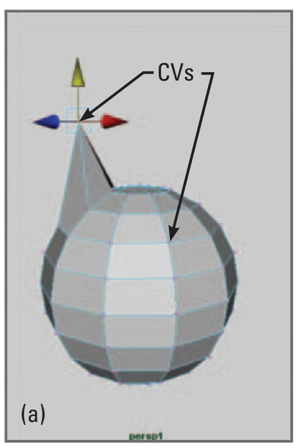
Now that you're familiar with basic NURBS geometry, you'll create a polygon sphere and see how its components differ from its NURBS cousin. Once you begin moving components you'll notice that the form of the polygonal object lacks the smooth, organic form of the NURBS sphere. As a result the polygon sphere doesn't appear as *cell-like*—that is, until you apply a smoothing node to it. To get started, the selecting, renaming, translate, rotate, and scale exercises you went through for the NURBS sphere will be no different for the polygon, so we don't repeat the steps here. To create a polygon sphere:

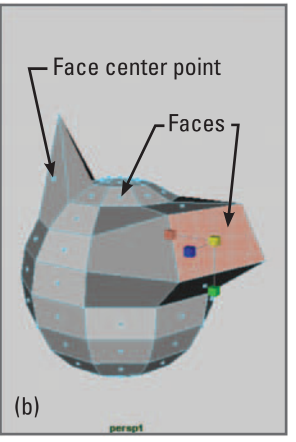
1. **Choose Create → Polygon Primitives**

2. **Turn off Interactive Creation.**

3. **Choose Create → Polygon Primitives again, but this time, select Sphere ❐. This launches the Polygon Sphere Options box.**







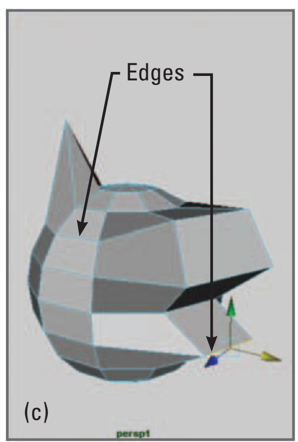


FIGURE 05.16

Polygon surfaces can be reshaped by transforming individual components: CVs, faces, and edges.

(a) Translating a CV.

(b) Translating and scaling a face.

(c) Translating an edge.

4. **Customize the settings:**
   (a) **Radius: 2**
   (b) **Subdivisions Around Axis: 10**
   (c) **Subdivisions Along Height: 10**
   (d) **Axis: Y**
5. **Click Create to create the sphere and close the Options window.**

This makes a new polygonal object with a transform node called pSphere1. If the pSphere1 overlaps nurbsSphere1, select one of the two and move it beside the other. Now let's look at the components of pSphere1. Figure 05.16 shows the results of the following steps.

Move a CV

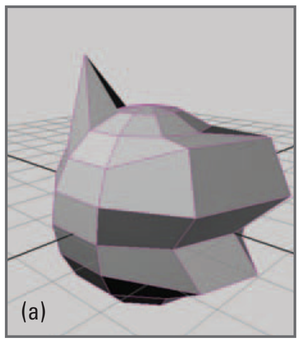
1. **RMB+click on pSphere1 in the scene view to activate the component Marking menu.**
2. **In the Marking menu, move the cursor over "CVs" and release the RMB.**
3. **Repeat the steps you used to move a CV of the NURBS sphere.**

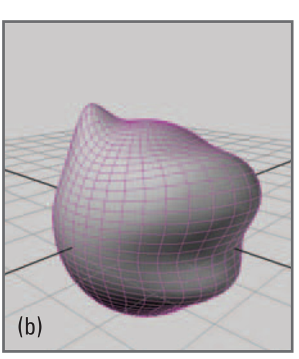
   The effect on the shape of pSphere1 will be somewhat different from that of nurbsSphere1; it lacks the smoothness we saw in the NURBS object.

Move and scale a polygon face

A face is an individual polygon, with corners defined by CVs.

1. **RMB+click on pSphere1 to bring up the Marking menu.**
2. **In the Marking menu, move the cursor over Face and release the RMB.**





INPUTS
polySmoothFace1
polyTweak1
polySphere1

Method	Exponential
Continuity	1
Divisions	2
Smooth UVs	on
Keep Border	on
Keep Selection Border	on
Keep Hard Edge	off
Keep Map Borders	Internal
Keep Tessellation	on
Subdivision Levels	1
Divisions Per Edge	1
Push Strength	0.1
Roundness	1

(c)

FIGURE 05.17

A polySmoothFace node subdivides a polygonal surface. As long as construction history is active, smoothing is reversible and editable.

(a) Geometry before smoothing.

(b) Geometry after smoothing.

(c) The polySmoothFace node and its attributes appear in the Channel Box when the smoothed object is selected.

A wireframe mesh will appear on the sphere. At the center of each face is a point used for selecting it.

3. **Hit the "w" hotkey to activate the Move Tool.**
4. **Select the point at the center of a face (its *center point*).**
5. **LMB+drag the Move Tool Manipulator to move it away from the center of the sphere.**
6. **With the face still selected, hit the "R" hotkey to activate the Scale Tool.**
7. **MMB+drag the center of the Manipulator to scale the face.**

Move an edge

An edge is a side of an individual polygon (or face) that spans two CVs.

1. **RMB+click on pSphere1 to bring up the Marking menu.**
2. **In the Marking menu, move the cursor over Edge and release the RMB.**

   A wireframe mesh will appear on the sphere indicating the polygon edges.
3. **Hit the "w" hotkey to activate the Move Tool.**
4. **Select an edge by LMB+clicking on it.**
5. **LMB+drag the Move Tool Manipulator to move it away from the center of the sphere.**

When you move an edge or a face, you alter the edges and faces that share the same CVs. Furthermore, while you cannot scale or rotate an individual CV, you can scale and rotate a group of CVs, a face, or an edge.

Smooth the sphere

Due to the number of subdivisions you began with, pSphere1 is rather coarse in appearance. Here you'll add a node that smooths the surface of pSphere1 (Figure 05.17).

1. **Select pSphere1.**
2. **Activate the Polygons menu set by pressing the hotkey, F3.**
3. **Choose Mesh → Smooth.** Don't worry about the Smooth options; with construction history turned on, you can edit them at any time.
4. **Select the new smooth node, polySmoothFace1, in the Channel Box and enter 2.0 for its Divisions attribute (see** Figure 05.17c**).**

If you plan to edit the surface of a smoothed object, do so with smoothing Divisions set to 0. After making the changes to the object's shape, set the Divisions back up to 1 or 2.

Figure 05.17 shows pSphere1 before and after smoothing. Divisions is the number of times Maya subdivides the polygons in order to create a smoother appearance. At any point you can set this attribute to 0 to return the sphere to its pre-smoothed appearance. If you plan to edit the surface of a smoothed object, it is best to do so on the original geometry—with subdivisions set to 0. After editing, you can set Divisions back up to 1 or 2 to smooth the object again. Through construction history, the shape changes will be reflected in the smoothed version of the model.

It's easy to imagine, with the modest beginnings of your smoothed sphere, how you could approach modeling distinct shapes—cells or organs, for example—from primitive geometry. And you haven't even touched the modeling tools that are used to subdivide, extrude, cut, and append the constituent polygon faces! While it's tempting to delve into specific examples here, we want to keep rolling toward your goal of writing MEL scripts to make models and drive in silico simulations. If you wish to explore Maya's model-making capabilities further, we encourage you to look up the resources listed under *Learning Maya* in the *Further reading* section. In the next tutorial, we will explore the nodes and connections that make up pSphere1 for a better understanding of what is actually going on when you create and edit geometry.

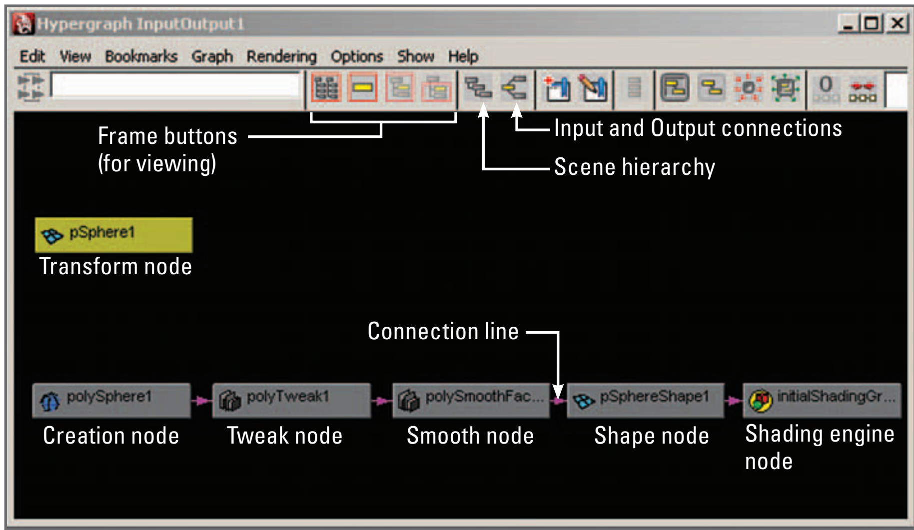
Tutorial 05.04: Construction history

In this exercise, you will see how your polygon sphere is actually constructed in the Maya Scene Graph and how you can use those connections, through construction history, to alter the creation attributes you set. To begin, open your scene from *Tutorial 05.03* or copy the scene file from the CD to your projects directory and open it in Maya (see file path below).

05_Modeling/scenes/tutorial_05_03_done.ma

The Hypergraph revisited

In the previous chapter we discussed the Dependency Graph (**DG** for short) and DG nodes, the entities that make up a Maya scene. Now that you have created some geometry in Maya, we can revisit DG nodes with a specific example. This section is less a tutorial than an exploration of what you've already created, using Maya's DG and scene hierarchy viewer, the Hypergraph. Figure 05.18 shows the Hypergraph UI Elements, some of which you'll use to explore the nodes comprising your polygon sphere. The tool bar buttons are shortcuts to items located in the menus. You can move about the Hypergraph the same way you would in a scene view with an orthographic camera—using the dolly and track key/mouse combinations. The view buttons labeled in Figure 05.18 become useful in more complex scenes for targeting specific nodes in a large network.



Tip: Popup Help will not work in the Hypergraph unless you make it the active window by clicking on it.

FIGURE 05.18

The Hypergraph displays the nodes and connections for your polygon sphere.

To view pSphere1 in the Hypergraph:

1. **Select pSphere1 in the Outliner or scene view.**
2. **Choose Window → Hypergraph.**
3. **In the Hypergraph Choose Graph → Input and Output Connections**

or **Press [icon] in the tool bar.**

When you first created pSphere1, four nodes determined what it looked like and where it was in space. These are:

pSphere1	the **Transform** node
polySphere1	the **Creation** *or* **History** node
pSphereShape1	the **Shape** *or* **Mesh** node
InitialShadingGroup	the **Shading Engine** node

You then deformed and smoothed the sphere, creating two more nodes:

polyTweak1	the **Tweak** node
polySmoothFace1	the **Smooth** node

You can select any of these nodes by clicking on it in the Hypergraph. When selected, a node and its attributes appear in the Channel Box and in the Attribute Editor if it's open. The pink lines with arrows indicate connections between attributes. In the DG one node, in relation to another, can be either an **upstream** or a **downstream node**, depending on the direction of information flow. The connection line arrows indicate this direction, which is typically shown left to right in the Hypergraph (Figure 05.19). You can see which attributes are connected between two nodes by moving your mouse cursor over the connection line; the attribute names will pop up.

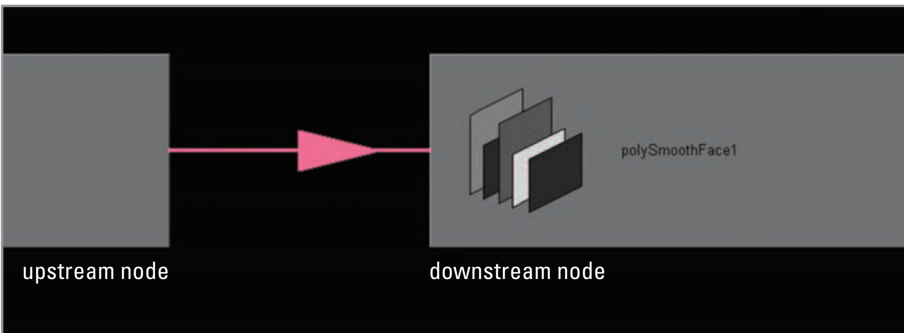


FIGURE 05.19

Arrows on connection lines in the Hypergraph indicate the direction of information flow.

Internally, Maya considers each node to be of a certain **type**. These types don't necessarily coincide with the common names given in Maya Help and employed by Maya users. For example, a **shape** node to a user is of type, **mesh** to Maya. Similarly, while we call polySphere a **history** node, to Maya it is of type, polySphere.

The transform node, pSphere1, has no attribute connections to the other nodes. Instead, it has a hierarchical relationship with the shape node. pSphereShape1 is the child of pSphere1 (clicking on the Scene Hierarchy button, , will isolate the transform and shape nodes). Below the transform node is polySphere1, the creation node. It holds the radius and subdivisions attributes that were set when you created the sphere. Placing your cursor over the connection line between polySphere1 and the tweak node, polyTweak1, reveals the attributes that are connected between the two: polySphere1.output and pTweak1.inputPolymesh. In plain language, the output of the creation node is the data that the tweak node operates on. Next polySmoothFace1 takes the *tweaked* data and applies its smoothing operation. The *smoothed* data is then input to the shape node, pSphereShape1, which put the data into viewable form of how the surface will appear.

To the right of pSphereShape1 is the default shading engine which is used for rendering the sphere. When created, all geometric primitives are connected to this default **render node**. It helps determine the appearance of the sphere—the combined effects of color, texture, and lights—when rendered. You can see the additional nodes used in rendering by selecting initialShadingGroup and clicking on . We will explore render nodes in more detail in *Chapter 08*. For now, they serve as an example of how everything in a Maya scene, even when it comes to rendering, is described by nodes and their connections. The Hypergraph provides a bare bones view of these nodes and connections, allowing you to zero in one or two at a time, or pull back and look at bigger chunks of the Maya Scene Graph. Other tools, such as the Attribute Editor, which we will explore next, provide a more detailed presentation of nodes and their attributes.

The Hypergraph

Maya Help → Using Maya → General → Basics → Basic Windows and Editors → Hypergraph

The Attribute Editor

The Attribute Editor is a convenient tool for viewing, setting, creating, and deleting attributes. It provides much of the functionality of the Channel Box, in terms of setting attributes numerically, but also allows you to make attribute connections

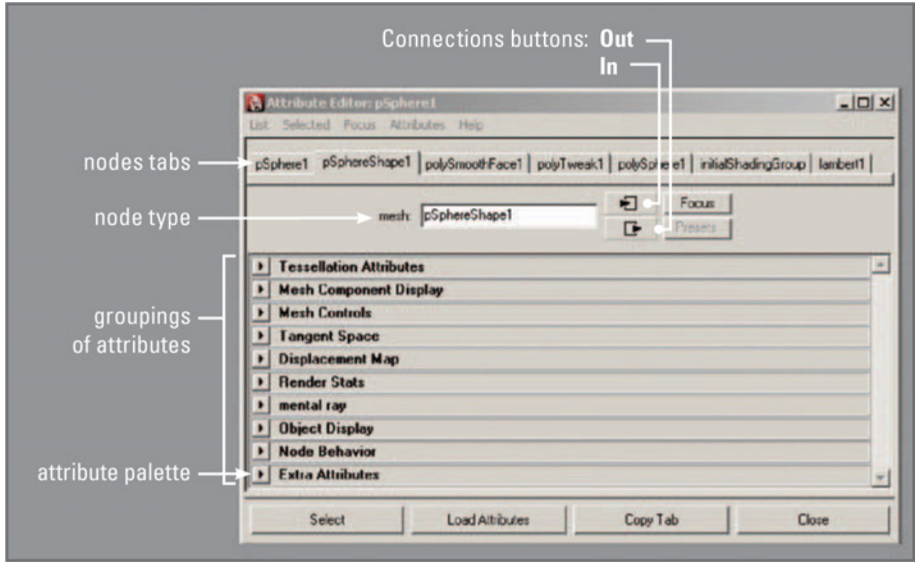


FIGURE 05.20

The Attribute Editor is a useful tool for viewing attributes, setting their values, and making connections with other nodes.

between attribute of different items, which comes in handy when creating shading networks. Let's open the Attribute Editor for pSphere1 (Figure 05.20):

1. **Select pSphere1.**
2. **Choose Window → Attribute Editor**

or **Use the hotkey A.**

or **Double-click on the pSphere1 icon in the Outliner.**

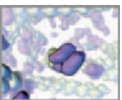
Nodes are represented as file-folder-like **tabs** running along the top of the editor (see Figure 05.20); by default, the shape node is displayed when you select an object. You can view each node by clicking on its tab. The In and Out Connections buttons reveal upstream and downstream nodes, respectively. In the next section, you will use the Attribute Editor to edit the polygon sphere through its history connections.

Attribute Editor

Maya Help → Using Maya → General → Basics → Basic Windows and Editors → Attribute Editor

Construction history

Your polygon sphere provides a very simple and convenient example of construction history in Maya; it involves only one node, polySphere1, along with its connection to the shape node, pSphereShape1. Let's change the attributes of polySphere1 and observe the effect. You could do this through the Channel Box, in the same way you set the transform attributes for the NURBS sphere. Instead, let's take this opportunity to become more familiar with the Attribute Editor.



1. **Select pSphere1 and get a good view of it in the perspective view.**
2. **In the Panel menu set, choose Shading → Smooth Shade All**

or **Use the hotkey 5.**

3. **Open the Attribute Editor.**
4. **Click on the polySphere1 tab.**
5. **LMB+Drag (scrub) the Radius slider and observe the effect in the scene view.**
6. **Scrub the Subdivision Axis and Subdivisions height.**

   You can instantly see the effect these attributes have on surface smoothness.

7. **When you're done adjusting the attributes, close or minimize the Attribute Editor.**

If you select polySphere1 in the Channel Box, you will see the attributes change there as you scrub the sliders in the Attribute Editor. Also, note the effect that changing Subdivisions has on the surface protrusions you created in the previous Tutorial—they tend to move around the surface as you scrub the Subdivisions attributes. In Figure 05.18 you can see that the upstream creation node polySphere1 connects to the downstream tweak node polyTweak1, a node that was created automatically when you began manipulating components. Those manipulations, or **tweaks**, are specific to numbered components of your sphere. As you change the number of CVs by increasing or decreasing Subdivisions in the creation node, the CVs themselves get renumbered. The tweaks follow their CVs by number resulting in protrusions that change position on the sphere depending on the number of Subdivisions. For this reason, it is a very good idea to settle on the number of Subdivisions before deforming a piece of geometry.

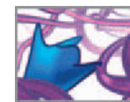
Unlike the Subdivisions of the creation node polySphere1, changing the polySmoothFace1 Divisions does not impact the location of shape tweaks on the sphere because the smoothing node lies downstream from the tweak node. In other words, the tweaks (push and pull of surface components) occur before smoothing is applied. This brings up an important fact about construction history: changes to a node can affect only itself and others that lie downstream form it, but not upstream nodes.

You can see how useful construction history can be for making changes to a model after its creation. However, history nodes add overhead to a Maya scene. The history nodes in our current example are the creation, tweak, and smooth nodes. These add to file size, increasing the time it takes to save and open a file. More importantly, they slow animation playback and rendering. This is because history adds to the calculations Maya must make in order to determine where an object is and what it looks like at a given frame. Deleting history—deleting the history nodes—is one way of optimizing a scene for faster playback and rendering. To delete an object's construction history, select the object and do one of the following:

Choose Edit → Delete by Type → History.

or

Select the history nodes in the Hypergraph and hit Delete on your keyboard.



If you deleted history for pSphere1, choose Edit → Undo to bring the history nodes back. Next, instead of deleting the nodes, you will delete the connection between pSmoothFace1 and pSphereShape1, effectively cutting the sphere off from its history. You will then reconnect the two nodes to learn how to make attribute connections using the **Connection Editor**. With pSphere1 selected:

1. **Open the Hypergraph.**
2. **Select the pink connection line running between polySmooth Face1 and pSphereShape1.**
3. **Hit the Delete key.**

Select pSphere1 in the scene view or the Outliner and inspect it in the Channel Box. You no longer have access to the creation (Radius and Subdivisions) and smoothing (Divisions) attributes. You can select polySphere1 and polySmoothFace1 in the Hypergraph and edit their attributes in the Channel Box. However, changing them will have no effect on the sphere because of the broken connection. Let's fix that by reconnecting the severed nodes using the Connection Editor.

Construction History

Maya Help → Using Maya → General → Basics → Transforming objects → Maya's interface → Construction history

The Connection Editor

The Connection Editor is shown in Figure 05.21. It is used to make and break connections between attributes of two nodes. The left- and right-hand fields display the attributes of upstream and downstream nodes, respectively. When two attributes are connected, the output value of the upstream (left-hand) attribute becomes the input value of the downstream (right-hand) attribute. To see the Connection Editor in action, let's use it to connect pSphere1 to its severed history node:

1. (a) (i) **In the Hypergraph, MMB + drag the icon for polySmoothFace1 over to of the icon for pSphereShape1.**

   The icon itself won't move, but your cursor will change to indicating that your are setting up a connection.

   (ii) **Release the MMB.** A Connection pop-up menu will appear

   (iii) **Select Other (**Figure 05.22**).** This launches the Connection Editor, with the attributes of both nodes displayed.

   or (b) (i) **Choose Window → General Editors → Connection Editor.**

   (ii) **Select polySmoothFace1 in the Hypergraph and click Reload Left in the Connection Editor.**

   (iii) **Select polySphereShape1 in the Hypergraph and click Reload Right in the Connection Editor.**

2. **In the left field, select Output. In the right field, select In Mesh (**Figure 05.21**).**

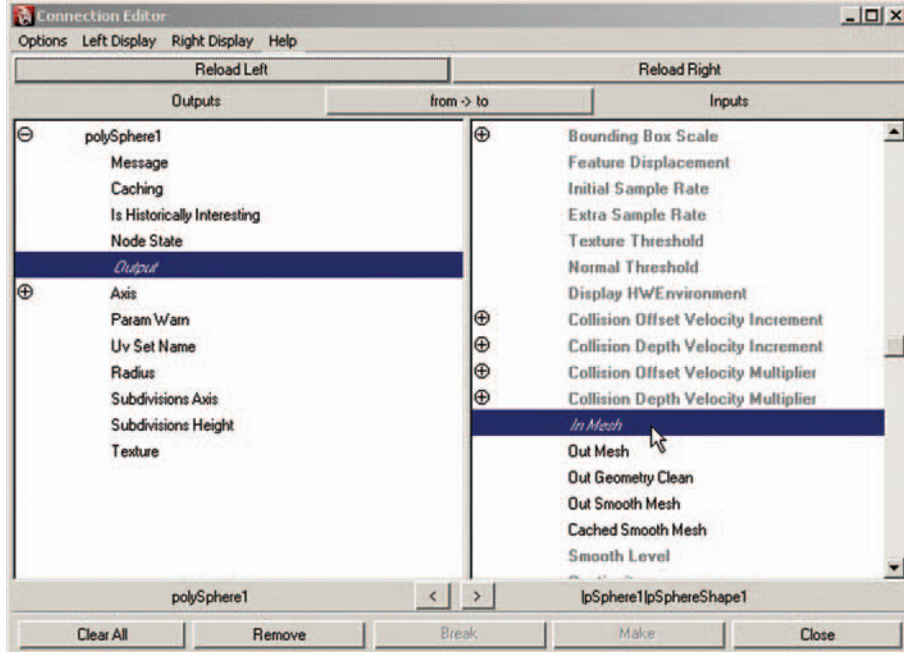


FIGURE 05.21

The Connection Editor allows you to make and break connections between attributes of different nodes. Shown here is the connection between the attributes of the history and the shape nodes of the polygon sphere.

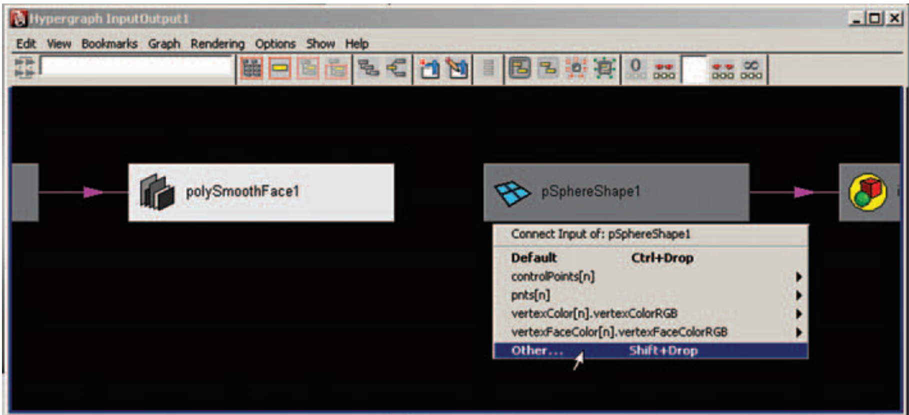


FIGURE 05.22

When connecting attributes in the Hypergraph, a choice of *Other* launches the Connection Editor.

That's all it takes to connect two attributes! You can now change the Radius and Subdivision attributes and pSphere1 will respond. The Connection Editor is a handy tool for making quick connections between nodes. As with other Maya techniques introduced in this chapter, attribute connection will be eventually be handled with MEL commands. To prepare for the next section, save the current file if you like, and then start a new one.

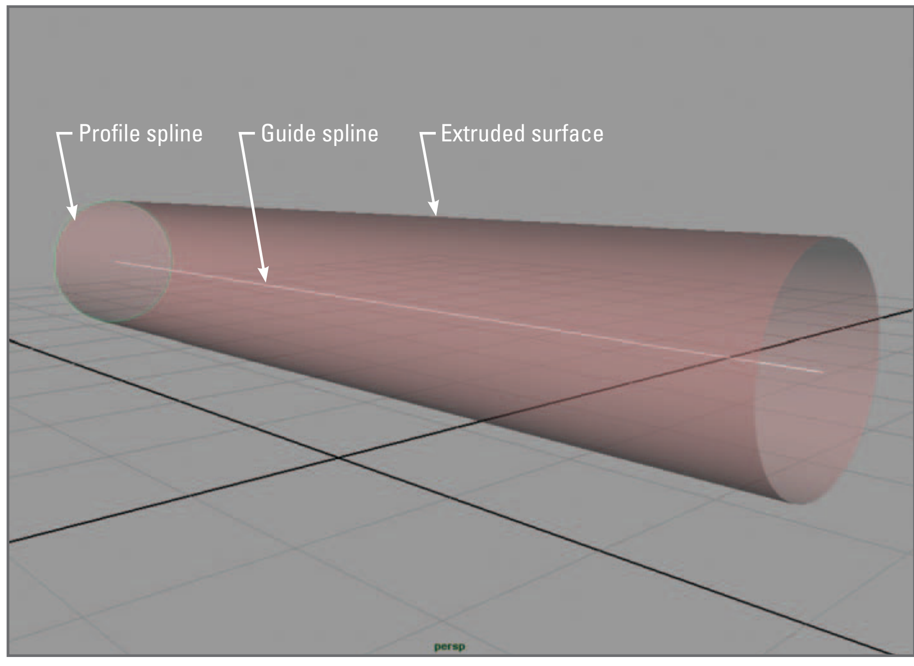


FIGURE 05.23

A NURBS surface extruded from a profile spline along a guide spline.

Tutorial 05.05: Create a NURBS "fiber"

It is common in NURBS modeling to create a specific shape, called a **profile**, with a spline and then use it to generate a surface. In this exercise, you will extrude a NURBS surface, a tube, from a profile spline, along a guide spline (see Figure 05.23). This technique has implications for modeling tubular and fibrous structures—both of which are plentiful in biology. Tubular structures conduct fluid (lymph and blood, for example) and manage forces (long bones such as the femur and humerus). Likewise, fibers fill many roles in the tissues of living things: from axons that conduct nerve impulses to the rigorously aligned collagen bundles that compose tendons. The techniques you'll explore in this tutorial will apply directly to a method for modeling a fiber matrix to simulate dense connective tissue in *Part 3: Chapter 17*.

Set up the scene view

When working in Maya, you have the option to constrain objects and their components to the grid that appears in the scene views. This is called "snapping" and is common to many graphics applications. Turning on "Snap to grids" makes it easy to create the straight line spline you want for the axis of your tube.

1. **Set up a Four panel view in the workspace.**
2. **In the Front view, make sure the grid is showing. If it isn't, turn it on by choosing Show → Grid in the Panel menu set.**
3. **Set the Grid size and subdivisions:**
   (a) **Choose Display → Grid ❐. This opens Grid options.**

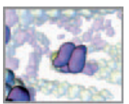


FIGURE 05.24

A spline drawn with the CV Curve Tool. It was drawn in the Front orthographic view with grid snapping turned on. To help you tell where a curve starts and ends, Maya uses a box (or square) to indicate the first CV, and a "U" to indicate the second.

(b) **Length and Width: 10 units**

(c) **Grid Lines Every: 1 units**

(d) **Subdivisions: 1**

(e) **Hit Apply and Close.**

You can also snap to CVs, curves, and view planes. The buttons for these actions are located next to the Snap to Grid button.

4. **Press the Snap to grids button, in the Status Line (toolbar) at the top of the main window.**

5. **Make sure History is turned on: the History icon, , must be depressed.**

   You will take advantage of History further down, when you alter the NURBS surface.

Draw the guide spline

1. **Click in the Front view to make it active.**
2. **Choose Create → CV Curve Tool ❐.**
3. **Hit Reset Tool for the default settings.**

   A Curve Degree setting of Cubic produces smooth splines, while Linear produces angular splines. A cubic value of 3 is sufficient for this exercise.

Curve degree is a measure of how many bends a curve can have between EPs. For most applications, a 3-degree (cubic) curve is sufficient. A minimum of four points (degree + 1) is required to make a cubic CV curve.

4. **Hit Close.**

   Note: you can work with the Tool Settings window open or closed.

5. **Draw a curve in the Front view:**

   LMB+click in at least four (4) different spots to create the spline, as shown in Figure 05.24**.**

   Take advantage of the Snap to Grid feature to get the points in a straight line. You can undo a point immediately after creating it by hitting the Undo hotkey,**z**.

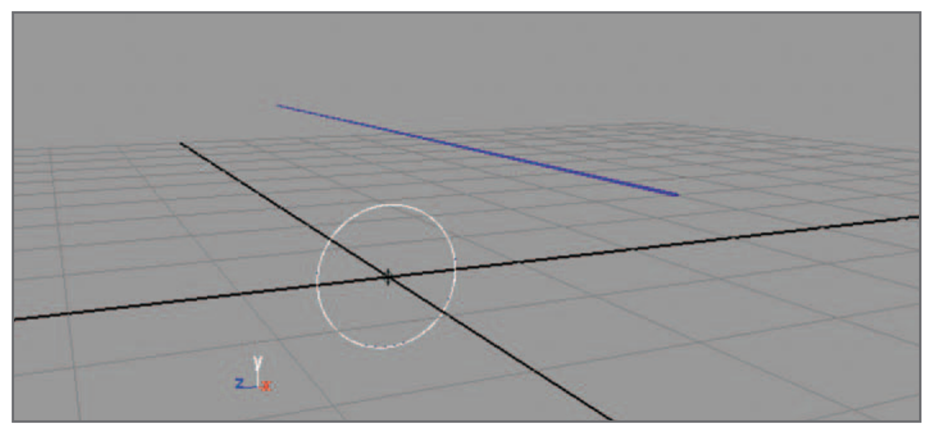


FIGURE 05.25

The guide spline was drawn at Y = 1 in the X–Y plane (Front view), whereas the profile spline (circle) was, by default, created at the world origin.

6. **Once you've created the three points to your satisfaction, hit Enter to complete the spline.**

7. **Hit the Q hotkey to turn the Select Tool on and the Curve Tool off. If you leave the Curve Tool on, you will create a curve point each time you click in the workspace.**

By default, when you draw a spline in the Front view, it is created in an XY plane at Z = 0. If you toggle the Perspective view around the curve you just made, you can see that all of its points lie on a plane perpendicular to, and at the origin of the Z-axis. Had you drawn the curve in the Perspective view, you would have gotten unpredictable results.

You can also create curves using the EP Curve Tool or the Pencil Curve Tool. With the former, you place EPs, rather than CVs, which the resulting curve will pass through. The Pencil Curve Tool allows you to draw a curve freehand.

Using the CV Curve Tool

Maya Help → Using Maya → General → Basics → Basic Menus → Create → CV Curve Tool

Create the profile spline

1. **Choose Create → NURBS Primitives → Circle ❐.**

2. **In the Options window, select Edit → Reset Settings.**

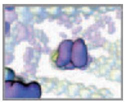
3. **Customize the settings:**

   (a) **Normal Axis: X**

   (b) **Radius: 0.5**

   Note: Normal Axis determines which way the circle will face in 3D space; in this case, you want it to face in the X-direction for extrusion along the straight line curve.

4. **Hit Create.** Figure 05.25 **shows the result in the perspective view.**



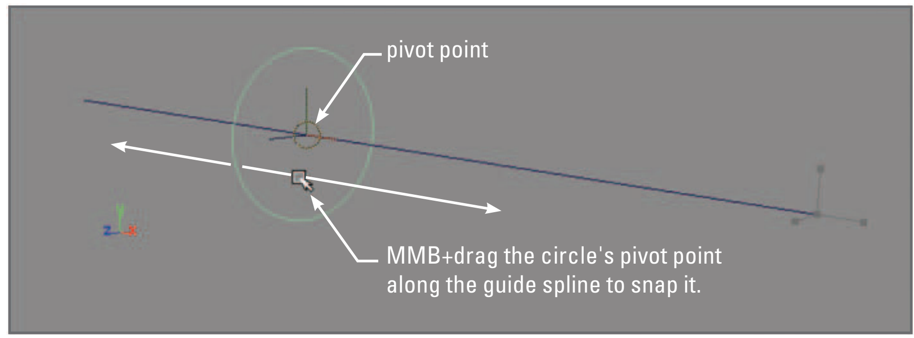


FIGURE 05.26

Objects can be made to snap to one another, to Grid points, or to view planes. In this figure we have snapped the profile spline (circle) onto the guide spline. This allows us to align the circle's plane perpendicular to the spline, and center it on the spline.

Snap the profile to the guide spline

Whereas you created the circle at the origin, you drew the guide spline above the X–Z plane so that it will show up better in the illustration (and not be blocked by the Grid axes). It's important to note that the guide and profile splines need not be aligned to create a useful surface; the Extrude tool options determine location and orientation of the resulting surface. However, if you wish to alter the surface through its History by moving or deforming either of the two splines, their locations relative to one another do matter. Here, you'll snap the profile to the guide spline, so you can deform the surface predictably a little further on.

Snapping to a curve can be tricky—patience helps. The key is to get the objects aligned visually before MMB+dragging to snap.

1. **Press the Snap to curves button, in the Status Line at the top of the Main Window.**
2. **Hit the hotkey "w" for the Move Tool.**
3. **In the perspective view, select the circle and LMB+drag its pivot point (center) so that it overlaps the guide spline.**
4. **MMB+drag the circle's pivot point along the guide spline until it snaps onto it. You'll know it has snapped when you can drag the mouse all over the workspace but the circle remains confined to the guide spline (**Figure 05.26**).**

To snap an object to a curve without using the Status Line snap buttons, hold down the hotkey "c" then MMB-drag the object over top of the curve until it snaps to it.

Extrude the surface

The Extrude tool requires that you select, in order, a profile spline and then a guide spline.

1. **Press F4 to activate the Surfaces menu set.**
2. **Choose Surfaces → Extrude ❐.**
3. **Choose Edit → Reset Settings, then customize the Extrude options:**
4. **Set Result Position to At Path.** This aligns the new surface with the guide path. The remaining attributes are okay at their default values.
5. **Hit "Extrude".**

This creates a tubular surface aligned with the guide spline. If you select it, you'll see that its pivot is located at the world origin, not the object's origin. You can rectify this by centering the pivot:

Choose Modify → Center Pivot.

Alter the tube through history connections

Try moving the guide spline, the circle, and the tube individually. You'll notice that moving either of the guide or circle splines moves the tube. This is because the tube is linked to both splines through construction history. You can break these links if you like by deleting History (**Edit → Delete by Type → History**). For now, leave history intact so you can see how it affects the surface when you transform it or its constituent splines:

1. **Select the guide *or* the circle spline with the tube, and move them together.**

For every unit the spline moves, the surface moves twice as far. This is known as a **double transformation**. Through History, the tube's transform node is dependent on the spline's transform node. However, the tube's position also depends on its own transform node. Therefore, when you move both the spline and the tube, you're effectively moving the tube twice as much. It's important to be aware of situations that result in double transformations since they can be the cause of unpredictable results in animations.

2. **Return the spline and tube to their previous positions. You can do this by hitting the undo hotkey, z.**

Now you'll take advantage of Construction History to alter the shape of the tube.

1. **Select the circle in the Outliner or the workspace.**
2. **In the Channel Box, under INPUTS, click on makeNurbCircle1. This is the creation node.**
3. **Enter 2 in the Radius field.**

Notice that the tube "inherits" the change in radius and becomes wider. Next, deform the guide spline by moving a CV. Before starting, reset the radius to 0.5.

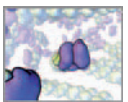
1. **RMB+click on the guide spline and choose "CV" from the Marking menu.**
2. **LMB+click on one of the CVs to select it, then drag it away from the curve as shown in** Figure 05.27a.

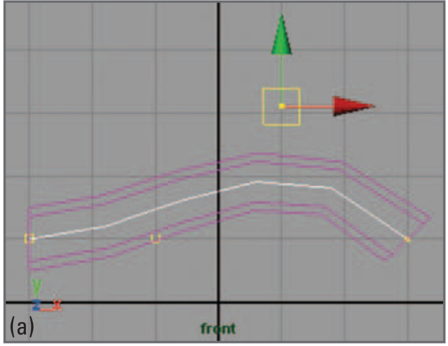
or **in the Channel Box, Click on "CVs (click to show)" and change the Y-value for the selected CV.**

The tube will distort as you drag the CV because of the connection History connection between the curve and the tube.

Rebuild the curve

The angular appearance of the tube is due to the number of points (CVs) you used to create the guide spline; the more points, the smoother the curve. Next you'll rebuild the guide spline to increase the tube smoothness.





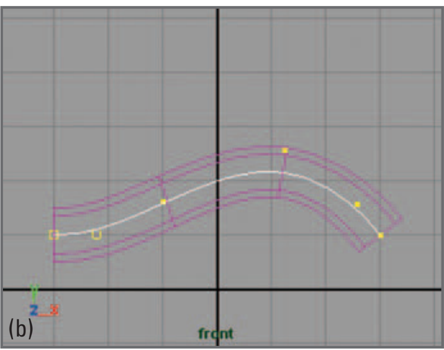


FIGURE 05.27

(a) Through Construction History, moving points on the guide curve deforms the NURBS surface. (b) Rebuilding the guide spline by adding spans, in turn makes the extruded NURBS surface smoother.

1. **In the Outliner, select the guide spline, curve1; this automatically selects it in object mode.**
2. **Choose Edit Curves → Rebuild Curve ❐.**
3. **In the Options window, choose Edit → Reset Settings.**
4. **Set Number of Spans to 2.**
5. **Hit Rebuild.**

A span is a length of curve between two EPs. Your original drawn spline had only one span. By adding another one, you improved its curvature and that of the tube (Figure 05.27b). You haven't seen the last of Rebuild Curves. You will make use of it in MEL command form to smooth out the curves in *Chapter 17*. Save your file if you like. The finished scene file is included on the CD:

05_Modeling/scenes/tutorial_05_05_done.ma

Summary

This chapter introduced NURBS and polygon modeling in Maya. The tutorials provided examples of how to create primitive sphere models, which will form much of the basic geometry for the projects in *Part 3*. You learned that an object can be transformed interactively with the Move, Rotate, and Scale Tools and numerically by changing attributes in the Channel Box. Furthermore, NURBS and polygon are made of different components, which can be individually selected and transformed with the same tools used to move, rotate, and scale whole objects. Maya Deformers, which are used extensively in character modeling and animation (and are therefore a classic animation subject treated deeply by others), work on a fundamental level, by transforming object components.

Your inspection of the Hypergraph revealed the nodes and connections that make up a primitive sphere, plus those that were added to tweak and smooth the surface. Those same nodes appeared in the Attribute Editor, which you used to interact with the sphere's construction history by changing attribute values in the creation node, polySphere1. Altering these attributes affected the downstream shape and tweak nodes, and therefore the appearance of the object—its radius, subdivisions, and the location of surface deformations. Since the polySmoothFace node lay downstream

from the tweak node, changing its Divisions attribute had no effect on the surface deformations. As long as construction history is maintained for an item, you can edit its creation attributes. History can be deleted by deleting individual nodes, or disconnected by deleting connections in the Hypergraph, as you did with pSphere1.

Attributes can be connected to each other in different ways. Connections are made automatically when you create a new object, camera, or light in Maya. They can also be made manually in several ways. In this chapter you used the Connection Editor to reconnect the pSphere1 to its history node. In later Case Studies, you will connect attributes using MEL commands.

Finally, you learned how to create a surface from splines. This is the foundation of NURBS surface modeling, and a simple technique that you will call upon to help build a complex fiber environment in silico.

References

1. Kuhn R, Zhang W, Rossmann M, Pletnev S, Corver J, Lenches E, Jones C, Mukhopadhyay S, Chipman P, Strauss E: Structure of dengue virus: Implications for flavivirus organization, maturation, and fusion. *Cell* 108(5): 717–725, 2002.
2. Petterson EF, Goddard TD, Huang CC, Couch GS, Greenblatt DM, Meng EC, Ferrin TE: UCSF Chimera – A visualization system for exploratory research and analysis. *J. computational Chemistry* 25: 1605–1612, 2004.

This page intentionally left blank

06 Animation

Introduction

In the previous chapter you learned about model types in Maya (NURBS and polygons) and how to create primitive objects. You saw that an object is composed of nodes, each with a number of attributes. Using the transform tools and the Channel Box, you changed certain attribute values in order to move, scale, rotate, and change the shape of objects. In this chapter, you will record such changes at different times—called **keyframes**—on the timeline in order to create animation.

The relevant Maya windows, menus, and tools will be introduced in the upcoming tutorials. You will use the Hypergraph to see what happens behind the scenes when attributes are animated and examine how animation is stored within a node. By the end of this chapter you will have learned how to animate a simple object using both manual keyframing techniques and automated **procedural** methods.

As in the chapter on modeling, our goal here is to provide an introduction to concepts and techniques that will serve you well in the projects in *Part 3* of this book. This chapter is by no means exhaustive; as with modeling, one could easily fill a book or book series with techniques, tips, and tricks on animation with Maya. Under the heading, *Learning Maya*, the *Further reading* section lists helpful resources for exploring this topic further.

Animation

In Maya, animation is simply the change over time in the value of an attribute. An attribute that can be animated is said to be *keyable*, in reference to the word **keyframe** (also *key*), which is both a noun and a verb in Maya. As a noun, it means a *frame*, or unit of animation time, at which a value has been recorded, or *set*. As a verb, it refers to the action of setting the value.

While keyframes are a means of recording animation for playback, they are not necessarily a requirement of animation; all that is required is for an attribute to change with time. Keyframes are merely a convenient way to store attribute values at different times, within a Maya file. Alternately, values may be stored in an external file, or they may not need to be stored at all.

Maya store keyframe data for every animated attribute in a separate animation node. The input for this node is time and its output is an attribute value. This data is in the graphical form of attribute versus time is visualized in the Graph Editor, a handy Animation Editor you'll meet shortly. Like other nodes in Maya, you can inspect and edit an animation node's values in the Attribute Editor.

Procedural versus keyframe animation

Generally speaking, **Procedural animation** refers to the use of computer procedures, or algorithms, to change attributes over time. The procedure can be as complicated as an algorithm for DNA replication or as simple as an instruction to make an attribute equal to a constant value; the point is that it uses an instruction or set of instructions, not a recorded value, to determine the attribute value.

The following example illustrates the difference between keyframe and procedural animation. Figure 06.01 shows a cube that moves along the X-axis from point X_A at time T_A, to point X_B at time T_B. The animation here is the change in the Translate

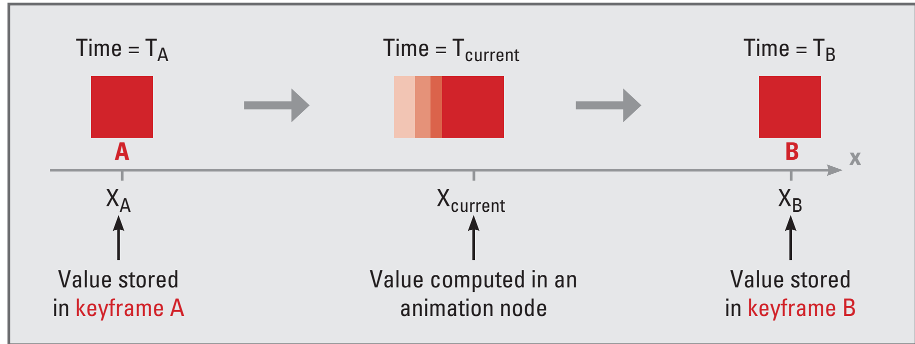


FIGURE 06.01
Animation of a cube's Translate X attribute between keyframes A and B.

X attribute over time. If this action were keyframed, Maya would use the current time, $T_{current}$, to look up $X_{current}$ in the animation node. If instead the action were procedural, Maya would calculate the $X_{current}$ value using a source other than an animation node. This source could be an **animation expression** node, a MEL script, or another type of node, such as a **procedural texture** which calculates its output value using an internal algorithm. We will examine the nodes and connections used in animation through specific examples in the tutorials later in this chapter.

Keyframe and procedural animation need not be exclusive from one another. Keyframes are often used to record the outcome of procedural animation for later playback. Nor is procedural animation limited to the physical properties of objects. Procedural textures are texture nodes that use mathematical procedures to create interesting patterns for shading objects. Similarly, a light can have its attributes animated procedurally to produce interesting effects.

Keyframes and memory

Each time you set a keyframe, the relevant information is stored in RAM until you save the Maya file, at which time it is written into the file. The more keyframes you set, the larger your file. In a simulation using procedural animation, where you want to record animation for many objects (e.g. interacting molecules) over many time increments, setting keys can eat up RAM and drive your computer to use **virtual memory**, with its associated time penalties. In *Chapter 13*, you will learn how to read and write attribute values to and from a text file. You can use these techniques to record animated attribute values to an external file rather than keyframing them in Maya. Such an approach keeps RAM use down, and Maya file size to a minimum.

Virtual memory refers to the practice of using a hard drive for storage and retrieval of data once RAM becomes full. Helpful for alleviating low-memory situations, it comes with a heavy time penalty—about an order of magnitude slower than RAM.

The Animation menu set

In addition to tools for keyframing, this menu set provides access to ones used to deform and rig objects. Rigging, which is widely used in character animation, is the practice of endowing a model with attributes that deform its shape in a controlled manner. A common example is the rigging of a character, the “skin”, with a jointed skeleton. Joints are then rotated, deforming the skin to bend limbs. A skeleton’s joints can be animated to make a character walk and talk. Deformers work like joints, changing an object’s shape by moving its components.

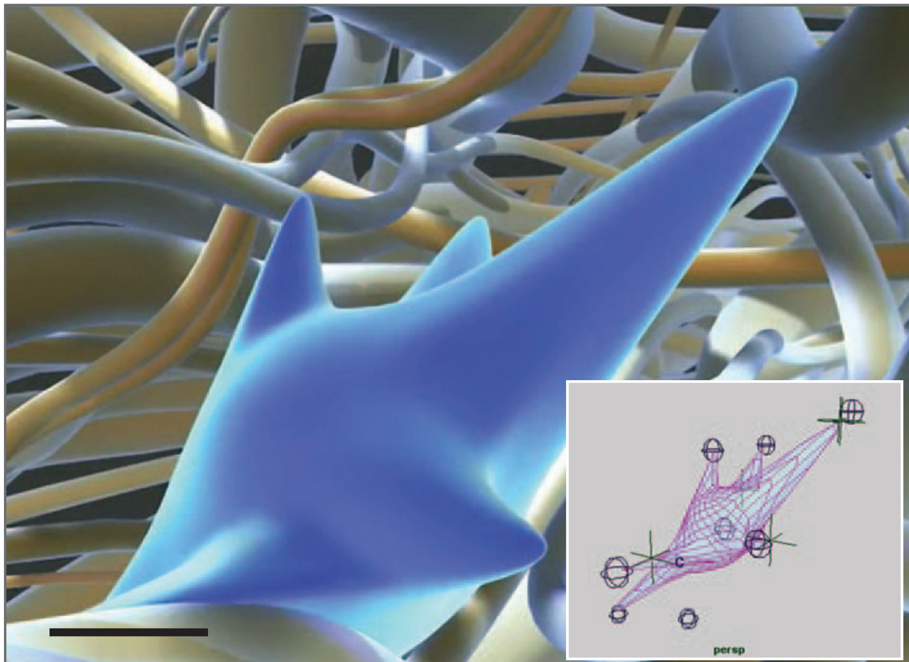


FIGURE 06.02

In this animation of a migrating fibroblast cell, the cell body is rigged to extend and retract appendages called pseudopodia. The rigging uses joints and deformers (inset) to deform the cell surface smoothly as it crawls through its environment. Scale bar ≈10 μm.

For our in silico work, we have for some projects used rigging techniques to deform motile cells as they locomote through scaffolds (Figure 06.02). You'll use some of these techniques when you animate a crawling cell in *Chapter 16*. Another example is the heart model shown in Figure 05.01 in the previous chapter, which was rigged to beat at regular intervals. Nonetheless, since the Projects in *Part 3* don't require rigging and deformations, we will leave their discussion to the resources listed in the *Further reading* section, and focus here on the menu items concerned with keyframing. To activate the Animation menu set, use the Status Line pull-down menu at the far left of the Status Line.

Setting keys

Below are a few ways to set keys for, or *keyframe*, an attribute.

Using the Channel Box (Figure 06.03**)**

1. **Select the item (object, camera, etc.) for which you want to key an attribute.**
2. **Select the attribute(s) you wish to key, by name in the Channel Box.**
3. **RMB + click over a selected attribute name. This brings up a context menu.**
4. **Choose Key Selected and release the RMB. This will set a key for all selected attributes.**

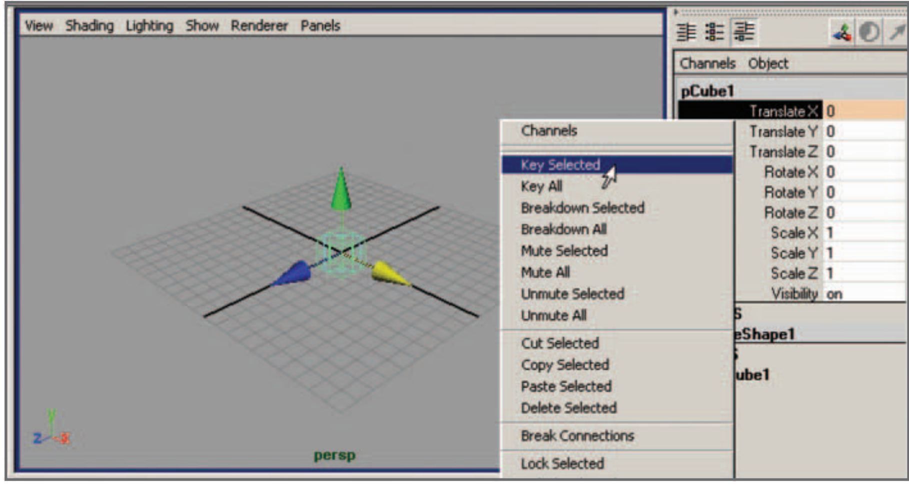


FIGURE 06.03

Using the Channel Box to set a keyframe for the Translate X attribute.

Hotkey	Action
S	Set keys for all transform attributes
Shift+W	Set keys for Translate X, Y, and Z
Shift+E	Set keys for Rotation X, Y, and Z
Shift+R	Set keys for Scale X, Y, and Z

TABLE 06.01

Hotkeys to set keyframes for selected objects.

In the Attribute Editor:

1. Select the item for which you want to key an attribute.
2. Open the Attribute Editor (Ctrl+A).
3. RMB+click on the attribute name to key all attributes corresponding to the name (e.g. clicking on **Translate** allows you to set all three (X, Y, and Z) Translate values).

   or

   RMB+click on the attribute field to key only a single attribute name (e.g. Translate X).
4. In the context menu, select "Set Key".

Using a hotkey:

1. Select the item for which you want to key an attribute.
2. Use one of the hotkeys shown in Table 06.01 to set a keyframe.

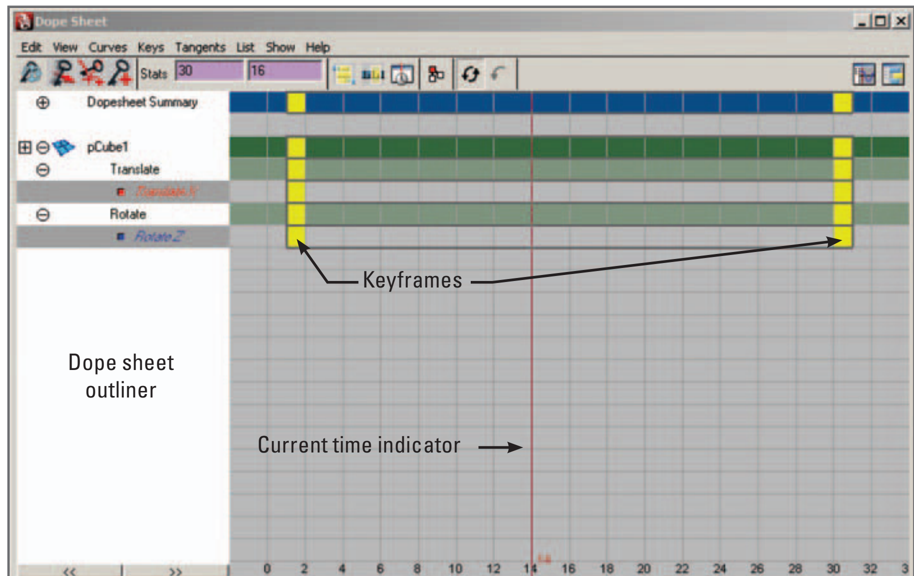


FIGURE 06.04

The Dope Sheet provides a tabular view of keyframes for a selected object.

Auto keyframe

When you turn on **auto keyframing**, Maya automatically sets a key every time you change an attribute. For example, if you were to key the position of an object at time 1, move the time indicator ahead to time 2, then drag the object to a new position, Maya would set a key for the new position at time 2. This can certainly speed workflow in some circumstances, but it can also be dangerous because it can lead to setting keys accidentally. This can ruin a carefully arranged animation. To turn auto keyframing on:

1. **Choose Window → Settings/Preferences → Preferences. Select Settings → Animation.**
2. **Under Auto Key, check Auto Key (uncheck to turn off auto keyframing)**

   or

   Press the auto keyframe icon in the bottom right corner of the UI.

In *Chapter 18*, you'll use the MEL command `setKeyframe` to record animation.

Graphing animation

Maya represents animation graphically in two ways, using the **Dope Sheet** and the **Graph Editor**. The Dope Sheet provides a tabular account of keyframes for a selected item (Figure 06.04). You can use it to edit animation by selecting and moving keys along

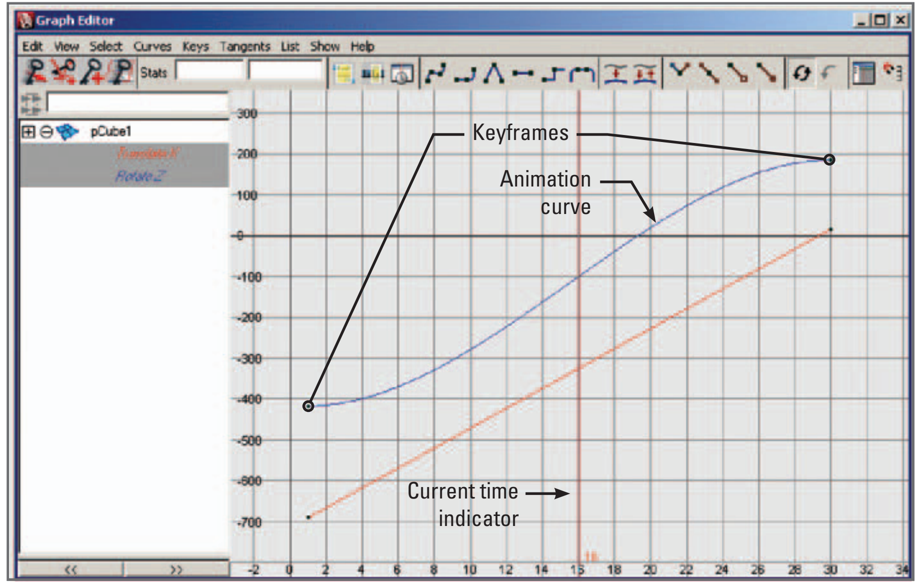


FIGURE 06.05

The Graph Editor. The blue curve, which represents the Rotate Z attribute of pCube1, uses spline interpolation. Its slope approaches zero at either end, making for a gradual increase and decrease in the rate of change of Rotate Z. The red curve represents the Translate X attribute and uses linear interpolation. Its slope is constant, which makes for an instantaneous increase and decrease in the rate of change of Translate X.

the timeline. In a traditional animation workflow, the Dope Sheet is used to coordinate event and sound synchronization and timing.

The Graph Editor (Figure 06.05) is a 2D graph displaying **animation curves** (also called **keysets**). These represent attribute values plotted on the vertical axis against time values plotted on the horizontal axis; they are the **in-between** animation spanning the keys, which are the plot points. Their interpolation through and between keys determines the smoothness of animation. A curve can use **linear** or **spline** (nonlinear) **interpolation**, as shown in Figure 06.05. Spline curves correspond to smooth acceleration and deceleration in animation. In other software applications spline interpolation is sometimes called *ease in* and *ease out* (*in*to and *out* of a keyframe). This is the opposite of linear interpolation which makes for abrupt changes—instantaneous acceleration—in attribute values at keyframes. This is not to say that spline interpolation can't generate abrupt changes in direction.

In addition to displaying animation curves, the Graph Editor contains tools for adjusting interpolation and for moving and scaling keys. For keyframe animation, we find it to be one of the most useful tools in Maya. You will explore it further, along with animation curve interpolation, in the upcoming tutorial.

Dope Sheet and Graph Editor

Maya Help → Using Maya → Animation, Character Setup, and Deformers → Animation → Animation Windows and Editors → Editors
→ Dope Sheet
→ Graph Editor

Deleting keys

Once keys are set, they can be deleted in one of the following ways:

Use the Dope Sheet or the Hypergraph to delete one or more keys for the selected object(s):

Select the keys in the Dope Sheet and hit Delete.

or **Select the keys in the Hypergraph and hit Delete.**

Use the Timeline to delete all keys for the selected object(s) at a specific frame:

1. **Select the red key tick mark in the Timeline.**
2. **RMB+click on the key and select Delete.**

Use the Channel Box or Attribute Editor to delete all keys for the selected attribute(s):

1. **Select the attribute in the Channel Box or Attribute Editor.**
2. **RMB+click on the attribute and select Break Connections.**

Time units

In *Chapter 04*, we introduced the Timeline and Playback controls, which you will use to scrub and to play your animation. In addition, Maya lets you determine the working units for time, which are set to 24 frames per second (fps) by default. You can choose from a variety of fps settings, including broadcast standards **PAL** and **NTSC**, and clock settings: hours, minutes, seconds, and milliseconds. A clock setting of "milliseconds" is equivalent to an fps speed of 1,000. A setting of seconds is equivalent to one fps, and so on for minutes and hours. To access the Time settings:

1. **Open the Preferences Window and choose Settings.**
2. **Under Working Units → Time, select an appropriate playback frame rate. (In North America, it is common to use NTSC (30 fps) which is a television video broadcast standard.)**

It is good practice to set your Time working units at the start of a project. Changing units midway will shuffle keyframes along the timeline unless you have **Keep Keys at Current Frames** checked in the Working Units settings.

The Time working units are as important as is the speed of the action in a rendered movie. For instance, if an action were to occur in one second as seen by an audience, then you would set the working units in Maya to match those of the viewing technology. Suppose, for example, you were animating the **cell cycle** for a European television audience, and you had one second to show the **cytokinesis** phase. In Maya, you would set the Time working units to PAL (25 fps) (the European television broadcast standard) and animate cytokinesis within 25 frames.

For in silico simulations, the working units are generally flexible. Our practice is to use NTSC (30 fps) because we often output movies to video for a North American audience.

Playback settings

After setting the working units, you can further specify speed of playback in the scene view. This can be the speed you set in Time working units, half or twice that speed, or a different frame rate altogether. Since this determines only how quickly Maya plays frames in the workspace, it does not affect the per-second rate of animation in your scene.

It is not uncommon for a scene to be too complex to play back at the specified fps. In this case, Maya skips frames to keep pace. This is generally fine for keyframed animation, but is a major pitfall for scenes involving dynamics or procedural animation; calculations are missed in the skipped frames, leading to bogus animation results. To prevent this, Maya must be set to play every frame, independent of a desired frame rate:

1. **Open the Preferences Window and choose Settings → Timeline.**
2. **Under Playback, select Play every frame.**

Within the Playback settings, Update View determines if Maya will redraw all windows in the workspace during playback or just the active one, which requires less memory and processing. You can also choose whether animation is to play once or loop. Finally, you can skip frames for quicker playback of complex scenes by setting Playback by to a number other than 1; for instances a setting of 3 means Maya will play every third frame. For scenes involving dynamics and procedural animation, this setting should always remain at 1, for the reason mentioned in the previous paragraph.

Tutorial 06.01: A keyframe animation

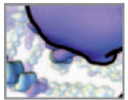
This is a quick exercise to become familiar with setting time-dependent attribute values using keyframes. You will animate the Translate X value for a primitive cube. We're starting with this very simple example in order to demonstrate the core concept of animation in Maya: the change of an attribute's value over time. Armed with this understanding and the fact that almost all attributes for all the nodes in a Maya scene can be animated, you will have at your fingertips enormous creative potential for simulations and visualizations of complex biological phenomena. Let's get started.

Preparation

To start, create a polygon cube, then select it in the scene view. Next, you'll set the Time working units, the Playback settings and the duration of the animation.

1. **Choose Create → Polygon Primitives → Cube.** If Interactive Creation is turned on in this menu, you'll need to click and drag in the workspace to make the cube.
2. **Choose Window → Settings/Preferences → Preferences. Make the following settings:**

   (a) Settings → Time → Working Units to NTSC (30 fps).



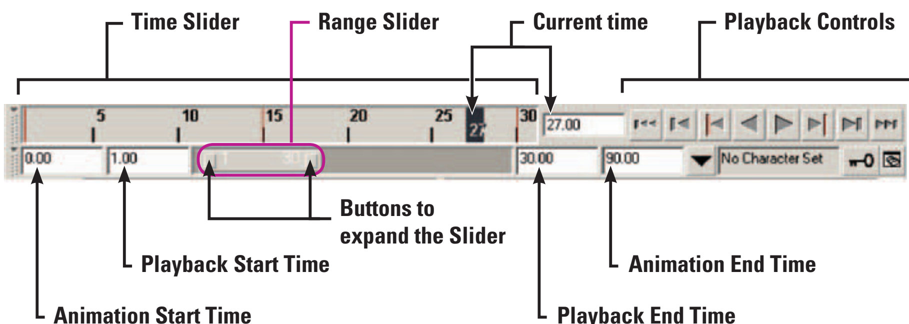


FIGURE 06.06
The Animation Controls.

Setting Playback to **Real-time** will give you three seconds of animation for 90 frames at 30 fps. This will make the animation easier to watch than would a setting of **Play every frame**. The latter would play back very quickly because this simple animation requires very little computer horsepower to compute and draw each frame.

(b) **Settings → Animation → Tangents → Default In Tangent and Default Out Tangent both to Linear and check Weighted Tangents.**

(c) **Settings → Timeline → Playback to Real-time (30 fps), and Looping to oscillate** (so that the animation will play back and forth continually until you press Stop in the Playback Controls).

Hit Save to close the Preferences Window.

3. **Set the current time to 1.0 by LMB+clicking on 1 in the Time Slider, or by entering 1 in the Current Time field.**
4. **Set the Start and End times in the Range Slider to 1.0 and 90.0, respectively** (Figure 06.06)**.**

In Step 1b you set the animation tangents to be linear. We'll explain why shortly.

Set the keyframes

Now use the Channel Box to record a key for the cube at frame 1.

When an attribute has been keyed, its value field in the Channel Box turns from white to orange.

1. **Select the cube.**
2. **Select the Translate X (not its value field) in the Channel Box.**
3. **RMB+click over Translate X to bring up a context menu.**
4. **Choose Key Selected and release the RMB.**
5. **Repeat steps 2 through 4 for Rotate Z.**

You have just keyed the cubes position at (0, 0, 0) and its rotation at (0, 0, 0). Next you will change the time, translate and rotate the cube, and set a new key.

1. **Enter 90 in the Current Time field or click on 90 in the Timeline to move to frame 90.**

2. **Hit the W hotkey to activate the Move Tool and drag the cube with the X handle to X = 16.**

or **Enter 16 in the Translate X field in the Channel Box.**

3. **Set another key for Translate X at this new time.**
4. **Enter -720 in the Rotate Z field in the Channel Box.**
5. **Set another key for Rotate Z.**

You can also key translate values for individual *CVs* in the Channel Box, in the same way you would select and key transform attributes. The resulting animation curve nodes will be connected to the object's shape node, rather than its transform node.

Play, scrub, and stop the animation

Note the red key ticks in the Time Slider which indicate keyframes. To play your animation:

In Preferences, you can change the display size of key ticks in the Time Slider.

1. **In the Animation Controls, press the Go To Start button to return to the beginning of the playback range.**
2. **To play the animation, press the Play button in the Animation Controls.**

or **Hit the hotkey, Option+V.**

You should see the cube roll back and forth across the scene view (Figure 06.07). To scrub the animation:

LMB+drag in the Time Slider.

To stop playback:

Press the Stop button in the Animation Controls.

or **Hit the hotkey, Option+V.**

or **Hit the ESC key.**

Edit the animation curves

Notice that during playback the reversal of motion at either end of the cube's trajectory appears abrupt; it appears to instantaneously change direction. This is the result of setting the keyframe interpolation to *linear* in the Animation Tangents Preferences. It's often favorable to have linear versus nonlinear changes in motion when starting to animate, in order to rough in the motion. You can then refine the motion by adjusting the keyframe tangents in the Graph Editor, which is precisely what you will do in a few pages.

The Graph editor is a good item to add your custom shelf.

When using procedural animation for in silico biology, we often set keys to record the action for later playback in a movie file. By adjusting the interpolation of the resulting animation curves you can smooth out the motion and make for a more watchable movie in the end. Let's explore the Graph Editor and use it to change the interpolation so that the cube eases into and out of motion.

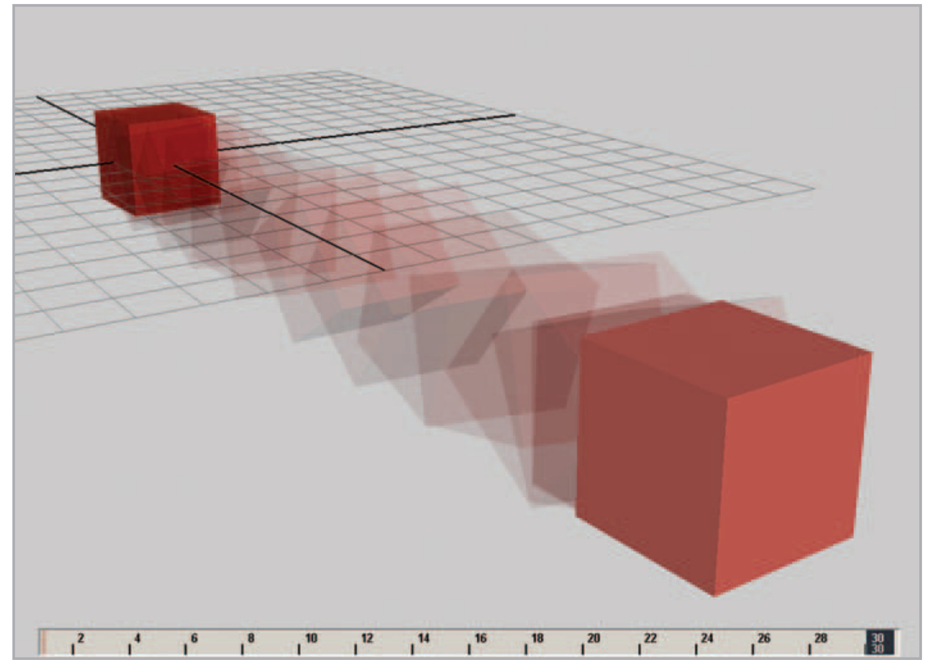


FIGURE 06.07

With its translate and rotate attributes keyed and playback set to Looping: Oscillate, the cube will roll back and forth in the scene view.

1. **Select the cube.**
2. **Open the Graph Editor: choose Window → Animation Editors → Graph Editor.**
3. **In the Graph Editor, choose View → Frame All.**

or **Hit the hotkey, A to frame all animation curves for the selected object.**

You should see something resembling Figure 06.08. Note that the curves are color-coordinated with their corresponding attributes.

The Graph Editor outliner

This panel displays only selected items. Under each one is listed its animation curves (corresponding to its keyed attributes) by name. The curves for **Translate X** and **Rotate Z** appear under the transform node, pCube1. None of the other nodes that make up pCube1 appear here because you did not key their attributes.

The Graph Editor graph view

When an attribute is selected in the Graph Editor outliner, its animation curve appears in the graph view. Table 06.02 shows hotkeys and key combinations used to adjust this view and work with keys. When you select a key, **Bézier handles** (or **tangents**) appear. Figure 06.09 shows the common types of animation tangents in Maya. You can MMB+drag a tangent to modify its curve. More control can be gained by weighting the tangents and unlocking their weights.

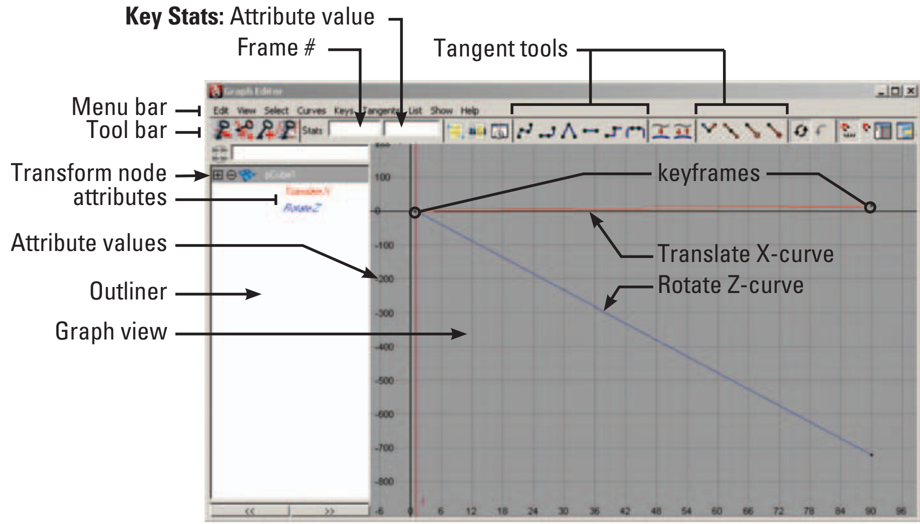


FIGURE 06.08

Features of the Graph Editor. Because of the keyframe settings we used in Preferences, the cube's animation curves are linear—of constant slope—resulting in abrupt changes in direction at the beginning and end of the animation.

Hold	Drag	Hotkey	Function
Alt	MMB		Track view
Alt	LMB+MMB		Dolly view
K	MMB		Move current time indicator
	MMB		Move the selected key
Shift	MMB		Move the selected key, constrained to one of the two axes
		A	View all
		F	Frame selected key(s)
Note: To move keys, you must activate one of the ***Move, Rotate***, or ***Show Manipulator Tools*** in the main ***toolbox***.			

TABLE 06.02

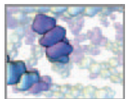
Shortcuts for the Graph Editor graph view and the Dope Sheet.

1. **Select the cube.**
2. **In the Graph Editor, select the choose Curves → Weighted Tangents.**
3. **Press the Free tangent weight button in the toolbar.**
4. **Drag a tangent to distort one of the curves:**

   (a) Hit the hotkey, "W" to active the Move Tool.

   (b) LMB+click on a tangent to select it.

   (c) MMB+drag the tangent.



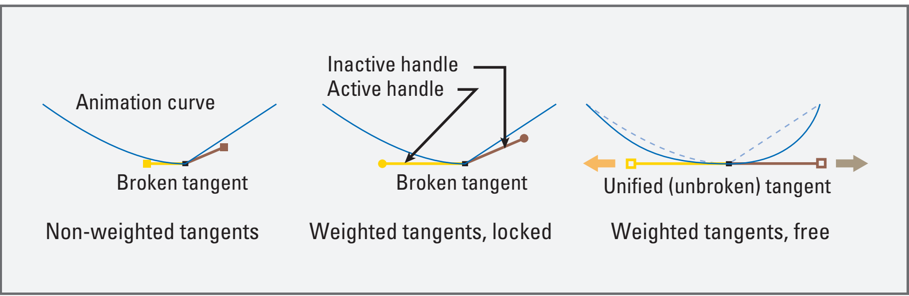


FIGURE 06.09

Different animation curve tangents displayed in the Graph Editor. Free, weighted tangents give you the most control when reshaping an animation curve.

Take a moment to modify the curves using their tangents and play the animation back to observe the effect. Don't hesitate to really distort them and see what happens to the animation—you may have to pull your camera back to see the full range of motion in the scene view. Below, you will use the toolbar buttons to reset the curves and then adjust them automatically.

Editing animation curve tangents

Maya Help → Using Maya → Animation, Character Setup, and Deformers → Animation → Keyframe Animation → Edit Tangents

The Graph Editor toolbar

The most commonly used curve editing tools are accessed through the Graph Editor toolbar buttons; several of these are also found in the menu bar.

1. **Select the cube.**
2. **With one of the Select, Lasso, Move, Scale, or Rotate tools active, select both animation curves in the Graph Editor, the same way you would select an object in the scene view.**
3. **Press the Linear tangents button in the toolbar. This will return the curves to their original linear state.**
4. **Press the Flat tangents button in the toolbar.**

The flat tangents at the start and end of the animation cycle give smooth motion into and out of each translation and rotation direction change. Smooth acceleration is key to natural-looking animations.

Moving keys

In the Graph Editor you can select, move, or delete keys.

1. **Select the cube.**
2. **Hit "W" to activate the Move Tool.**

3. **In the Graph Editor, select the two keyframes at frame 90.**
4. **Hold down the Shift key while you MMB+drag the keys left to frame 30. Then release the mouse.**

or **Enter "30" in the Frame # field at the top-left of the Graph Editor.**

Because you changed only the frame number, and not the attribute values of the selected keys, the cube will now cover the same distance and rotate the same amount in 30 frames as it did previously in 90. Before you hit Play, change the Playback Range to span 1 to 30 frames to match the animation range, or else you'll spend two seconds watching nothing.

Save your scene as you will need it for the next tutorial.

Animation nodes in the Hypergraph and Attribute Editor

In the previous chapter you used the Hypergraph to inspect the nodes that composed a simple object. Here you'll employ it to look at the nodes that were created and/or connected when you keyed the cubes attributes.

1. **Select the cube and open the Hypergraph.**
2. **Choose Graph → Input and Output Connections.**

Figure 06.10 shows the nodes composing a polygon cube called `pCube1`. The transform, history, shape, and shading engine nodes are familiar from the last chapter. The two new nodes connecting to `pCube1` are animation curve nodes that were created when you keyed the translate and rotate attributes.

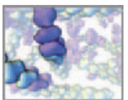
Figure 06.11 shows the Translate X animation curve node represented in the Attribute Editor, with which you can edit key values and interpolations.

Tutorial 06.02: A simple procedural animation

In this exercise, you will add custom procedural animation to the existing keyframe animation on the cube created in the previous tutorial. It will cause the cube to rotate back and forth about its Y-axis, giving the appearance of a wiggle. We're getting slightly ahead of ourselves, as this task will require working with the **Expression Editor** and a touch of MEL script, before either have been formally introduced (which will happen in *Chapter 12*). But it's an exciting taste of what's to come and will demonstrate a strength of the Maya environment: one can make use, quickly and effectively, of procedural animation techniques, involving a host of built-in MEL commands, with little or no prior programming experience.

Animation expressions in brief

An animation expression is an instruction or set of instructions, usually invoked to control keyable attributes, that executes in coordination with Maya's timeline. The instructions work much like an animation curve does, telling an attribute what



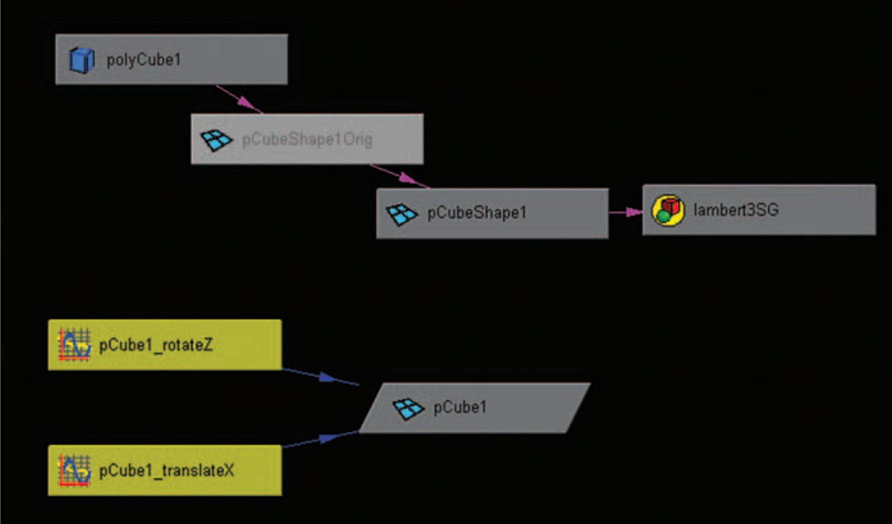


FIGURE 06.10

Maya creates an animation curve node for each attribute you key. Two such nodes are shown here in the Hypergraph for the animated polygon cube. The shape of a node changes from a rectangle to a parallelogram to indicate that it is animated.

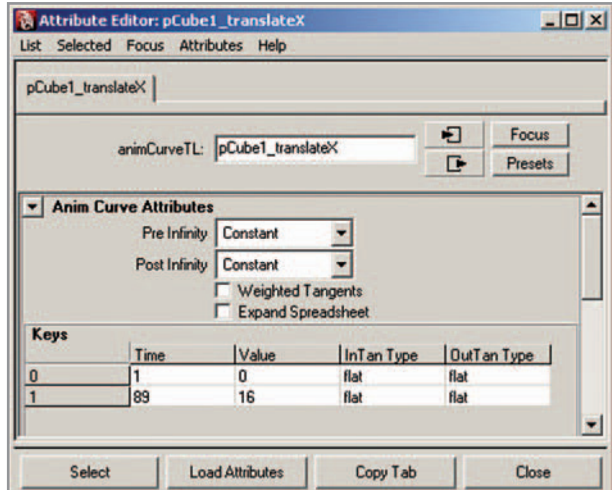


FIGURE 06.11

Animation nodes can be viewed and altered in the Attribute Editor. Here you can edit keyframe times and values as well as set the pre- and post-infinity behavior of the animation curve—that is, how the curve is extended beyond the first and final keyframes.

value to assume at a given frame. Rather than go into a lengthy explanation right now, we'll use an example to show you what an animation expression is and what it can do. There will be much more on this exciting topic beginning in *Chapter 12*, through to the end of this book. To start, open the scene file with the animated cube from the previous tutorial. If you didn't create the file on your own, you can find it on the CD.

06_Animation/scenes/tutorial_06_01_done.ma

FIGURE 06.12

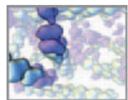
The Expression Editor displaying an expression created to drive the Rotate Y attribute of the polygon cube. An attribute is referred to in an expression as a node name followed by a period (or *dot*), followed by the attribute name.

Create an animation expression

1. **Select the cube.**
2. **Select its Rotate Y attribute in the Channel Box.**
3. **RMB+click on Rotate Y in the Channel Box and select Expressions from the context menu. This will launch the Expression Editor (**Figure 06.12**).**
4. **In the Selected Obj & Attr field, select the text pCube1.rotateY and MMB+drag it into the Expression field below.**

This "dot notation" (`pCube1 "dot" rotateY`) is the standard notation for nodes and their attributes in Maya. Here you will assign the attribute the value of a built-in trigonometric function for the sine of an angle, which is shortened to `sin()` in MEL as in most other programming languages. Trigonometric functions are often advantageous in computer animation because of their cyclic or periodic nature, which can be used to create oscillating motion. You will use `sin()` to give the polygon cube an oscillating rotation about its Y-axis.

Like most functions in Maya, `sin()` requires an argument, which is number on which to operate. The current frame number makes a suitable argument because it increases steadily as the animation plays. It is therefore a good stand-in for the elapsed time itself. As it increases or decreases, `sin()` will oscillate predictably through a range of positive and negative numbers, which will in turn rotate the cube back and forth about Y. The current frame is represented by a **global variable** called `frame`. Variables will be discussed much more in *Chapter 12*, but for now it's enough to know that



`frame` is a value that can be queried by its name anywhere with Maya. Let's complete the animation expression:

1. **In the expression field of the Expression Editor, type = `sin(frame);` to the right of pCube1.rotateY.** Your expression so far will look like:

   ```
   pCube1.rotateY = sin(frame);
   ```

2. **Press the Create button.** This creates a new animation expression.

A semi-colon is used to separate MEL statements. While its use is not strictly necessary when working with a single statement, it is good programming form.

The name of the new animation expression appears in the Expression Editor under the heading, Selection → Expressions. You can edit an expression at any time by selecting it by name in this field. However, by default the field shows only the expression for selected nodes. To show all expressions in a scene,

Choose Select Filter → By Expression Name.

When an animation expression has been assigned to an attribute, the attributes value field turns purple in the Channel Box.

Now play the animation to see the effect on the cube's Y-rotation. You'll notice it's very subtle. To make it more pronounced, increase the amplitude of the `sin()` function.

1. **In the lower Expression field, type:**

   ```
   pCube1.rotateY = 20 * sin(frame)
   ```

2. **Press the Edit button to enter the expression.**

Now when you hit Play, the magnitude of the Y-rotation will be considerably greater (20 times greater, in fact). Equipped with this understanding of a simple, one-line animation expression, you'll see shortly it's not too great a leap to begin programming more complex instructions to drive attributes in much more complicated models of organic structure and function.

The finished scene file for this tutorial is included on the CD:

06_Animation/scenes/tutorial_06_02_done.ma

Animation expression nodes

Before leaving this example, let's take a quick look in the Hypergraph at the nodes that were created (Figure 06.13).

1. **Select the cube and open the Hypergraph.**
2. **Choose Graph → Input and Output Connections.**

There are two new nodes: an animation expression node and a time node. The time node provides the frame number to the animation expression node, which updates the transform node, `pCube1`.

Summary

In this chapter you learned that in Maya, animation boils down to the change over time in the value of an attribute. Any attribute that is "keyable" can be animated.

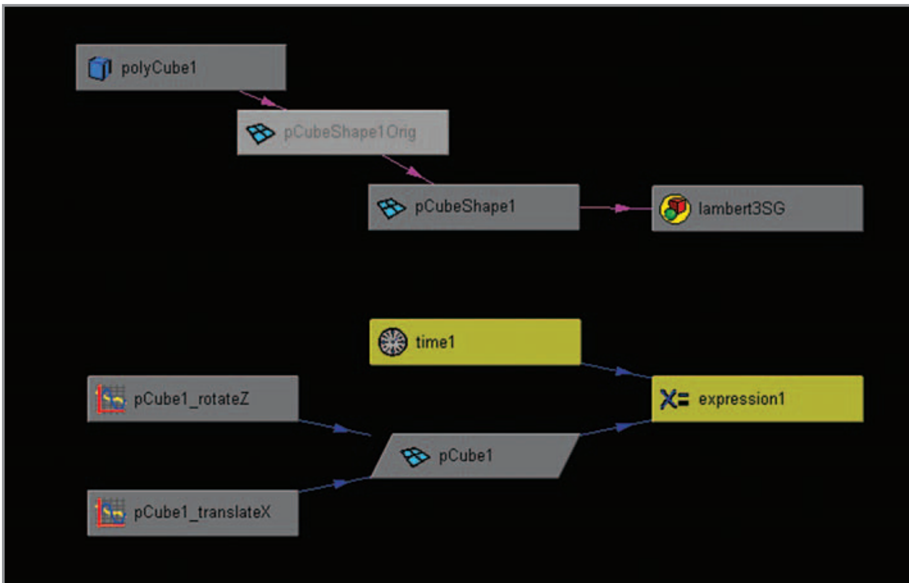


FIGURE 06.13

When you create an expression, you are creating an expression node and a time node, shown here in the Hypergraph.

Furthermore, an attribute can be animated by keyframing or using an animation expression. The latter is a type of procedural animation.

Keyframing creates an animation curve DG node and connects it to the attribute in question. An animation curve spans keyframes and returns a value, at any given frame number, which drives the attribute in question. The shape of a curve, described as its interpolation, is governed by its tangents, which can be manipulated by hand or automatically (i.e. through a procedure). Curve interpolation, in turn, is a measure of the acceleration of animation into and out of keyframes. This acceleration can be gradual or instantaneous and determines the degree to which animation appears smooth or abrupt.

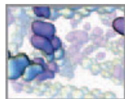
An animation expression is a DG node which contains an instruction or set of instructions that drive the value(s) of one or more attributes. A time node updates the animation expression node at each frame during playback. The expression may be a single line of script, like the example in the last tutorial, or many lines which evaluate equations and use the results to drive attributes, which is the approach we use when simulating of behaviors of molecules and cells.

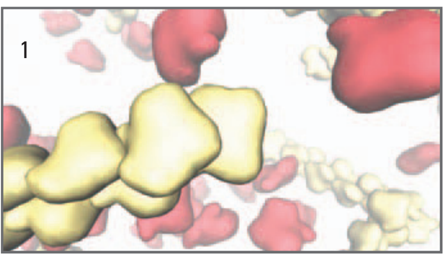
The animation examples in this chapter open the door to powerful concepts that will come into play all through the rest of this book. In silico simulation methods use keyframing as a way of capturing the results of a simulation, rather than creating animation. The animations are thereby created procedurally using animation expressions and another scripting device called a procedure.

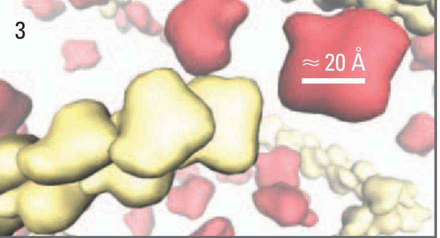
In the next chapter, you will be introduced to Maya Dynamics, which simulates forces acting on and between bodies to create animations. Dynamics is a powerful feature of the Maya environment and is widely used in special effects animation for film and television. For in silico biology, we use dynamics in our projects to emulate random motion and collisions in systems of individual molecules and of whole cells.

This page intentionally left blank

07 Dynamics







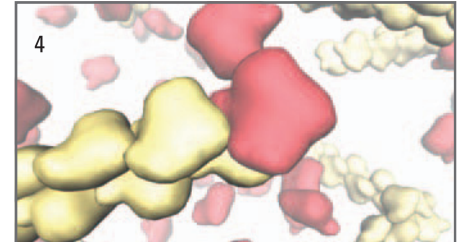
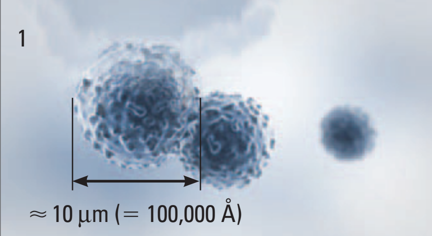
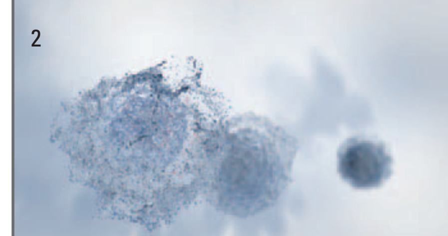


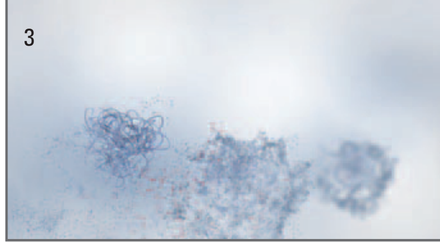
FIGURE 07.01

Four frames from a simulation of polymeric protein (actin filament) assembly. The scale bar (≈20 Å) in frame 3 pertains to the red foreground protein.

This model involved a combination of Maya Dynamics and custom procedural (mathematical) simulation. With very little effort, dynamics was used to get the objects diffusing and colliding so that we could focus our efforts on the mathematics of the chemical reactions between objects as the filaments assembled and disassembled.







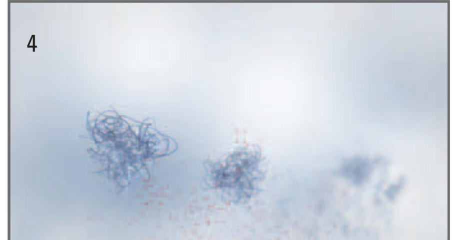


FIGURE 07.02

Four frames from an animation illustrating the use of a detergent to lyse (or dissolve) HIV-infected T-lymphocyte cells (spherical objects). This process is used to collect viral material (RNA (red specks)) for study. The detergent cloud is modeled using Maya 2D Fluids and can be seen entering the top left of panel 1. As it washes over the cells, their membranes disintegrate—a process modeled using Maya particles. The viral RNA fragments are modeled with particles as well. Scale bar ≈10 µm.

Courtesy Shaftesbury Films and AXS Biomedical Animation Studio. Copyright Shaftesbury ReGenesis II Inc.

Introduction

Dynamic simulations are used in productions for the entertainment industry to create animations that would otherwise be intractable using traditional techniques. These typically involve many objects, like particles comprising a billow of smoke, and complex physical interactions, like the response of fur to an animal's movements. Common examples in Hollywood films include simulations of water, explosions, hair, and fabric. The field of scientific animation uses the full range of Maya Dynamics capabilities in order to emulate a wide variety of natural phenomena: molecular interactions (Figure 07.01); fluid dynamics in vivo and in vitro (Figure 07.02); virus particles budding from an infected cell (Figure 07.03); bacterial flagellae (Figure 07.04); cell surface deformations (Figure 07.05).

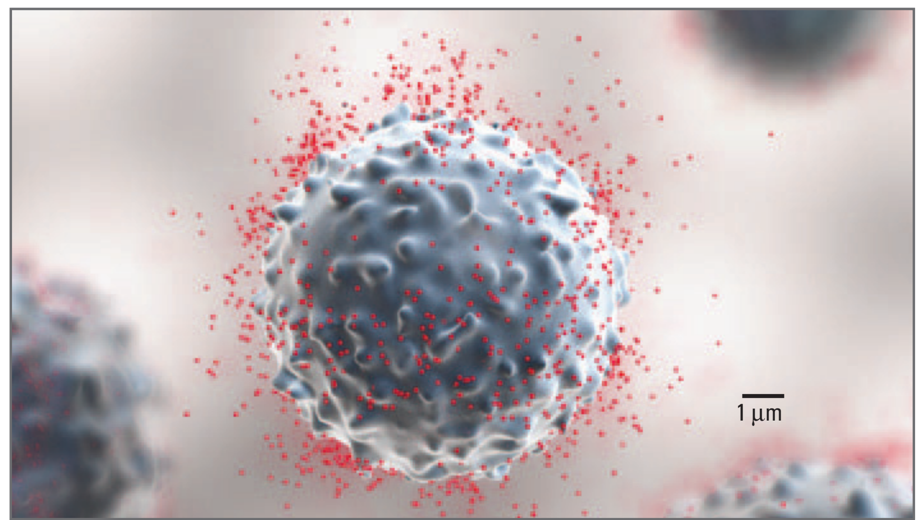


FIGURE 07.03

The budding of HIV virus particles (the red specks) from an infected T-lymphocyte cell was animated using Particle Dynamics. Scale bar ≈1 µm.

Courtesy Shaftesbury Films and AXS Biomedical Animation Studio. Copyright Shaftesbury ReGenesis II Inc.

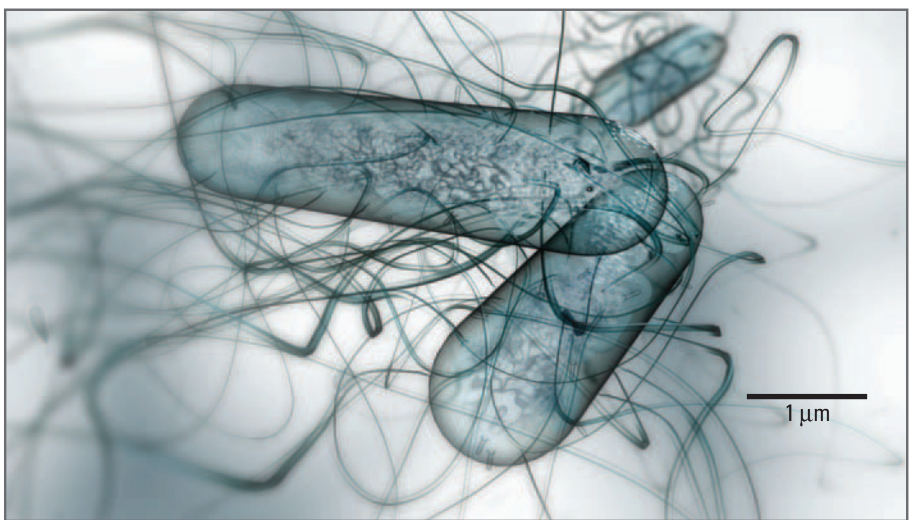


FIGURE 07.04

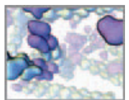
Flagellae, the swimming appendages employed by these bacteria, *Bacillus cereus*, were modeled with Maya Hair. Hair uses specially rigged spline curves to simulate dynamic effects on fibrous structures. Scale bar ≈1 µm.

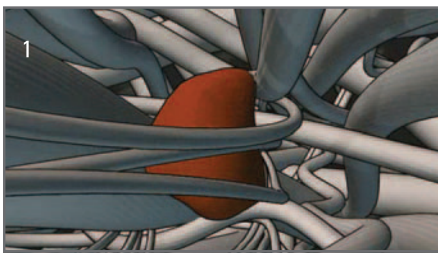
Courtesy Shaftesbury Films and AXS Biomedical Animation Studio Inc. Copyright Shaftesbury ReGenesis II Inc.

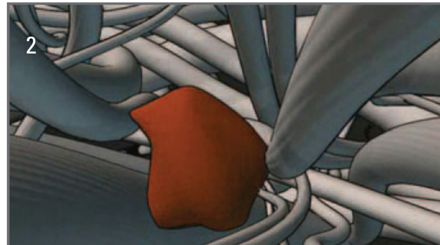
This chapter provides a basic introduction to the Dynamics module, which includes particles, soft body, and rigid body dynamics. The Dynamic engine simulates the effects of physical forces on objects to make them move about, collide with, repel, and attract one another. The **force fields** and collision detection capabilities of the Maya Dynamics engine are ready-made tools that can be built on with custom scripted animation to approximate events in the cellular and molecular realms—the random motion, interactions, and deformations of agents (cells, molecules, and components of tissues). If you plan to use Maya for interpretive visualization purposes, the Dynamics module will be an essential tool.

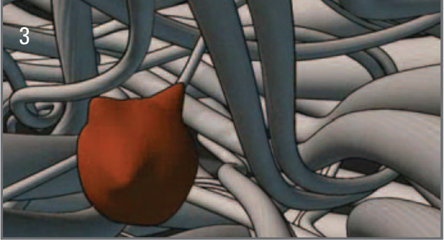
The Dynamics module in Maya Unlimited includes Fur and Fluids simulation capabilities, which aren't included in Maya Complete or Maya PLE.

The fact that you may be considering Maya as a platform for biological simulation warrants a comment here on the realism of Maya's physical simulation capabilities.









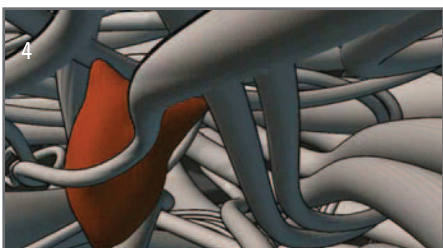


FIGURE 07.05

Still frames from a movie of cell locomotion. The amoeboid crawl of a lymphocyte (pictured here) was animated using soft body dynamics. Attraction forces (Newton fields) were used to extend protrusions (pseudopodia) from the cell membrane which was modeled as a soft body. The cell body is roughly 10 μm in diameter.

Because Maya evolved as a tool for the entertainment industry, the implementation of dynamics focused on visual fidelity to real-world events rather than fidelity to the original governing equations of motion used. In some cases, successive iterations of the force calculations are made (behind the scenes) to generate better looking motion. Moreover, many attributes have been given animator-friendly names that are difficult to trace back to their role in the original physics equations—names such as bounciness and goal Smoothness. It is therefore difficult to directly relate attributes in Maya (the magnitude of a force field, for example, or the friction attribute of an object) to coefficients within the original motion equations. This fact presents a challenge to the scientist who wants to factor in a specific drag coefficient or fluid viscosity, for example, because those parameters don't exist in any easily accessible form in Maya.

This is not to say that Maya Dynamics cannot be used for in silico biology. In our experience, Dynamics has been tremendously helpful for rapid prototyping purposes in cellular and molecular simulation models (Figure 07.01). Furthermore, in cases where you don't require real-world physical parameters, but instead are looking at simulation events in a relative and subjective manner, Maya's built-in physics engine can save you a great deal of time and effort that would other be spent writing custom code.

The first tutorial in this chapter is a simple introduction to rigid body Dynamics. In it you'll animate a sphere to move randomly within a container, while colliding with its walls. The second tutorial lets you explore Particle Dynamics, with a twist: not only will you make particles move in response to a force field, you will also simulate collisions amongst them, something which Maya was not designed to do. Since this requires more than the built-in particle dynamics tools, you will get a sneak peek at how to use a simple, custom MEL expression in collaboration with Dynamics. In the final tutorial, you will create an animation movie file, called a **playblast**, from the particle simulation.

The Dynamics module

The Dynamics module is accessed through its own menu set, which can be selected from the menu at the far left of the Status Line or activated using the hotkey, F4.

This set is used to created particle objects and to manage the dynamic behavior of particles and of existing objects you made using the Create menu, or the NURBS and polygonal modeling tools. Any surface can have dynamic behavior applied to it, by turning it into a **rigid body** or **soft body** object. Strictly speaking, a spline cannot be made into a dynamic object. The caveats to this are Hair and Fur which do make use of dynamic curves. The Hair and Fur modules ship only with Maya Unlimited.

Maya Hair and Fur

Maya Help → Using Maya → General → Dynamics and Effects
→ Hair
→ Fur

The Dynamics engine

Whereas the Dynamics module is the set of nodes, menus, and tools that let you perform dynamic simulations in Maya, the *Dynamics engine* is the behind-the-scenes software that does the work; it calculates the motion of objects based on the attribute values you input for the various dynamics nodes in a scene; for example, the `Intensity` attribute of a force field node or the `Mass` attribute of a particle node.

Forces: Collisions and fields

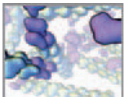
Forces come in two varieties in Maya:

Collision forces are generated when dynamic bodies contact one another.
Fields act on objects from a distance (like gravity, for example).

Collision forces (or **impulses**) are calculated by the Dynamics engine using the masses, velocities, and accelerations of the colliding objects. Other attributes are also accounted for, including friction, damping, and resilience (or elasticity). You will get a crash course on collisions in *Tutorial 07.01: Rigid body dynamics*. While collision forces in Maya can be thought as the forces at work when a ball bounces off a surface for example, force fields (called simply **fields** in Maya) are akin to gravity, magnetism, or the pressure generated by wind. Objects within the effective range of a field fall under its influence. Furthermore, fields exist as nodes with attributes in a Maya scene, whereas collision forces are not represented discretely in the Dependency Graph (DG), but are calculated *on the fly* in order influence objects (geometry and particles) that are part of the DG. Like many other DG items, a field has a transform and a shape node which determine its physical location in the 3D scene and its characteristics (strength, orientation, etc.), respectively.

In Maya, fields emanate from one or more points of origin, called an emitter. In the case of a gravity field, for example, the origin is the location(s) toward which objects under its influence are attracted. An emitter can be the field's own transform node, the transform node of a geometric object or it control vertices (a surface emitter), or individual particles from a particle object (more on particles below). To influence objects or particles in a scene, a field must be connected to them in the DG. Making these connections is simple and will be covered in the tutorials to follow.

Maya has nine fields in total: Air, Drag, Gravity, Newton, Radial, Turbulence, Uniform, Vortex, Volume Axis. Some of these, such as the Newton field, mimic a single



force vector, resulting in the acceleration of the affected object(s). Others, like the Air field, apply a balance of force vectors, resulting in constant velocity (zero acceleration). A Turbulence field applies a force that changes direction randomly over time. The resulting motion is a quick stand-in for Brownian diffusion. We will explore the Turbulence field in more detail in this chapter's tutorials.

Maya Help → Using Maya → Dynamics and Effects → Dynamics → Fields → Overview of fields

Particle objects

Particles are perhaps the mostly widely used Dynamics feature in computer graphics (CG) special effects. Stripped to its essence, a Maya particle is a point in space with attributes describing its motion, including velocity and acceleration, and its static properties, including mass and color. Because Maya particles are points in space, and without shape (geometry), they can be created, destroyed, and animated by the thousands quickly without large computer processing penalties. Because they are easy to work with in large numbers and can be made to respond to physical forces, they are a well suited to the many natural phenomena that can be modeled in a particulate fashion, such as water, smoke, clouds, rain, snow, and insect swarms.

Maya Help → Using Maya → Dynamics and Effects → Dynamics → Particles → Overview of particles

Particle emitters

When you create a particle object in Maya, you are creating a particle group node. Individual particles then are "born" (see Figure 07.06), or temporarily created, from the group node. Particles are birthed from the group node either manually, using the Particle Tool or automatically through an emitter node, of which there are five types (Figure 07.06). A Surface emitter was used to birth the red virus particles in the HIV animation shown in Figure 07.03.

Particle attributes

A particle group node has particle attributes which influence all particles within the group equally. In addition, there is a large number of attributes whose effects can be varied from one individual particle to the next. These are called per particle attributes and will be discussed in *Tutorial 07.02* below.

Particle goals

Maya lets you assign a geometric surface (NURBS or polygon object) to particles as a goal object. As the particle simulation plays, particles are attracted to their goal with a strength that you control using attributes. This can be used to make a swarm of particles follow a geometric model around the scene or to make particles move toward

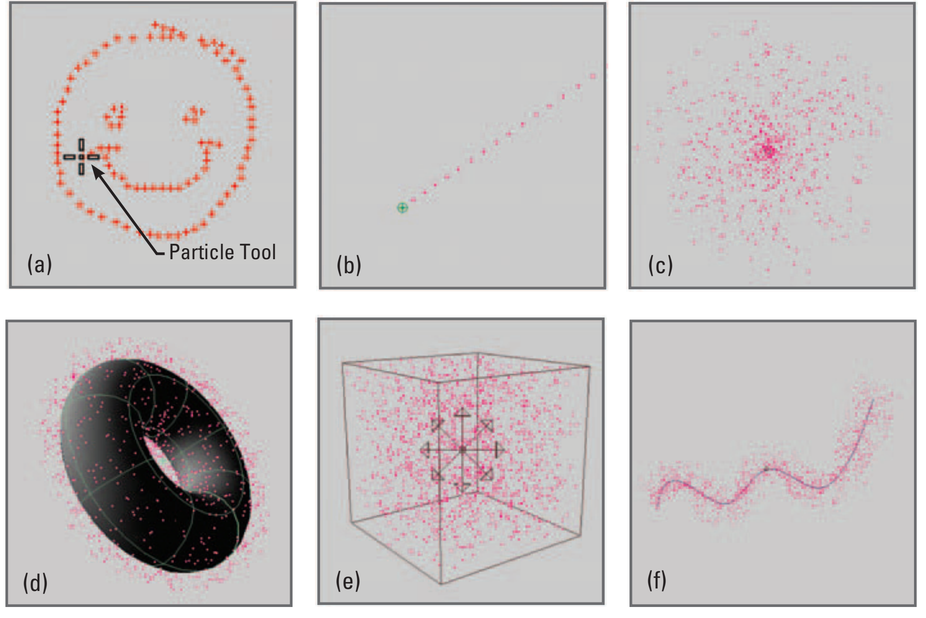


FIGURE 07.06

Particles are born from the Particle Tool (a) or from one of five types of emitter ((b) through (f)).

(a) The Particle Tool is used to place, or draw, particles in a scene.

(b) Directional Point emitter.

(c) Omni Point emitter.

(d) Surface emitter.

(e) Volume emitter.

(f) Curve emitter.

and then stick to a surface. Furthermore, particles themselves can be goal objects for other particles.

Rendering particles

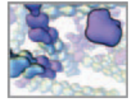
Because a particle is merely a point in space, something must be done to make it visible; to make it look like water, smoke, or a bumble bee when it is rendered. How it ultimately appears is determined by an attribute called its particle render type, of which there are 10 varieties (Figure 07.07). As you will see in a later chapter, Maya has several rendering engines which it uses to render pictures of a scene. Two of these, called the Hardware and Software Renderers, are used to render particles. Briefly, the Hardware renderer uses your computer's **graphics processor** to create images, while the Software renderer uses software algorithms native to Maya and your computer's **CPU** to make the images. Of the ten particle render types, seven use Hardware rendering and three use Software (s/w for short). Hardware particles cannot be rendered using the Software rendering engine and vice versa. On top of choosing a particle render type, you also assign a shader to a particle group, in the same way you would to a surface. The shader determines color and lighting properties.

Rendering particles

Maya Help → Using Maya → Dynamics and Effects → Dynamics → Particles → Render particles

Curve flow

Curve flow is a particle dynamics feature that uses pre-defined expressions to flow particles along a spline. Attributes control the nature of the flow. Curve flow is useful



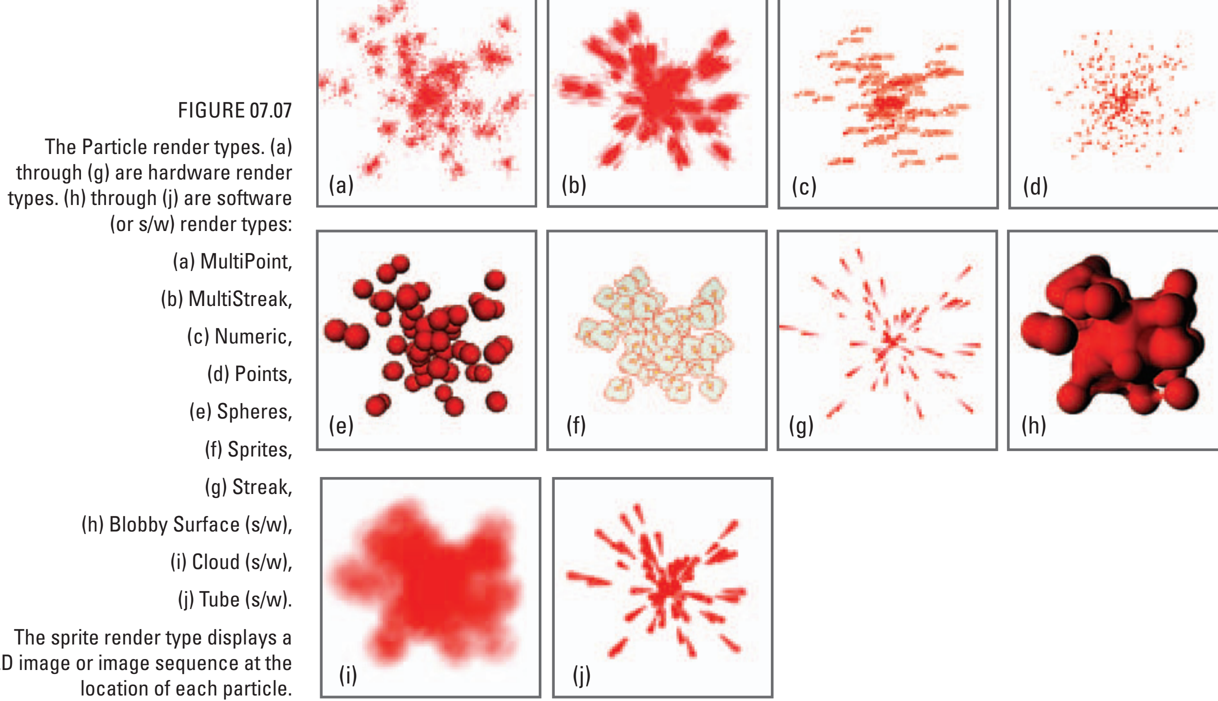


FIGURE 07.07

The Particle render types. (a) through (g) are hardware render types. (h) through (j) are software (or s/w) render types:

(a) MultiPoint,

(b) MultiStreak,

(c) Numeric,

(d) Points,

(e) Spheres,

(f) Sprites,

(g) Streak,

(h) Blobby Surface (s/w),

(i) Cloud (s/w),

(j) Tube (s/w).

The sprite render type displays a 2D image or image sequence at the location of each particle.

for many applications in which particle must follow a path. For example, it makes a snap of animating the flow of blood cells through a convoluted vessel.

Rigid bodies

Rigid body Dynamics treats objects as if they had rigid surfaces. Rigid bodies react to fields and friction forces, collide with, slide and roll along, and bounce off one another. You can specify whether Maya calculates a rigid body's physical interactions with the scene based on its CV-defined surface, or using a proxy geometric surface such as a cube or a sphere. The proxy method trades surface accuracy for computational efficiency; Maya can calculate collisions for a cube more quickly than for a detailed surface defined by hundreds or thousands of CVs.

A rigid body can be either static or dynamic, a setting determined by its "Dynamic" attribute. When static, an object is anchored in the scene, unable to move, like a concrete floor, for example. When dynamic, it is free to move about the scene in response to forces and collisions. When a dynamic body collides with a static one, the latter does not react. We use static bodies for compartments in molecular dynamics simulations for example.

Rigid bodies

Maya Help → Using Maya → Dynamics and Effects → Dynamics → Soft and Rigid bodies → Rigid bodies

Soft bodies

Soft bodies are deformable surfaces that respond to fields and collisions. A soft body is created from a regular NURBS or polygon surface by substituting a particle for each CV in the original surface. The particles are like all particles in Maya except that they are connected together, forming a surface. They can use the original surface as a goal object, which plays the same role as a particle goal object; the soft body particles try to conform to the surface of the goal in the face of outside forces.

The lymphocyte cell in Figure 07.05 was modeled as a soft body. As the cell moves through an environment of fibrous **extracellular matrix (ECM)** proteins, force fields along its path pull on the cell's surface as it passes by. This gives the appearance of the cell is extending protrusions in a bid to find anchorage points for locomotion. When the cell passes out of range for a particular force field, the protrusion returns to the cell body. Setting the soft body to collide with the fibers, which are modeled as static rigid bodies, ensures that the cell protrusions don't penetrate the fibers. The result is an amoeboid crawl, with its characteristic transient contacts with ECM proteins.

The lymphocyte animation described above uses Dynamics to do what you will do with joints—another animation construct—in *Chapter 16*. However, caution should be used when rigging surfaces using Dynamics since the resulting animation can be very difficult to control. This is due to the number of variables the Dynamics engine uses to calculating the final motion. If you have a clear idea of how you want an object to deform as it moves in a scene, it is often better to use joints and deformers which you can control very deliberately. Nonetheless if you require collision detection, Dynamics can play a role.

Soft bodies

Maya Help → Using Maya → Dynamics and Effects → Dynamics → Soft and Rigid Bodies → Rigid bodies

Nucleus and nCloth

With Maya 8 Autodesk introduced a new dynamics engine called Nucleus. As of Maya release 8.5 one module uses the engine **nCloth**, a system of dynamically linked particles available only in Maya Unlimited. Although designed to simulate the natural flow of clothing for **computer-generated imagery (CGI)** characters, nCloth can be used on all NURBS and polygon objects in Maya for the same kinds of projects you'd use soft bodies for, including the simulation of a wide variety of deformable materials. The advantage of nCloth objects over soft bodies is a more advanced and robust physics engine with better performance. Collision detection is much improved over the primary Maya dynamics engine that is responsible for the soft and rigid body calculations.

Dynamic Relationships Editor

This editor (Figure 07.08) allows you to make and break DG connections among dynamic objects and fields. When you select a dynamic object, its name appears highlighted in the outliner window of the editor; objects or fields to which it is connected appear highlighted under the appropriate heading—Fields, Collisions, Emitters, or All—in the relationships window. You will use this editor to connect a particle object to field in *Tutorial 07.02*.

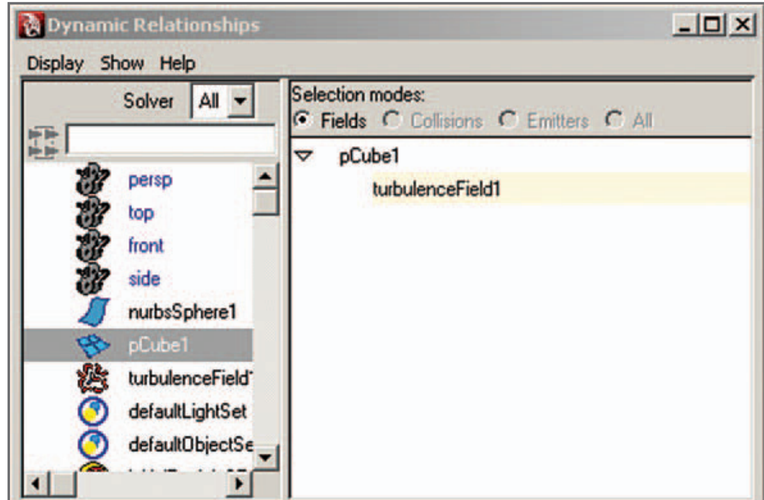


FIGURE 07.08

The Dynamic Relationships Editor is used to display dynamic objects by name and to make and break connections between them.

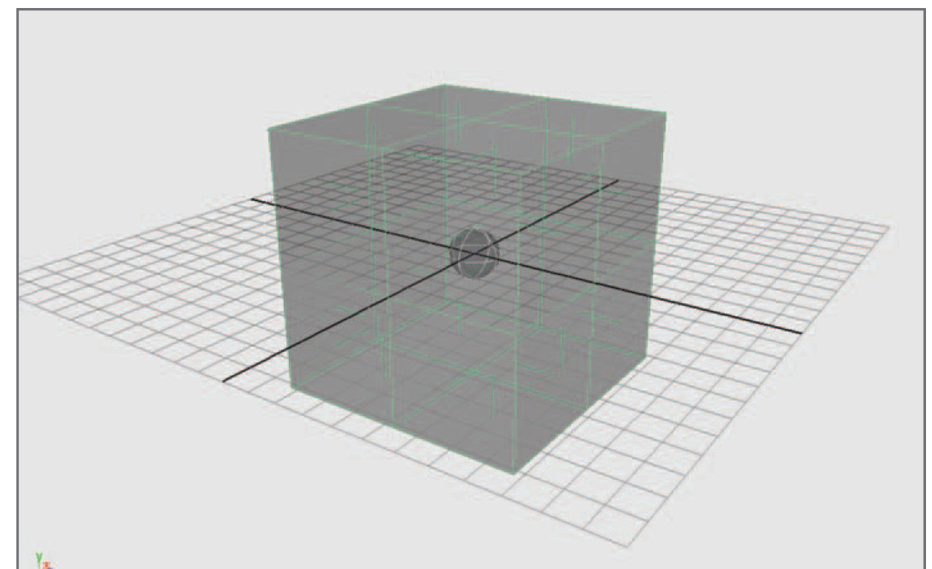


FIGURE 07.09

The setup for the rigid body dynamics tutorial is straightforward: a NURBS sphere inside a larger polygon cube. The surface normals of the cube have been reversed so that the sphere will collide with its inner surface.

Tutorial 07.01: **Rigid body dynamics**

This tutorial introduces rigid body dynamics; you'll create two primitives, a NURBS sphere inside a polygon cube, make them into rigid bodies, and make them collide with one another. The cube is passive and the sphere active, with a Turbulence field to push it around. Figure 07.09 shows the setup for this scene.

Set up your scene

1. **Start a new scene.**
2. **Select the Dynamics menu set from the menu at the left end of the Status Line.**
3. **Choose Window → Settings/Preferences → Preferences.**
4. **Under Settings → Working Units, set Time to NTSC [30 fps].**
5. **Under Setting → Timeline → Playback, set:**
   (a) **Looping to Once.**
   (b) **Playback Speed to Play every frame.**
6. **In the Range Slider at the bottom of the main window:**
   (a) **set the Playback Start Time to 1.**
   (b) **set the Playback End Time to 3000.**
   (c) **set the Current Time to 1.**

At 30 fps, you'll get 100 seconds of animation. With Maya set to play every frame, the actual playback speed depends on computer speed. On a Dell Dimension 8300 PC, our benchmark machine, this animation played at 150 fps, for 20 seconds.

Create and position the objects

1. **Create a polygon cube with the following settings:**

   Length = Width = Height = 10.

   Subdivisions X, Y, and Z = 2.
2. **Create a NURBS sphere with the following settings:**

   Radius = 1.

   Sections = Spans = 5.

Maya Dynamics works on both NURBS and polygonal geometry, and combinations of the two. In this tutorial, you could just as easily use two polygonal or two NURBS objects.

By default, and with Interactive Creation turned off, the objects should be created at the world origin, with the sphere inside the cube (if not, position them so).

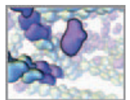
Viewing the scene

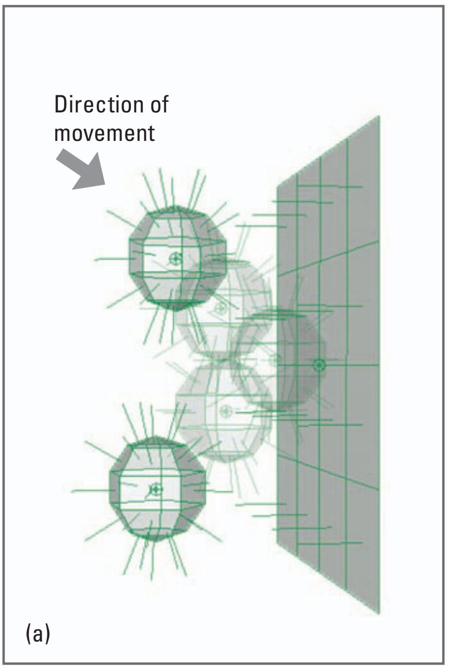
In order to see the sphere as it bounces about inside the cube, you'll want the cube to be somewhat transparent. With your scene view shading mode set to Wireframe, you will be able to see the sphere inside the cube. However, if you wish to view the surfaces in Smooth- or Flat-shaded mode, the cube will obscure the sphere. One solution is to use **X-ray** shading which makes all Flat- or Smooth-shaded objects semi-opaque.

In the Panel menu set, choose Shading → Shade Options → X-ray

or **Shading → X-ray** (depending on your Maya version)

Alternately, if you're familiar with creating and assigning shaders, you can assign separate shaders to the sphere and cube, with the transparency dialed up on the latter. We'll cover shaders in the next chapter.





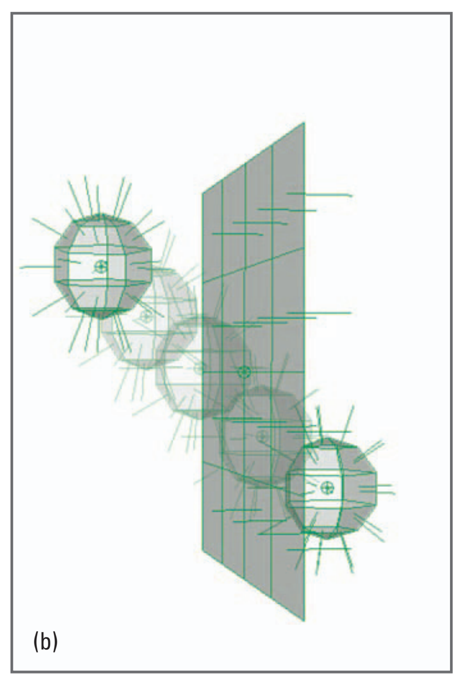


FIGURE 07.10

Surface normal direction affects rigid body interaction. When the moving sphere encounters the positive-normal side of the plane surface, a collision occurs. When it encounters the negative-normal side, it passes right through the plane.

Normal direction

In *Chapter 05*, we noted that one can tell the inside of surface form the outside by the way the normals point. When the Dynamics engine evaluates potential collisions for a surface, it looks for objects approaching from the outside, or the positive-normal direction (see Figure 07.10). Objects approaching from the inside will be ignored and pass right through the surface.

1. **Select the cube, open the Attribute Editor, and select the shape node.**
2. **Under Mesh Component Display, check Display Normal.**
3. **In the Normal Size field, enter 0.4.**

By default, surface normals point outward from the center of an object; the cube surface normals are facing away from the sphere. Therefore, the sphere will approach the cube from the negative-normal direction and pass through it. To remedy this, you must reverse the Normals:

The Hotbox provides a quick way to access menus for software modules other than the one currently displayed in the main window.

1. **Select the cube.**
2. **Press and hold the Space bar to activate the Hotbox.**
3. **Choose Edit Polygons → Normals → Reverse ❐.**
4. **From the Mode menu, select Reverse.**
5. **Press the Reverse Normals button to apply the change and close the options window.**

Differential equation solver method	Description
Midpoint	Faster with less accuracy
Runge–Kutta	Medium speed and accuracy
Runge–Kutta adaptive	Slower speed, greater accuracy (default setting)

TABLE 07.01
Solver Method attribute settings.

Create the rigid bodies

When you make an individual object into a rigid body, you can choose whether to make it active or passive. Alternately, if you make multiple objects dynamic all at once, they will be either all active or all passive. You can then alter the active/passive attribute for each object.

1. **Select the cube and the sphere.**
2. **From the main menu, choose Soft/Rigid bodies → Create Active Rigid Body ❐.**

In the options window you will see attributes for Mass, Friction, Bounciness, Damping, and Impulse. Leave these set to their default values and press the Create button. In the Channel Box, you will see a new `rigidBody` node for both the cube and the sphere, along with a long list of attribute values.

If you hit Play in the Animation Controls now, nothing will happen. This is because there is currently no force field in the scene and because the Impulse values were set to 0 for both objects.

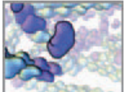
The rigidSolver node

Below the `rigidBody` node attributes and under the heading INPUTS, you will see another node called `rigidSolver` with its own list of attributes. A solver does the calculations that determine the motion in a dynamic simulation. Its attributes provide high-level control over rigid body dynamics for objects to which it's connected. A scene can have multiple rigid solvers, but only one can be the active or "current" solver.

Three rigidSolver attributes determine the balance between speed and accuracy of dynamics calculations. These are Step Size, Collision Tolerance, and Solver Method. Decreasing Step Size, the time between calculations, improves simulation accuracy at the expense of speed. The smaller the Collision Tolerance value, the more accurate the collision detection calculations and the slower the simulation. Table 07.01 shows the different settings for the Method attribute and their effects on accuracy and speed. With a simple scene like the one you just created, the solver method will make little if any difference in the playback speed of the animation. It will, however, make a big difference when the number of rigid bodies begins to increase.

rigidSolver attributes

Maya Help → Using Maya → Dynamics and Effects → Dynamics → Dynamics nodes → Soft and Rigid Body nodes → rigidSolver node



Make the cube passive

Since you made both objects active in the previous step, you'll now make the cube passive, so it behaves as a stationary container for the dynamic sphere.

1. **Deselect both objects then select the cube on its own.**
2. **In the Channel Box, scroll down until you locate the attribute called, Active.**
3. **In the Active field, enter 0 or off.**

Create a Turbulence field

1. **Select the sphere.**
2. **In the main menu, choose Field → Turbulence ❐.**
3. **Set Magnitude to 50 and Attenuation to 0.**
4. **Press the Create button.**

Magnitude controls the strength of the field and Attenuation, the degree to which the magnitude decrease with distance from the field's transform. The field has its own transform node, located at the world origin. Its XYZ position only matters if attenuation is set to a value greater than zero.

Play the animation

When you press Play, the sphere should move about and bounce off the walls of the cube. Take a few minutes to adjust the attributes in the sphere's `rigidBody` node, such as Mass, Bounciness, and Friction, and observe the effects. Mass has a strong influence on motion. As in the real world, mass impacts the inertia and momentum of objects in a Maya scene. In a collision between a light and a heavy object, the latter will prevail.

Note that each time the Current Time Indicator returns to the start of the playback range, the sphere returns to the position it was in when you first created it. This is called its **Initial State** and was set to the position the sphere was in when you turned it into a rigid body. You can set any position to be the Initial State by selecting the sphere and choosing

Solvers → Initial State → Set for Selected.

cycleCheck

From time to time, you may get a warning such as "`Cycle on <attribute name> may not evaluate as expected`". This due to a possible **cycle** in the DG; an attribute value depends on a value that in turn depends on it. This can result in the improper evaluation of an attribute in a dynamic simulation, and therefore affect the resulting

motion. You can check if an attribute is actually in a cycle by using the MEL command, cycleCheck, as follows:

1. **Choose Window → General Editors → Script editor.**
2. **In the Command Input (lower) field of the Script Editor, type the following MEL script:**

```
if (`cycleCheck <attribute name>` > 0) {
     print( "<attribute name> is in a cycle\n" );
}
else print "no cycle\n";
```

3. **Hit Enter on your keyboard's numeric keypad to execute the script.**

The Script Editor and MEL commands will be discussed in detail in *Chapter 12*.

One of the two messages, "<attribute name> is in a cycle" or "no cycle" with appear, or *print*, in the History (upper) field of the Script Editor and in the Command Feedback field of the Command Line.

However, even if there is no cycle, you will continue to get the warning. To disable the warning, use the cycleCheck command to turn off checking:

In the Command Input (lower) field of the Script Editor, type the following:

```
`cycleCheck -e off`;
```

Memory caching

You may have noticed that scrubbing the Timeline produces strange results in this simulation. When you scrub backwards, nothing happens at all. This is because Maya calculates dynamic attribute data at each frame, and only in forward playback mode. When combining keyframed animation with dynamics, or animating a camera to follow the action, it's helpful to move freely along the timeline, and see your simulation behave predictably. You can facilitate this with **memory caching**, which stores dynamic attributes in RAM and then disables their connection to the Dynamics engine. Memory caching can speed up rendering since Maya needs only to read the cached attributes of objects rather than calculate them anew using the Dynamics engine. To activate memory caching:

When your scene is ready to render, turning on **memory caching** can speed up rendering since Maya will not have to perform dynamic calculations for the cached objects. Instead, Maya will read the appropriate dynamic attribute values from the cache which is typically faster than calculating them on the fly using the Dynamics engine.

1. **Select at least one of the dynamic objects you wish to cache; in this case, the cube or the sphere. All objects connected to the selected one, through a rigidSolver node will be cached.**
2. **Set the Current Time equal to the Playback Start Time.**
3. **Choose Solvers → Memory Caching → Enable.**
4. **Press Play. A cache will be created for as many frames as you allow to play.**

Now you can scrub the Timeline and the animation will update properly using the cached data. With memory caching enabled, changes to dynamic attributes, such as the strength of a force field, will have no effect on rigid body attributes. Disabling memory caching reconnects rigid bodies to the Dynamics engine, allowing their attributes to

update. However, if you were to then re-enable memory caching, rigid body attributes would revert to their previously cached state. For dynamic attributes to update correctly, you must delete, then re-enable memory caching in the Solvers menu.

Make the cube active

For a little fun, you'll reset the cube's Active attribute to "1" and see how it responds to collisions with the sphere. Start by deleting the memory cache:

1. **Select the cube.**
2. **Choose Solvers → Memory Caching → Delete.**
3. **In the Channel Editor, enter 1 or in the Active attribute field.**

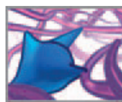
On playback the cube won't move (because it's not connected to the Turbulence field) until the sphere collides with it the first time. After that, it will change direction each time the sphere hits it. Next, let's connect the cube to the Turbulence field and observe the change in motion.

1. **In the main window, choose Window → Relationship Editors → Dynamic Relationships.**
2. **In the outliner portion of the Dynamic Relationships Editor, select the cube.**
3. **The Selection Modes portion of the editor displays a list of fields in the scene—in this case, only one appears.**
4. **Click on the Turbulence field name in the editor to connect it to the cube.**

Now when you press Play, the cube will be pushed about by the Turbulence field in addition to being knocked about by the sphere. To disconnect the cube from the Turbulence field, repeat steps 1 through 4 above. To return the cube to its role as a stationary container set its Active attribute to **0** or **off**. One can imagine duplicating the sphere ten, a hundred, or a thousand times, then tailoring the rigid body, rigid solver, and field attributes to approximate Brownian motion, for example. Add to this, the ability for spheres to form complexes with one another, and you have the makings of a molecular dynamics simulation! You'll find the finished scene file for this tutorial on the CD-ROM:

07_Dynamics/scenes/tutorial_07_01_done.ma

So let's get a feel for what such a step would involve by using particle dynamics to simulate hundreds of colliding objects within a container. Particles are much less expensive computationally than rigid bodies for the same number of moving bodies in a scene. Nonetheless, particles are only points in space, with none of the surface detail of rigid bodies, and are therefore not well suited to dynamic situations where surface interaction is essential. Where a rigid body collision will impart rotation to an object, based on the location of its center of mass relative to its peculiar topology, there is no such attribute as rotation for individual particles.



Tutorial 07.02: Particles in a container

In this tutorial, you will get Maya to emit particles into a volume, a cylinder, and move them about using a Turbulence field. Next, you will create a radial field that the particles both emit and are repelled by simultaneously, in order to simulate inter-particle collisions. By definition, particles have no radius since they represent only points in space and not space-filling objects. A force field on the other hand can have a radius—represented by its effective range—much like an atom has an effective radius of repulsion, inside of which other atoms cannot approach due to the field generated by its inner electron shells. By emitting a Maya force field from the position of a zero-radius particle, you can therefore simulate a collision radius for the particle. Finally, you will use the per particle attribute colorPP to make colliding particles more obvious by turning them bright red when they experience forces beyond a threshold value—a sudden increase in force due to a close encounter with a force field (i.e. another particle) or with the container wall. To begin, follow the setup instructions from the previous tutorial.

Create the container

Create a Polygon cylinder as follows:

1. **Choose Create → Polygon Primitives → Cylinder ❐.**
2. **In the Polygon Cylinder Options window choose Edit → Reset Settings.**
3. **Make the following changes to the default attribute settings:**

   Radius: 50

   Height: 100

   Subdivisions Around Axis: 10

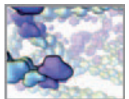
   Subdivisions Along Height: 4
4. **Press the Create button.**

The cylinder should be located at the world origin. If not, move it there now. Next, reverse the surface normals so that the particles will collide with the container:

1. **Select the cylinder.**
2. **Press and hold the Space bar to activate the Hotbox.**
3. **Choose Edit Polygons → Normals → Reverse ❐.**
4. **From the Mode menu, select Reverse.**
5. **Press the Reverse Normals button to apply the change and close the options window.**

Create the particle emitter

When you create an emitter, Maya automatically creates a particle object and connects it to the emitter. Conversely, you can create a particle group without making an



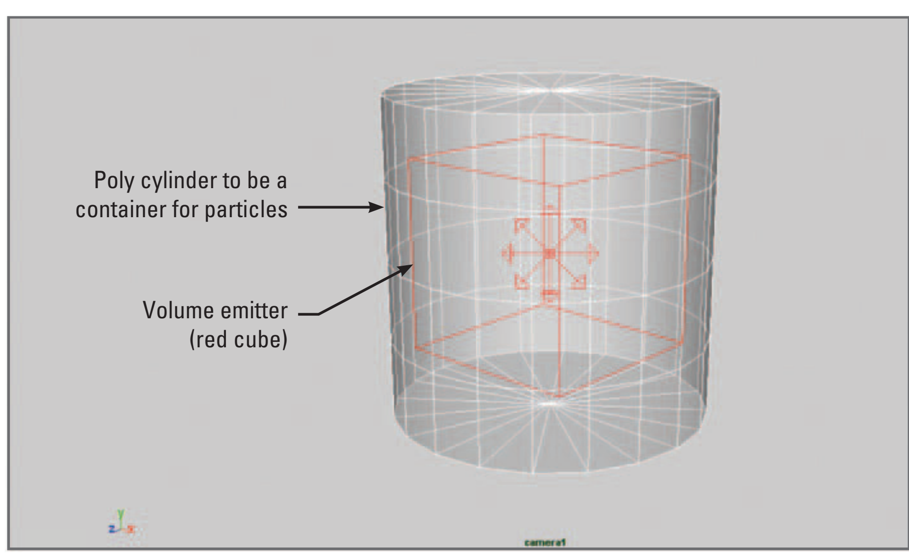


FIGURE 07.11

Setup for the current Tutorial. The cube volume emitter has been scaled to three times its original size.

Upon creation, particle objects are by default connected to the Dynamics engine.

emitter. You can therefore emit more than one particle object from a single emitter. For now, you'll create one emitter and one particle object. Later you'll add another particle object to the emitter.

1. **In the main window or the Hotbox, choose Particles → Create Emitter ❐.**

2. **Choose Edit → Reset Settings.**

3. **Under Basic Emitter Attributes, enter the following settings:**

   (a) **Emitter Type: Volume**

   (b) **Rate (Particles/Sec): 1000**

If "Volume" is not available in the Emitter Type menu, put your cursor in the field and type, "v".

4. **Under Volume Emitter Attributes, Volume Shape, select Cube.**

5. **Under Volume Speed Attributes, enter 50 for Away From Center.**

6. **Press the Create button.**

7. **Select the emitter and scale its transform node up by a factor of three using the Channel Box.**

You can change the emitter type and the other creation attributes at any time in the Channel Box. Figure 07.11 shows what your scene should look like so far.

Particle and emitter attributes

Before discussing the attributes that control particle behavior, let's have a look at the simulation you've set up so far.

Press the Play button, ▶ in Playback Controls.

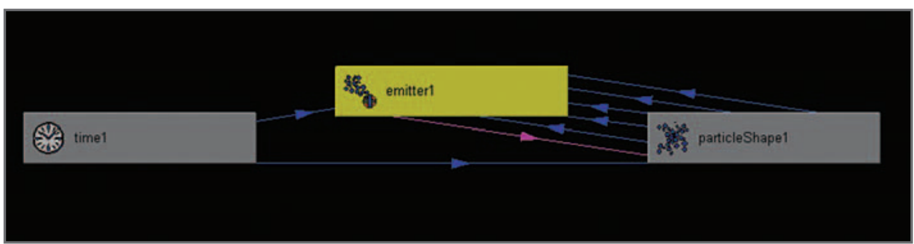


FIGURE 07.12

An emitter is represented by a single node. The Hypergraph does not reveal connections to Maya's Dynamics Engine, since they cannot be modified in the UI, the way other attributes and connections can.

The particles should appear randomly within the cube and move out from the center at constant speed. This behavior is due to the attribute settings of the emitter and the particle object. Below, we'll take a look at some of these attributes; a complete list of them is beyond the scope of this book. As with other components of Maya, what can be done with particles and their attributes is a large topic and has been covered well by other writers.

Emitter node

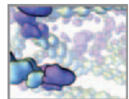
Inspecting the emitter in the Hypergraph (Figure 07.12) reveals that it is composed of only one node. This node holds the transform and shape attributes. In the Channel Box, you'll see the Emitter Type and Rate attributes you set in the Create Emitter Options window. The Rate is the number of particles emitted per second. At 30 fps, your emitter births 200/30 ≈ 7 particles/frame. Together, Direction X, Y, and Z specify a direction vector along which the Directional Speed value (near the bottom of the attributes) is applied. Spread and Speed apply only to directional type emitters. Speed Random applies to all emitter types; it specifies a range of random variation in emission speed. You set Volume Shape when you made the emitter. You can change it here: clicking in the attribute fields brings up a volume shape menu. The Volume Offset attributes offset the emission in space by the specified amounts. Volume Sweep through to Around Axis apply to different volume emitter shapes. Away From Center controls is the main speed setting for cube and sphere volume emitters. Random Direction multiplies the Speed Random attribute. Directional Speed adds a speed component along the vector specified above in Direction X, Y, and Z. Maya Help contains detailed information on every emitter attribute.

Emitters

Maya Help → Using Maya → Dynamics and Effects → Dynamics → Particles → Emitters

Particle shape node

Strictly speaking, a particle has no *shape*. Nonetheless, like most other items in a Maya scene, a particle object has a shape node which defines many of its properties. Just as particles are *born* into a scene, they can die after a finite lifespan. The second particle object attribute listed in the Channel Box, `Emission Volume Exit`, specifies whether a particle will live or die after it leaves the emitter volume. The next attribute, `Lifespan Mode`, has four possible settings, which are superseded by the previous attribute. When set to Constant, the `Lifespan` attribute (last one on the list) determines how many



seconds each particle will live. Lifespan can be randomized per particle with the `Lifespan Random` attribute. `Expressions After Dynamics` sets the order in which Maya evaluates attributes. If set to **0** or **off,** Expressions will not take into account dynamic changes to the scene in the current frame. This setting can make a difference to simulation results and should be considered carefully when evaluating results.

A Conserve attribute value of less than 1 will ultimately result in the particles coming to a stop unless a field keeps them in motion.

The `Conserve` attribute sets the conservation of energy; when it's set to one, particles will continue moving endlessly as a result of initial velocities given them by their emitter. A value less than one will cause particles to eventually come to rest, unless kept in motion by force fields or collisions. `Max Count` is the maximum number of particles that can be emitted. `Start Frame` is the frame at which the first particle is born; this is by default set to the frame that is current when you create a particle object.

1. **Select the particle object.**
2. **In the Channel Box, set Max Count to 200.**

The particle attribute Start Frame determines when the first particle will be born. When you create a particle object, Start Frame is set to the current time on the Time Slider. It can easily be changed later on in the Channel Box.

When you assign a goal object to particles, Maya creates a `Goal Weight` attribute, which sets the strength of attraction (0 to 1) to the goal. `Goal Smoothness` sets how smoothly this attraction changes with the goal weight setting. It has a profound effect on particle motion; higher values (>3) used with high goal weights (close to 1) can have the effect of making particles "sling shot" past their goal(s). Tweaking this value along with `Goal Weight` is often the key to smooth particle/goal motion. For the `Particle Render Type` attribute, you have ten choices; seven use Maya's Hardware Renderer and the rest, the Software Renderer. Examples of the different render types are shown in Figure 07.07. The default setting is Points. Let's set it to Spheres and change the default sphere radius:

When the count of an emitted particle object equals its Max Count value the particle object is said to be **full**.

1. **Select the particle object.**
2. **Open the Attribute Editor and select the particle shape node tab.**
3. **Scroll down until you find the heading, Render Attributes.**
4. **For Particle Render Type, select Spheres.**
5. **Press the Add Attributes For Current Render Type button.**
6. **Enter 3.0 for Radius.**

For efficiency, Maya doesn't preload a particle object with all possible attributes. Many are created only as needed, appearing under the appropriate headings in the Attribute Editor.

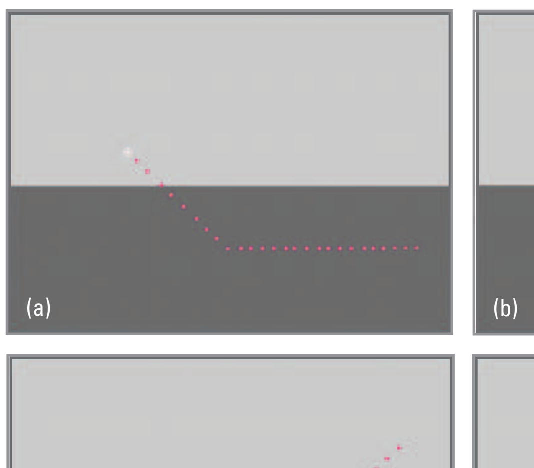
We have skipped over many attributes listed in the Channel Box and the Attribute Editor. These can be left at their default settings for now. Maya Help provides good documentation for particle object and per particle attributes:

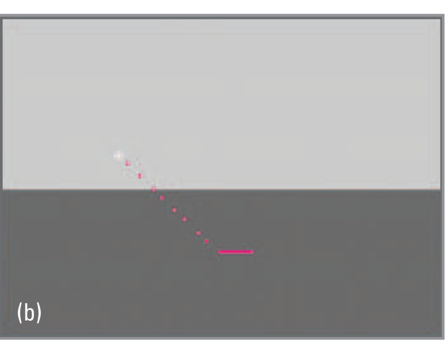
Particle object and per particle attributes

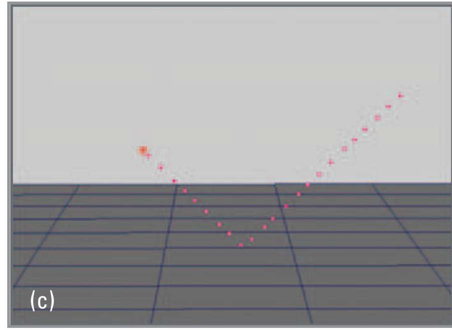
Maya Help → Using Maya → General → MEL and Expressions → Particle expressions → Assign to vectors and vector arrays → List of particle attributes

Make the particles and cylinder collide

Press Play to see the effect of the current attribute settings.







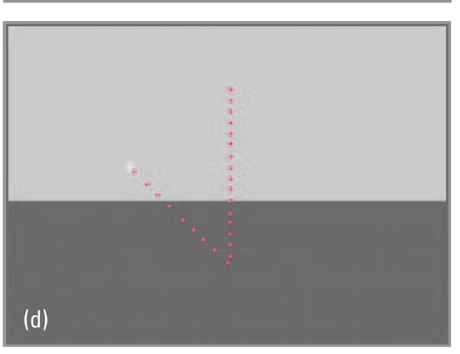


FIGURE 07.13

The effects of Resilience and Friction attributes on particle collisions with a plane.

(a) Resilience = Friction = 0. Particles contact the plane without bouncing and then slide along the plane, unimpaired by friction.

(b) Resilience = 0; Friction = 0.1. Friction slows particles to a halt shortly after they make contact with the plane.

(c) Resilience = 1; Friction = 0. Particles rebound from the plane with no loss of momentum in the vertical or horizontal directions.

(d) Resilience = 1; Friction = 1. Particles rebound from the plane with no loss of momentum in the vertical direction, but complete loss in the horizontal direction due to infinite friction.

Currently when you press Play, 200 particles are created in the space of five frames. They should move outward at constant speed—governed by the attribute Away from Center—and pass right through the cylinder. That is because the cylinder has not yet been connected to the Dynamics engine. Maya uses a special node, a geoConnector, to connect a regular geometric object to the Dynamics engine, so that it can emit or collide with particles, or be the source of a force field. The following steps create a geoConnector node for your cylinder and link it to the particle group for collisions. Collisions between the cylinder and particles can subsequently be turned on and off in the Dynamic Relationships Editor.

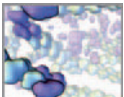
1. **Select the particle object.**
2. **Add the cylinder to the selection by Shift+selecting it in the scene view or Ctrl+selecting it in the Outliner.**
3. **Choose Particle → Make Collide ❐.**

When using Make Collide, it is important to select the particle object **first** and the collision object **second**. Not doing so will generate an error, and the operation won't work.

Two attributes govern collisions: Resilience and Friction. By default, Resilience, the elastic property of a collision, is set to 1. Friction is set to 0. These settings assume no energy is lost in a collision. Figure 07.13 shows the results of different combinations of Resilience and Friction for particles colliding with a polygon plane.

4. **Press the Create button.**

When you play the simulation, the particles should collide with cylinder. With the particle object Conserve attribute set to 1, they will continue to bounce off the walls until the **Playback End Time** is reached.



Add random motion

Here you'll create a Turbulence field to randomize the particle motion and dial back the particle's Conserve attribute to slightly dampen the turbulence effect. With the particle object selected:

1. **In the Channel Box, set the Conserve attribute to 0.99.**
2. **Choose Fields → Turbulence ❐.**
3. **Set Magnitude to 50 and Attenuation to 0.**
4. **Press the Create button.**
5. **Press Play to see the effect of the Turbulence field.**

Inter-particle collisions

In Maya, the particles of one object can be made to collide with those of another object. However, there is no built-in capability for one particle to collide with another particle from the same object. A work-around is to emulate collisions by making every particle within an object the source of a radial field, which at the same time acts on the particles. Start by creating a Radial field using the default settings. You will adjust these settings below using a Manipulator Tool.

1. **If anything is currently selected, clear the selection by:**

   (a) Clicking a clear spot in the scene view

   or

   (b) Entering select-clear in the Command Line.
2. **Choose → Fields → Radial ❐.**
3. **Choose → Edit → Reset Settings.**
4. **Press the Create button.**

The Manipulator Tool

When emanating from each particle, we want the radial field to be effective over a limited distance from the particle's location, and for its strength to diminish, or attenuate, with distance from the particle. This way, the repulsive force felt by another approaching particle will increase as it gets nearer. The field attribute, `Max Distance` sets the effective range, while `Attenuation` attenuates its strength within that range. The Manipulator Tool for the radial field (Figure 07.14) allows you to edit `Max Distance` and `Attenuation` interactively in the scene view.

1. **Select the radial field in the Outliner.**
2. **Press the W hotkey to activate the Move Tool and drag the field transform outside the cylinder for a clearer view.**
3. **Activate the Manipulator Tool by selecting it in the Toolbox or hitting the hotkey T.**

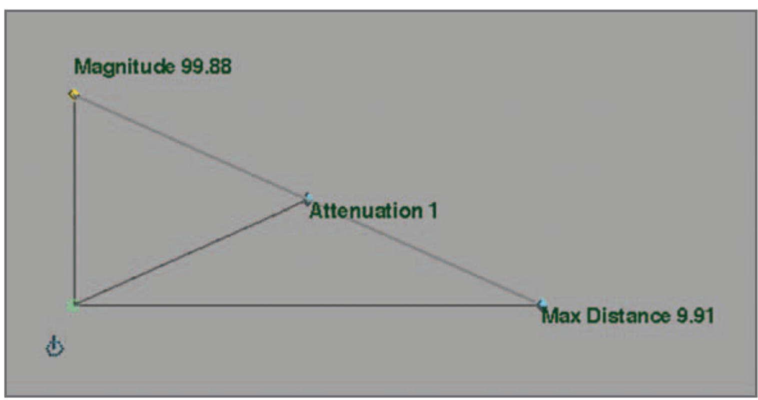


FIGURE 07.14

The Manipulator Tool provides a quick way to adjust commonly used attributes, like the ones shown here for a radial force field. The X-axis here represents the maximum distance at which the force acts. The Y-axis represents the force magnitude. Attenuation is a curve plotting the degree to which Magnitude drops off as Max Distance increases. The icon at the bottom left lets you cycle through other attributes that can be set interactively.

4. **LMB or MMB+drag the Magnitude attribute to a value of roughly 100.**
5. **Drag the Max Distance attribute to a value of roughly 10.**

Once you connect the particle object to this new radial field (in the next section), the particles will be affected by a repulsive force inside of (10 + 10 =) 20 Maya units from one another. With `Attenuation` set to 1, this force will be nil at the 20-unit distance and grow toward full strength as the particles approach each other's position. Note that the radius of your particle sphere—which is only a *visible* radius and has no geometric or space-filling meaning—is 3, while the radius (Max Distance) of the radial field is 10. You could try setting Max Distance to 3 and attenuation to 0, meaning that particles would experience the full, unattenuated Magnitude of the radial field only when they are within (3 + 3 =) 6 units from one another. However, this results in very high accelerations of particles away from each other after they *collide*. We found out quickly by trial and error that a Max Distance of 10 and an Attenuation of 1 allowed particles to almost touch before being repelled away from one another. This more gradual approach kept the collision reactions smooth and in proportion to the speeds at which particles approach each other.

A radial field emanates repulsion (for positive magnitudes) and attraction (for negative magnitudes) radially, in all directions, from the location specified in its transform node.

Connect the particle object to the radial field

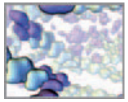
In order to get particles to emit the radial field, they must be connected to it in the Hypergraph.

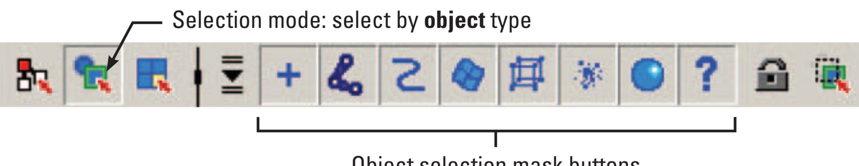
1. **In the main window, choose Window → Relationship Editors → Dynamic Relationships.**
2. **Select the particle object in the outliner portion of the editor.**
3. **In the Dynamic Relationships Editor, under Selection Modes → Fields, select the radial field.**

The radial field won't have the desired effect until you make the individual particles its source.

Make the particles the source of the radial field

Normally, when you make geometry the source of a field, you select the field and then the object and Choose Fields → Use Selected as Source of Field. However, when working





Selection mode: Select by **component** type

Component selection mask buttons

FIGURE 07.15

Selection masks for individual components are created using the Selection mask menu items in the Status Line.

with particles, you must make a component selection—the components being the individual particles. Every particle you wish to be a source for the field must be selected. You must therefore play the simulation until the particle count equals its Max Count value. You will then use a component selection mask to select the individual particles.

1. **Rewind and play the simulation for at least 5 frames, until all 200 particles have been born.**

2. **In the Status Line near the top of the main window, press the Select by component type button.**

3. **LMB+click on the Set by the component selection mask button, .**

4. **Select All Components Off. This wipes the component selection slate clean.**

5. **RMB+click on the Points button and select Particles (see** Figure 07.15**).**

6. **Select the radial field in the Outliner.**

With *Select by component type* on, you won't be able to select the field's transform in the scene view.

If you were to make the particle object the source of the radial field instead of the individual particles within the group, the particles would not influence one another.

7. **Activate the Move Tool or the Select Tool.**

8. **Shift+LMB+drag a selection around *one* of the particles.** You will have to select first one, and then all of the particles at once for this technique to work (see the next step).

9. **Shift+LMB+drag a selection around *all* of the particles.** If the particles do not turn yellow, indicating a selection, you will have to repeat this step.

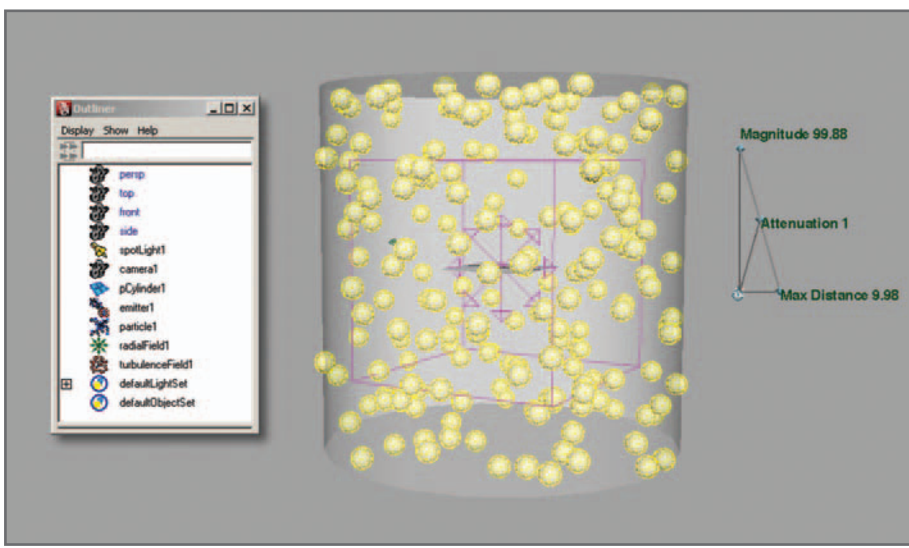


FIGURE 07.16

The particles must be selected as components, and the radial field as a transform, before making the particles the source of the field.

Your selections should look similar to what's shown in Figure 07.16.

10. **Choose Fields → Use Selected as Source of Field.**

11. **Select the radial field on its own.**

12. **In the Channel Box, locate the attribute, Apply Per Vertex and set it to 1 or on.**

13. **Rewind and play the simulation to see the effect of the above steps.**

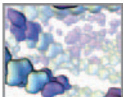
When geometry is used as the source of a field, Apply Per Vertex specifies whether the field will emanate from the pivot point of the object (a setting of 0 or "off") or the object's vertices (a setting of 1 or "on"). When a particle object is used, the individual particles are treated as vertices by the field.

The particles should now repel each other within a certain range, the Max Distance of the radial field. However, this can be difficult to observe in a fast-moving simulation. Suppose you wanted to see the collisions occurring. In the next step, you will use a per particle attribute called `rgbPP` to change the color of individual particles for the brief instances in which they collide.

Per particle color

There are many ways to report data in a simulation. For reporting dynamic events as they occur, color change can be effective. It can be used as an indicator of an individual particle's age or its velocity, for example. Here you'll use it to report when the combined forces acting on a particle exceed a threshold value, indicating a likely inter-particle collision. The `rgbPP` attribute will be linked in an expression to another per particle attribute called **force**, which is a vector attribute that stores the sum of all external forces on each particle. When the force attribute approaches or exceeds a threshold value, the **r** (for *red*) value of `rgbPP` will in turn approach or exceed **1**, the full red value. We determined the threshold value by trial and error; it had to be large enough to filter out the small forces constantly being generated by the Turbulence field, without being too large to cancel out the stronger forces generated by the radial field.

Where color is not a reliable indicator for simulation events—when color-blindness is a consideration, for example—**value** can be used instead. In color theory, value is a measure of light and dark. In Maya, value is measured from 0 to 1; 0 being pure black, and 1 being pure white.



Create the `rgbPP` attribute

1. **Select the particle object, open the Attribute Editor and select the tab for the particle shape node.**
2. **Scroll down to the heading, Add Dynamic Attributes and press the Color button.**
3. **In the Particle Color editor, check the box next to Add Per Particle Attribute and press the Add Attribute button.**

Create a per particle expression

To give you a taste of what's to come, this expression involves essential programming concepts including a MEL command `mag()` and variables of types vector and float. These will be explained in detail in *Chapter 12*. The double slashes, //, indicate a comment—code that will be ignored by Maya.

1. **Select the particle object and open the Attribute Editor.**
2. **Locate `rgbPP`, under the heading, Per Particle (Array) Attributes.**
3. **RMB+click in the field next to `rgbPP` and select Runtime Expression After Dynamics.**
4. **Type the following in the Expression field:**

The MEL command **mag()** calculates the magnitude of a vector.

```
//Query the force PP attribute.
vector $force = particleShape1.force;
//A number to scale the magnitude of the force:
float $thresholdScale = 300;
//mag() returns the scalar magnitude of a vector.
float $mag = mag($force)/$thresholdScale;
//Set the rgbPP value.
particleShape1.rgbPP = <<$mag, 0, 0>>;
```

The magnitude of the Turbulence field fluctuates but remains under 100. For the most part, particles are dark red to black. When the total force acting on a particle exceeds 300 Maya force units in magnitude, due to a close encounter with the radial field, it turns bright red. This is a quick and easy way to report visually on inter-particle encounters. When working with a rigid body simulation, you can use the rigid solver node to report on true collisions between bodies.

5. **Press the Create button.**

If you get a red-highlighted error message in the Command Feedback field at the bottom of the main window, check that you entered the expression script exactly as it appears above, and press Create again.

Play the simulation a few times and observe the particles' behavior. After some trial and error, we set the radial field's Attenuate attribute to 0.5. This makes the force between particle stronger at `Max Distance`, resulting in more pronounced color changes to report the collisions. Figure 07.17 is a screen capture from the simulation.

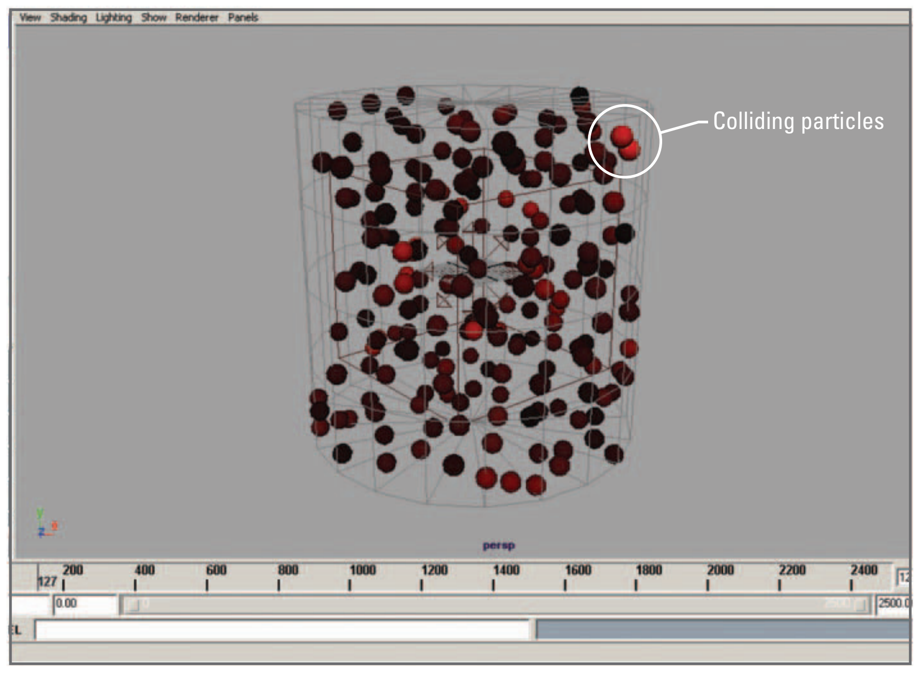


FIGURE 07.17

Linking the `rgbPP` attribute to the per particle force attribute allows us to visualize colliding particles by changing their color from black to red. This technique is most effective when viewed in motion. We have included a Playblast movie of this simulation in the directory for this chapter on the book's CD-ROM.

Save the current file for use in the next tutorial, in which you will create a rough-quality animation to watch in a movie player. You'll find the complete scene file on the CD-ROM:

07_Dynamics/scenes/tutorial_07_02_done.ma

`rgbPP` for Software render types

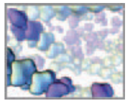
The per particle attribute `rgbPP` works for hardware render type, but not for software render type particles. For software-rendered particles, as special node called the Particle Sampler Info Node allows you to alter the color of a material, using `rgbPP` as an input. With the material assigned to the particle group, the color-change effect works in a software rendering.

Using the Particle Sampler Info Node for per particle colors

Help → Maya Help → Using Maya → General → Tools, Menus, and Nodes → Nodes → Particle nodes → Particle Sampler Info Node

Caching particle data

As you did for rigid bodies in the previous tutorial, you can cache data for particles once you're satisfied with their behavior. This can be done in three ways: with memory caching; using particle disk caching; using a particle startup cache. Caching stores data for specified particle objects for quick playback. It temporarily disables their connection to the Dynamics engine, over a specific playback range.



Particle caching saves time when rendering since it precalculates particle dynamics, thus removing the need for dynamics calculations at render time.

Working with a particle caching

Maya Help → Using Maya → Dynamics and Effects → Dynamics → Solvers and caching

Tutorial 07.03: Create a playblast

A **playblast** is a hardware-rendered animation taken straight from the active view panel. It provides a quick means for previewing your animation. The result of a playblast is an .avi movie file which can be viewed in many types of computer movie player software, including Apple QuickTime, Windows Media Player, and Maya's own fCheck. In this exercise you will shorten the playback range and create a playblast of the particle simulation from the previous tutorial. To start, open the particle simulation scene from the previous tutorial or retrieve it from the CD-ROM.

1. **Set up a single perspective view panel, then maneuver the camera to get a view of the cylinder and particles that you want to render.**

2. **Set the Playback Start Time to 100 and the Playback End Time to 400. At 30 fps, this range will result in a 10-second animation.**

3. **Move or close any windows that obscure any part of the perspective view panel.** A Playblast is made by capturing images of the active view. Anything that obstructs this view can interfere with your Playblast.

4. **RMB+click in the Timeline to bring up Marking menu.**

5. **From the Timeline Marking menu, select Playblast ❐.**

6. **In the Playblast options window, make the following settings:**

   (a) **Select a movie viewer. If you're unsure, choose fCheck.**

   (b) **Set Display Size to Custom and enter 640 and 480 in the left and right dimensions fields, respectively.**

   (c) **Scale: 1.0.**

   (d) **Check Remove Temporary Files.**

   (e) **Check Save to File, if you wish to save the playblast.**

   (f) **Press the Browse button.**

   (g) **Navigate to where you want to save your movie and name it.**

7. **Press Playblast.**

You can interrupt a Playblast at any time by pressing the ESC key.

While the playblast is being created, resist the temptation to click on windows or leave Maya to check your e-mail. To make a playblast, Maya takes screen captures and then puts them together in one file. If you cover up the scene view with another window, that is what you'll see in the playblast. When it's finished, the movie should

open in the movie player that you specified. If you checked "Save" in the playblast options window, you can locate the file on your computer and play it when you wish.

Summary

This chapter introduced the Dynamics module which calculates the effect of physical forces in a scene. Forces exist as impulses and initial velocities stored as attributes for dynamic bodies, as contact or collision forces, and as fields, of which there are a number of different types. Turbulent fields are useful for creating motion resembling the random diffusion seen in a number of natural phenomena, including molecular-scale Brownian motion.

Dynamic model types include rigid bodies, soft bodies, and particles (Maya Unlimited ships with two additional types, Fur and Hair; we won't have an opportunity to explore those in this book). A soft body uses particles to model deformations of geometric surfaces caused by forces. Rigid bodies behave like solid objects, with fixed distinct geometries; they tumble, bounce, roll, and slide in response to force fields and collisions but do not change shape. Unlike rigid bodies, particles are points in space. A particle has mass, but no shape. While this limits their usefulness for simulations where shape is a factor in collisions, and rotational motion matters, particles offer a processing speed advantage over rigid body dynamics. Furthermore, the extensive list of particle attributes and the variety of render types make particles an indispensable tool for 3D visual effects.

Utilizing Maya's built-in dynamics capabilities where possible reduces the amount of custom physics programming you need to code.

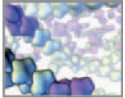
In the first of two tutorials, you created a rigid body collision simulation involving a force field and two objects, one active and the other passive. Passive objects are unmoved by fields but are implicated in collisions with active objects. Collision dynamics are governed by a `rigidSolver` node, which has attributes that control the speed and sensitivity of collision calculations. You saw that memory caching is a useful way to temporarily store dynamic data and disable calculations, for predictable playback of your simulation.

In the second tutorial you learned that particle behavior is governed both by the emitter and by the particle object. Emitter attributes determine the birth rate, initial speed, and direction of particle, while particle attributes set their lifespan, maximum count, and responses to fields and collisions. Dynamic connections between particles and geometry are made in the scene graph by `geoConnector` nodes. By default, individual particles within a particle object are not set up to interact with one another. You bypassed this limitation by emitting a force field from each particle to simulate inter-particle collisions. You then created a per particle attribute, `rgbPP` and connected it to another attribute, force, using a custom expression. This setup provided a simple example of real-time data visualization, and how Maya Dynamics and custom programming can work hand in hand.

Finally, you made a preview rendering of your scene using the playblast function. In the next chapters, you will learn how to create shaders, set up lights and cameras, and make finished renderings of your animations. That will complete your foundation of Maya basics and you're ready for a well informed approach to MEL programming.

This page intentionally left blank

08 Shading



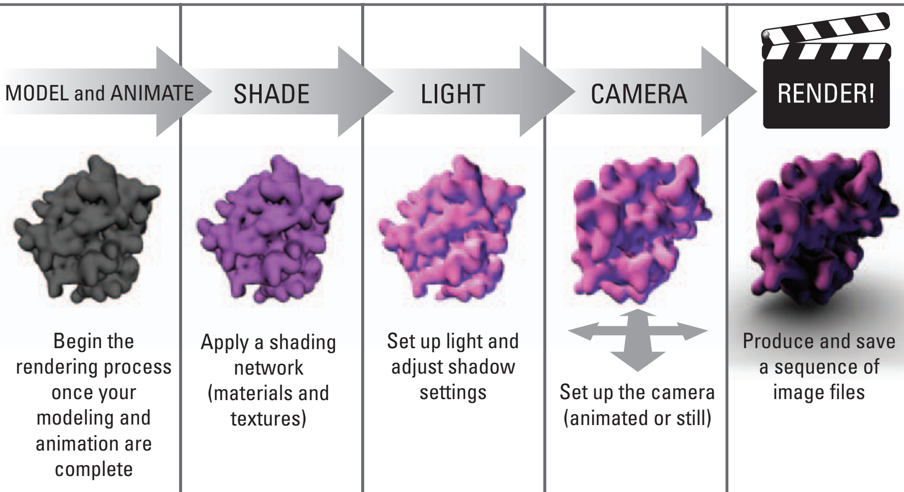


FIGURE 08.01

The rendering process. A shading network defines the various properties that determine how a surface will respond to light. Attributes of the shading network, of the lights, and of the camera are processed by a render engine to produce one or more digital image files—the "rendering".

Introduction

We saw in *Chapter 3* that rendering is the process of turning a scene into a picture or series of pictures (or frames) once the modeling and animation is completed. It is the final stage in the 3D computer graphics production workflow, before postproduction steps like compositing (the process of compiling rendered footage and adding special effects to produce a finished video). Rendering involves a shading network, lights, a camera, and one of Maya's rendering engines (or **renderers**). Figure 08.01 shows the rendering process in schematic form. Because Maya was developed for the entertainment industry, where image quality is paramount, the software has extensive capabilities for each stage of the rendering process, allowing you to achieve a wide range of visual effects in your projects. However, the many options available are only advantageous if used smartly and with purpose. It is easy to become bogged down experimenting with countless attributes and observing their effects on the rendered image. We suggest you always begin with an idea of what you would like the rendering to look like—its visual style—and work with the tools to achieve that look, or something close to it.

In this chapter we'll tour the big topic of rendering so as you complete each in silico project you'll be ready to give them a polished professional look. In the tutorials you will learn how to create a shading network, set up lights and a camera, and render a short (three seconds) animation using the Software Renderer and the Hardware Render Buffer. Don't be fooled by the amount of material we've collected in these pages, though—this is just an introduction to the subject! But we will equip you with the essentials, so that you'll know how to shade, light, and render your projects at an introductory level. Once you've mastered this content, we invite you to check out the titles listed under the heading *Rendering* in the *Further reading* section for additional tips and techniques. As well, the Help references listed throughout the chapter will point you to more detailed, topical information in Maya's Help Library.

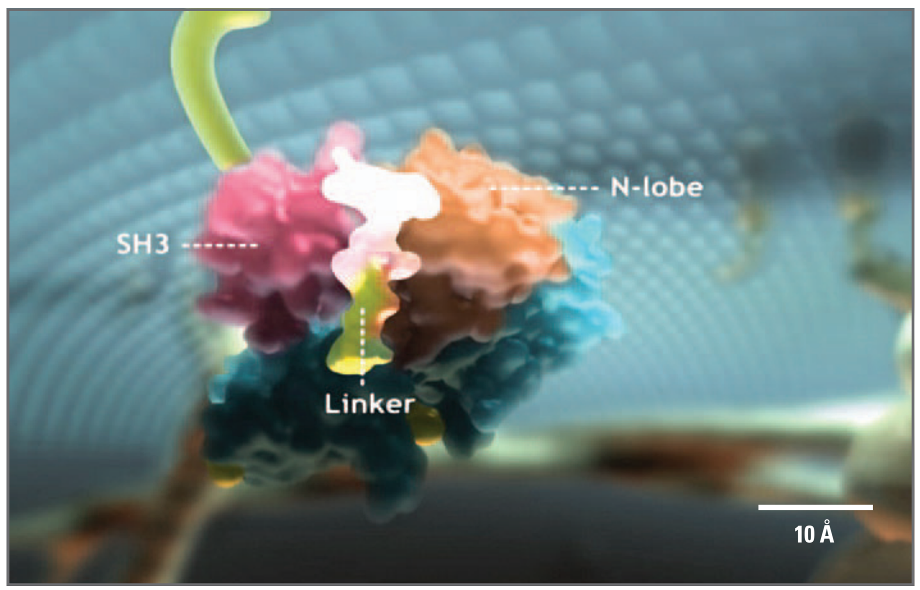


FIGURE 08.02

For this rendering of the protein kinase CSrc, animator Eddy Xuan created the illusion of translucence using a feature called **subsurface scattering** and the **mental ray for Maya** renderer.

Courtesy and copyright © 2006 Eddy Xuan.

A matter of style

Rendering is based on Maya's computer model of how light will interact with each surface, particle, or volume in the scene, as observed from the camera's position. Represented mathematically as rays and beams of illumination, the rendering algorithm works out the color and illumination falling onto each pixel of the simulated camera image. Since you can tune many of the parameters defining the interaction of light and surface, such as the color of the inbound light, the color absorption properties of the surface, and so on, you have enormous flexibility in crafting the rendered image.

Choosing an appropriate rendering style often depends on the end use of the image(s). For example, animation used in a television broadcast science program might call for a certain degree of photorealism in the renderings (Figure 08.02); an increasingly sophisticated lay public has come to expect a high level of verisimilitude in synthetic portrayals of biology. On the other hand, a rendering of a protein dynamics simulation for an audience of biochemists might leverage the benefits of **non-photorealistic rendering** (**NPR**) in order to reduce visual clutter and stress specific data. A hybrid style—which leverages the didactic advantages of NPR and the esthetic familiarity of photorealism—could well be used in animations designed to instruct students in biochemistry (Figure 08.03) or explain to patients how a drug treats disease. However, these stylistic suggestions are merely a few possibilities for you. Ultimately, the choice of rendering style depends on the interests and goals of the individual artist, and the communication requirements of each of your projects. The process of creating effective visual communications has yet to be distilled down to a set of hard, fast rules (thank goodness!).

The economy of rendering

The choice of rendering style is also a matter of economy. Generally, greater visual complexity means longer rendering times, which is a hindrance in almost any production

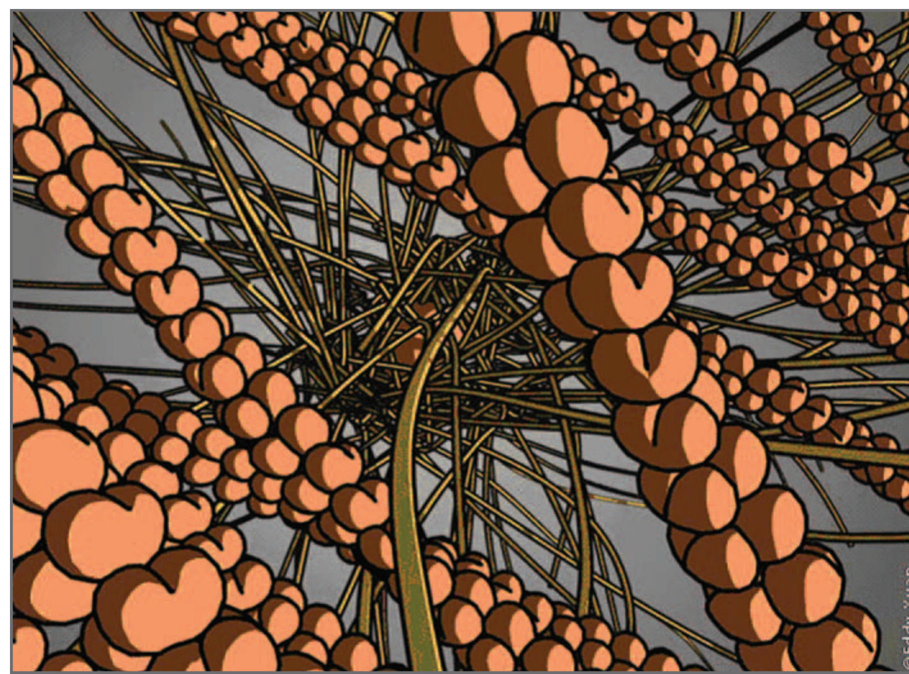


FIGURE 08.03

NPR cartoon outlines were used to highlight key structures in this animation depicting cross-linked structural proteins (actin filaments) within a cell. One filament is approximately 1 nm in diameter.

Courtesy and copyright © 2006 Eddy Xuan.

Graphics processors make use of parallelism, whereby multiple processing tasks are divided and handled concurrently, rather than sequentially, the way single CPUs deal with them. Software developers are increasingly looking for ways to exploit this capability for tasks other than image processing.

environment—be it a laboratory or a commercial studio—where time is money and where money must be expended to secure access to the computer resources needed to compute the render frame by frame through the animation. Your goal with rendering is to realize a balance between speed and quality that is economical for the given project. A factor in the speed/quality equation is the choice of rendering technology. **Hardware rendering**, in which calculations are performed on the graphics processor of your computer's video card, is generally quicker than **software rendering**, which uses software algorithms that are processed on your main computer processor. The caveat to this is that software rendering in Maya can create many effects that aren't yet possible with hardware rendering. However, this is starting to change. There has been considerable research of late into ways to better leverage video graphics processing power. With each new release, Maya's hardware rendering capabilities improve. A rule of thumb is: if hardware rendering will suffice, use it.

The Render menu set

The Render menu set for Maya Unlimited includes Hair and Fur.

The Hypershade is a good item to have in your custom shelf for quick access. If you plan on rendering your projects, you will use it a lot.

Like other menu sets, this one can be selected from the menu at the far left of the Status Line. The Render menu set includes Lighting/Shading, Texturing, Render, Toon, and Paint Effects. Many of the functions in these menus can also be accessed using buttons in the Render shelf, through the **Hypershade** editor, which is used to create and edit shading networks, and in the **Render View** window, which we use often to make preview renderings (more on that shortly). The Render menu set and other UI features you'll deploy in the rendering process are labeled in Figure 08.04.

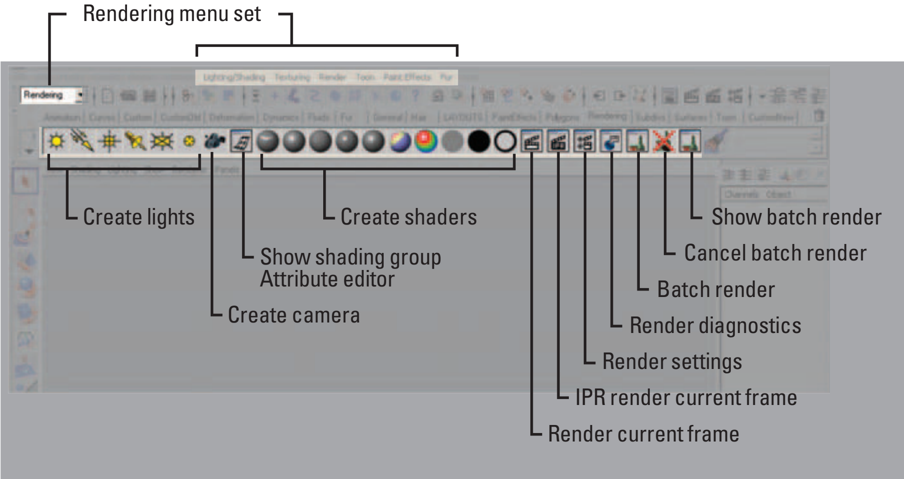
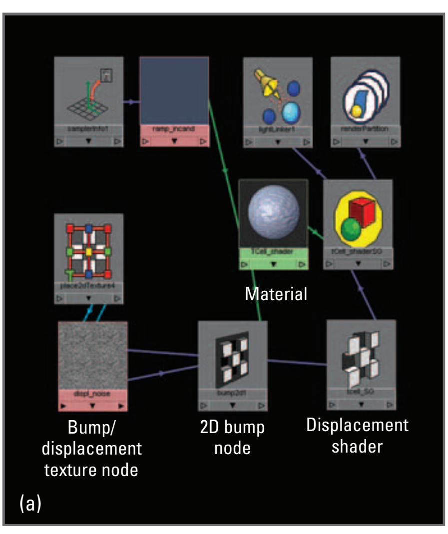


FIGURE 08.04

Maya user interface items specific to rendering.



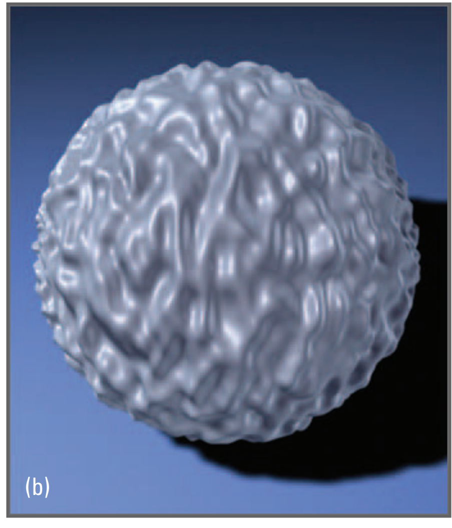


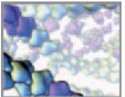
FIGURE 08.05

(a) The Hypershade view of a typical shading network involving a material, a bump map, and a displacement map.

(b) A NURBS sphere rendered using the shading network from (a).

Shading

Usually, the first step in the rendering process is **shading**, in which you assign colors and textures to items in your scene. You do this by creating **shading networks**—groups of connected **render nodes** which determine surface properties of your models' interaction with light, including color, transparency, and relief (or bumpiness)—and connecting them to geometry and other entities, like particles. Shading networks are like little recipes for how to transform incoming lighting to outgoing lighting. Figure 08.05a shows a typical shading network displayed in the Hypershade and in Figure 08.05b, we see the rendered view of that shader applied to a sphere, making it appear randomly bumpy. A basic shading network is automatically set up when you create a material



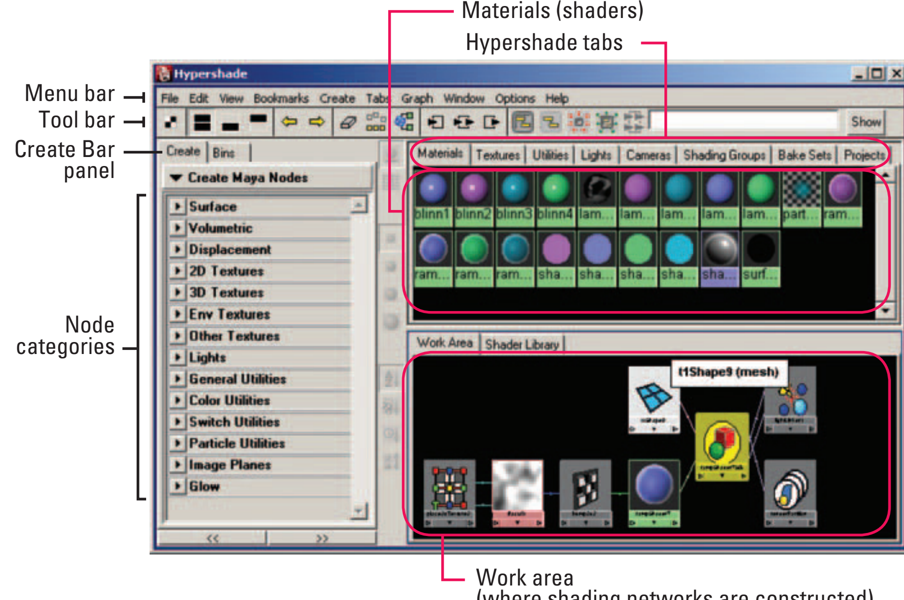


FIGURE 08.06

The Hypershade is Maya's tool for creating and editing shading networks.

and assign (or connect) it to an object in your scene. As with other features in Maya, it is possible to work with materials and textures without being aware of the underlying architecture of nodes in the shader network and their connected attributes. However, an understanding of how to *build* shading networks will enable you to create sophisticated visual effects easily. Maya makes it convenient to work with shading nodes directly, using the Hypershade.

The Hypershade

The Hypershade will soon be one of your best friends in the Maya world. Like the Hypergraph, it displays dependency relationships—not just for render nodes, but for any item in the scene graph. Its major features are labeled in Figure 08.06. We will explore them further in the first tutorial. To open the Hypershade:

Choose Window → Rendering Editors → Hypershade

Render nodes

Render nodes are DG nodes that can be interconnected to create diverse visual effects. One of the advantages of Maya's Dependency Graph architecture is its flexibility. Render nodes can be connected to other types of nodes to drive their attributes and vice versa. For example, a texture pattern (from a render node) could be used to control particle emission and particle age could, in turn, drive the color of a surface!

Material	Description	Sample rendering
Anisotropic	Used to render reflective surfaces with fine grooves, such as satin fabric or brushed metal. Attributes control the direction of grooves.	
Blinn	Used for simulating metal or glass surfaces, it produces high-quality, isotropic specular highlights. Named for Jim Blinn who originally developed the shading algorithm.	
Lambert	Used for matte surfaces with no specular highlight. Named in honor of the German physicist Johann Lambert (1728–1777).	
Phong	Used to represent glossy surfaces, such as hard plastic, with isotropic specular highlights. Provides less control over highlights than the Blinn material but is quicker to render. Named for Phong Bui-tuong.	
Ramp	Color gradients are used to control common material attributes such as color and transparency. Specular highlights are optional.	

TABLE 08.01

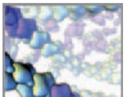
Commonly used Maya materials.

Materials

We have seen that rendering in Maya is a kind of simulated photography. As in real photography, there will be no image unless simulated light from your Maya world reaches the simulated lens of your simulated camera. So there will be no image unless you supply simulated light to scatter from the objects in your scene, as well as simulated physical surfaces on each model for the light to bounce off or radiate from. Materials are Maya's computer model of these light-scattering properties of each object's surface.

All materials native to Maya are of three types: surface, volumetric, displacement. Surface materials apply to any NURBS or polygonal surface in a scene. Volumetric materials determine the appearance of 3D volumes rather than surfaces. Fog and cloud type particles are examples of such volumetric entities in Maya. Finally, displacement materials change the topography of rendered objects to produce surface relief (as in Figure 08.05b). By default, Maya has one displacement shader, to which you connect other nodes to drive its attributes. There are many more material types, which are for use only with the mental ray for Maya renderer. They are described in the mental ray Shaders Guide.

A **material node** contains basic appearance attributes that are common to all materials. These include color, transparency, and incandescence, among others. Additional attributes exist which are unique to specific shaders. For example, not all materials have attributes to control surface **specularity** or *shininess*. Table 08.01 lists five of the more commonly used Maya material types, with rendered examples.



You can create a material node in the Hypershade in one of three ways:

1. **Select the appropriate swatch in one of the Surface, Volumetric, or Displacement panels under the Create Bar (see** Figure 08.06**)**

or 2. **Use the Create Render Node window as follows:**

   (a) In the Hypershade menu bar, choose Create → Create Render Node.
   (b) Click on the Materials tab and open the Surface, Volumetric, or Displacement Materials panel
   (c) Press the button of the material you want to create.

or 3. **Select the material by name in the Create menu.**

Materials

Maya Help → Using Maya → Rendering and Render Setup → Shading → About shading and texturing surfaces → Maya materials → Surface, displacement, volumetric

mental ray Shaders Guide

Maya Help → Using Maya → mental ray → mental ray Shaders Guide

Texture nodes

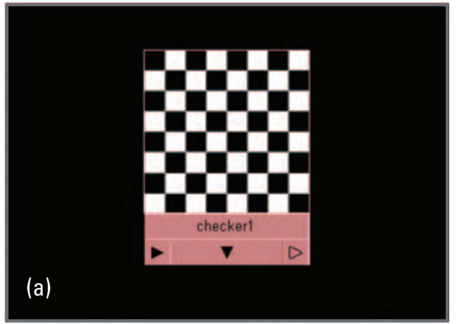
To Maya, a texture is an **image** that defines some property of a surface, such as a color or transparency pattern, or relief (elevations and depressions). **Texture mapping** describes how the texture image relates spatially to the surface: how a point in the texture relates to a point on the surface. When you map a texture, you connect its output—a color or **alpha channel**—to a **channel** (or attribute) of a material. In computer graphics literature, checker patterns are commonly used to illustrate how textures might be applied to geometry, as shown in Figure 08.07. For the purpose of rendering, texture mapping determines how a texture image will be oriented in relation to, and deformed with, a surface. In addition to defining rendering properties like color and transparency, textures are also used to control events on a surface, such as particle emission rate, as shown in Figure 08.08.

Common texture maps

Maya Help → Using Maya → Rendering and Render Setup → Shading → About shading and texturing surfaces → Mapping and positioning textures → Texture mapping

Use a texture to scale particle emission rate

Maya Help → Using Maya → Dynamics and Effects → Dynamics → Particles → Work with emitters → Use a texture to color emission or scale the rate



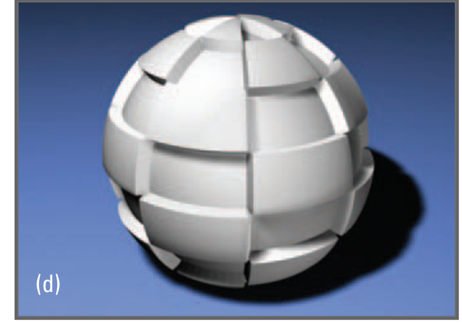


FIGURE 08.07

A checker texture node (a) mapped to a material's (b) color channel, (c) transparency channel, and (d) the shading group node's displacement channel.

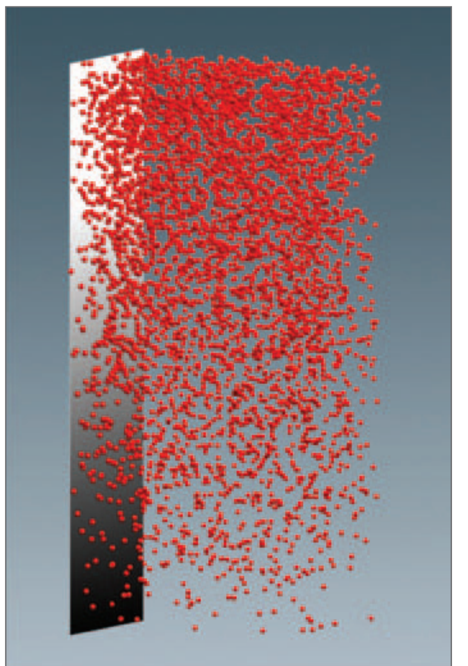


FIGURE 08.08 (Left)

A texture mapped to a surface can be used to drive attributes such as particle emission rate (pictured here). The emission rate spans a range of 0 (particles per second), where the texture is black, to 1000, where it is white.

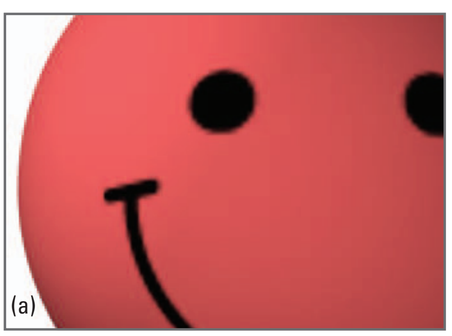
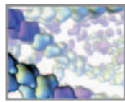


FIGURE 08.09 (Right)

The resolution of a bitmap image used in a file texture impacts the quality of the finished rendering. In this example, a bitmap image was used in the color channel of a lambert material. In (a) we used a low resolution, 256 × 256 pixel image. The image used in (b) was higher resolution: 1024 × 1024 pixels.



There are four types of Maya texture nodes, called 2D, 3D, Environment, and Layered textures. 2D textures map onto two-dimensional surfaces, like wrapping paper on a parcel, whereas a 3D texture fills a volume, projecting through an object like veins through marble. Environment textures are used for backgrounds or to create reflections. Layered texture nodes are used to combine the effects of two or more 2D textures.

Most 2D, and all 3D, textures in Maya are procedural. A **procedural texture** is a 2D or 3D plot of a mathematical function, and is therefore resolution-independent, meaning that the image it computes is always at the correct resolution for the distance of the textured object from the camera. In contrast, a **file texture** is a 2D texture node that references a bitmap image file of finite size into Maya. A file texture could be a scan of a photograph or illustration, a digital photograph, or a digital illustration or painting. The size of a bitmap, often described by its width and height in pixels, is called its **resolution**, and impacts the quality and speed of rendering. You want a file texture bitmap image to have as low a resolution as possible, for fast processing, but not so low as to cause blurring or **pixelation**—the visibility of blockiness or pixels on the surfaces of your objects. Figure 08.09 shows the difference between high- and low-resolution bitmaps used in a file texture.

Procedural textures are generally used to create regular and abstract patterns, whereas file textures are used when specific details are necessary. Take a model airplane, for example. Camouflage markings could be created with a procedural texture, whereas the decals—letters, numbers, and illustrations—would require a file texture. Because procedural texture images must be calculated, they will impact rendering times. It is sometimes desirable, therefore, to convert procedural textures into file textures which are generally faster to render.

Because a procedural texture is controlled by attributes (acting as the variables of the relevant mathematical function in the Maya rendering software), it can be made to change over time by animating those attributes. Furthermore, many attributes can be connected to, and therefore driven by, other texture nodes, be they file textures or procedural textures. Such texture networking lets you to achieve compound procedural effects with textures.

Understanding and working with Maya texture nodes

Maya Help → Using Maya → Rendering and Render Setup → Shading → About shading and texturing surfaces → Shading networks → About shading networks

Sprites, a hardware-rendered particle type, are 2D squares that always face the camera and act as placeholders for a file texture.

A file texture can be animated as well, but requires a bitmap image for each frame of animation. An animated file texture was mapped onto particle sprites to create the animation of HIV particles (or **virions**) shown in Figure 08.10. One virion was modeled and animated through a full rotation and rendered out in 210 frames (seven seconds of animation). The frames were then loaded into a file texture which was assigned to the particle object. Rendering the textured, flat sprites was considerably faster than rendering a scene full of geometry. With sprites, Maya doesn't need to compute the interaction of light with the topography of hundreds of separate 3D virus models. For this camera distance, one sprite fits all!

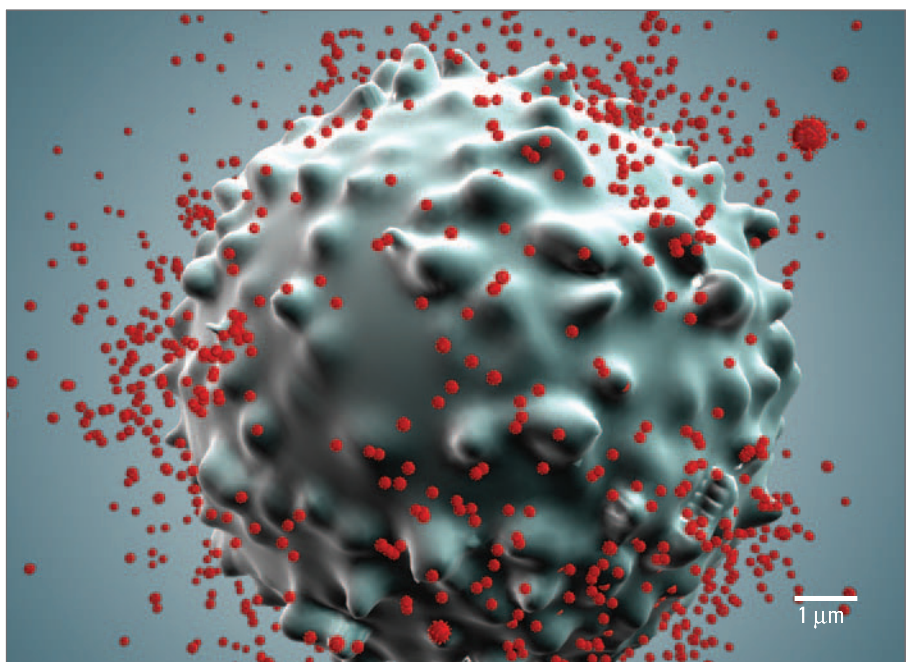


FIGURE 08.10

Rather than generating many small virions as geometry, an animated file texture was mapped to particle sprites (flat planes that always face the camera), creating the illusion of hundreds of tumbling HIV viruses in this animation showing infection of a white blood cell (T-lymphocyte).

Courtesy and © 2007 AXS Biomedical Animation Studio.

Assign an animated file texture to sprites

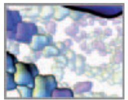
Maya Help → Using Maya → Dynamics → Particles → Work with advanced dynamics → Assign image sequences to sprites

Ramps

A ramp is a color or grayscale gradient. The gradient has one or more component(s), each defined by a position value between zero and one, and a color or grayscale value (Figure 08.11). A ramp can be either a “U” or a “V” ramp, referring to its direction when applied to a surface (we take a closer look at the UV coordinate system below). Ramps and their components can be connected to other nodes to drive attributes using color. Conversely, other nodes can be used to drive ramp colors.

Maya has a **texture ramp** node and a **ramp material** node which are use to apply gradients to object colors and other rendering attributes like transparency and incandescence.

Maya Help → Using Maya → Rendering and Render Setup → Shading → Shading Nodes → Texture nodes → 2D textures → Ramp



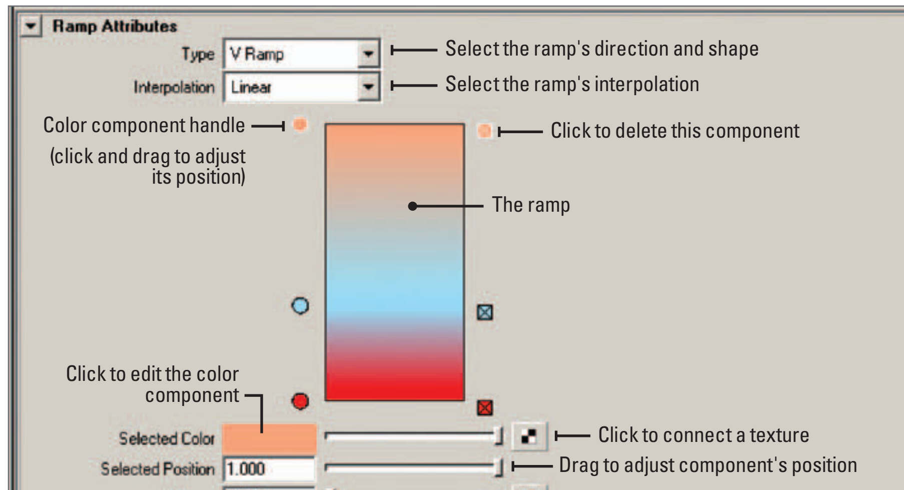


FIGURE 08.11

Ramps are color and grayscale gradients that can be used to drive rendering and other attributes.

Ramp material

Maya Help → Using Maya → Rendering and Render Setup → Shading → Shading Nodes → Material nodes—Maya Software → Surface Materials → Ramp shader

Bump and displacement maps

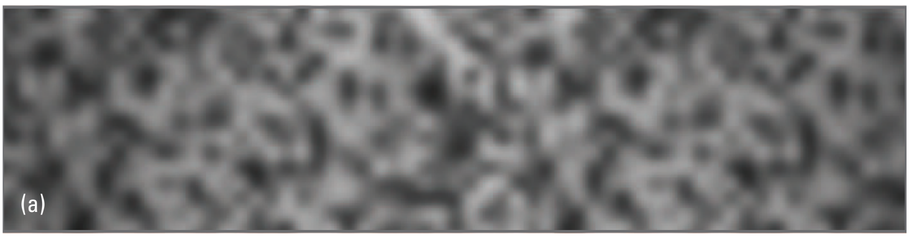
Because bump and displacement maps are applied at render time, they don't slow down your interaction with the Maya scene as you work on it.

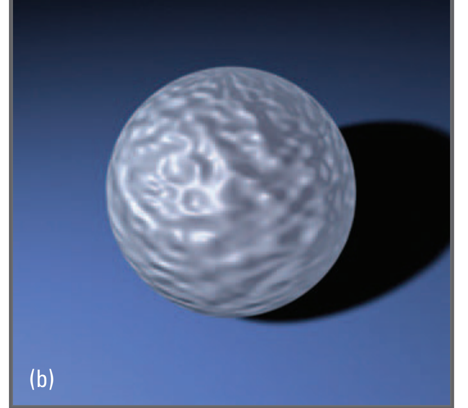
Bump maps and displacement maps (Figure 08.12) use grayscale procedural or bitmap textures to create surface relief at render time. This allows you to make alterations to surface topography that would be difficult if not impossible to model via conventional NURBS or polygon tools. Bump maps create the illusion of relief by altering the directions of surface normals, and therefore changing the way the light interacts with a surface. In contrast, a displacement map actually changes the topography at render time by adding depressions and elevations. Where necessary, you can adjust the **tessellation**—the degree to which a surface gets subdivided upon rendering—of an object, creating enough additional polygons to properly model the relief.

Bump maps are best for shallow relief, such as the small undulations on the surface of a cell. Because a bump map displaces normals, and not actual polygons, it cannot produce relief along the edge, or profile of an object, a fact that can be seen in Figure 08.12. Displacement maps are best for deep relief and when displacement of an object's profile is desired such as with the extension of cell processes like pseudopodia. Both bump and displacement maps can be animated to make surface topography change over time. Finally, bump maps render faster than displacement maps. Deep, detailed displacements can dramatically increase rendering times.

Bump maps and displacement maps

Maya Help → Using Maya → Rendering and Render Setup → Shading → Surface relief → About surface relief





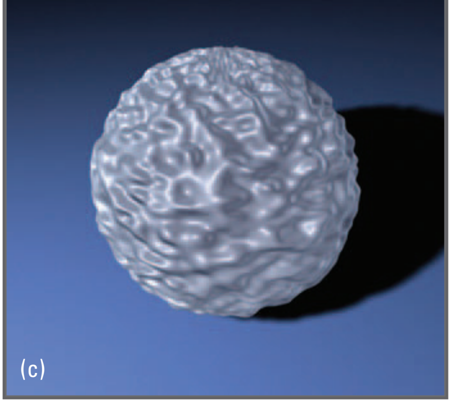


FIGURE 08.12

Bump and displacement maps allow you to alter surface topography at render time. The same grayscale procedural noise texture (a) was used to create relief with (b) a bump map and (c) a displacement map. Note that displacement alters the profile contour, whereas the bump map does not.

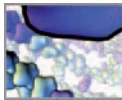
Black and white

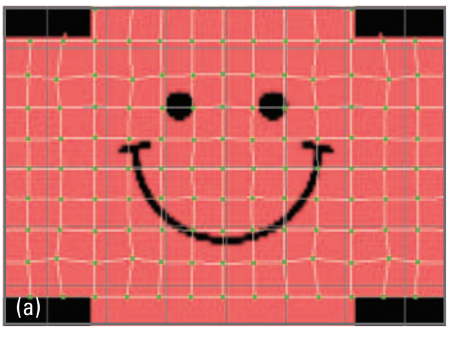
When working with textures, the color value range between black and white corresponds to a numerical range between 0 and 1. In a transparency map, black is zero transparency (or opaque), white is fully transparent, and gray values are semi-opaque. In bump and displacement maps, black is even-ground, and white, the maximum elevation.

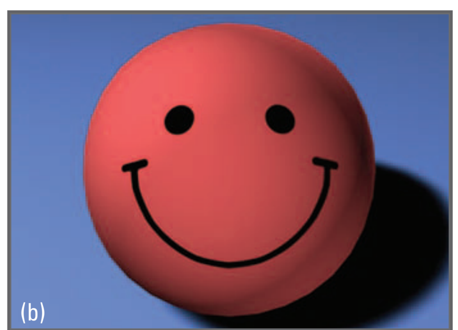
UV coordinates: life on the surface

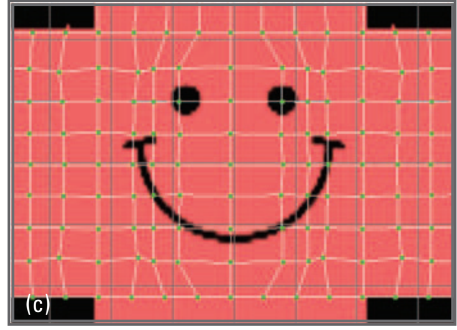
In addition to the XYZ local and world coordinates, Maya has a 2D coordinate system that it uses to map textures onto surfaces. U and V are analogous to X and Y on a flat surface, except that the U and V axes wrap around and deform with an object's surface. When you assign a texture to a surface (through a material channel such as Col or) it automatically maps to the surface's UVs; every location in a 2D texture image has a corresponding UV location.

When you create a primitive model, Maya automatically generates UV coordinates for the surface. Maya has an extensive toolset, available through the **UV Texture Editor** (**UTE**), to edit the way textures map to complex surfaces. Figure 08.13 shows the UTE displaying a file texture applied to a polygonal object. UV control points (or simply "UVs") can be manipulated individually or in groups. When you move a UV, it maintains its original position relative to its surface geometry, but changes position relative to the mapped texture. The net effect is that the texture changes position or shape relative to the surface geometry. Figure 08.13 illustrates the effect on texture placement of manipulating UVs.









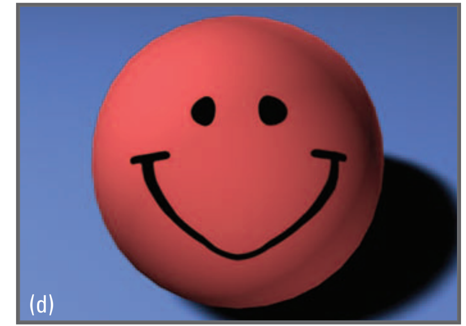


FIGURE 08.13

The position of UVs, relative to a texture image, determines how the image will appear on 3D geometry. Pictures (a) and (c) show the workspace of the UTE.

(a) UVs (green dots) in their default positions relative to the texture image.

(b) A rendering made with the default UV placement.

(c) UVs transformed relative to the texture image.

(d) A rendering made with the transformed UV placement.

Texture mapping (or **UV mapping**) is an important step in the character animation workflow for gaming and movies. It defines how elements like clothing, skin, and facial features appear on models when they are rendered. In biomedical and in silico animation, the default mapping is often sufficient for abstract procedural textures. There are, however, occasions in cellular and molecular animation that call for repeating patterns—fibrous proteins, for example—in which case editing texture placement becomes important.

The UTE

Maya Help → Using Maya → Modeling → Mapping UVs → UVs windows and editors reference → UV Texture Editor reference

Texture placement nodes

When you assign a texture to an object, Maya creates a texture placement node, which maps the texture to the object's UVs. This node gives high-level control over the mapping, allowing you to tile, rotate, and mirror the texture on the surface.

Working with texture placement nodes

Maya Help → Using Maya → Rendering and Render Setup → Shading → Mapping and positioning textures → 2D and 3D texture positioning

Shading engine nodes

All render nodes in a shading network—textures, placement nodes, displacement and bump map nodes, and materials—converge to a **shading engine node** (also called a **shading group**, or **SG node** for short). An SG node evaluates the other render nodes, along with lights and surface topography, to determine the rendered appearance of the surfaces to which it's attached. For any surface to be rendered, it must be connected to an SG node. However, you need not create and connect an SG node manually; it is done for you when you assign a material to an object.

Preconfigured Maya shading networks

In addition to building a custom shading network, you can load a preconfigured network which was created by a third party developer, or which came bundled with Maya in the **Shader Network Library**. To load a shading network from (a) a third party or (b) from the Maya Shader Network Library:

1. (a) Download or copy the shader network file (a Maya binary or Maya bitmap file) to your local drive. Place it in directory where you can easily locate it such as the Textures directory for your current Maya project.

   (b) Locate the directory in which the Shader Network Library was installed (if it wasn't installed with the main Maya installation, install it now). Within that directory, navigate to the shader network file (a Maya binary or Maya bitmap file) that you wish to load and copy it to a directory where you can easily locate it such as the Textures directory of your current Maya project.

   If no Textures directory exists, you can create one by choosing File → Project → Edit Current, and then entering a directory name in the Textures field, followed by hitting Accept.

Installing the Maya Shader Network Library

Maya Help → Using Maya → General → Installation and Licensing → Installing the Shader Library → Installing the Maya Shader Library

2. Start Maya and open the Hypershade.
3. In the Hypershade, choose File → Import, then navigate to and select the shader network file you wish to load—either the third party shader file or the one from the Maya Shader Network Library.
4. Press the Import button.

Maya Paint Effects

Paint Effects is an FX and texturing painting module that allows you to draw strokes (colored line and patterns) in either 2D or 3D. You can use it to apply custom patterns and particle effects to geometry and curves in a 3D scene. Invented by Autodesk developer and computer graphics guru Duncan Brinsmead, Paint Effects is used widely in FX workflows for film and television to create organic elements such as trees, grass, and flowers, which can be animated to simulate growth. Biomedical Animator Drew Berry pioneered the creative use of Paint Effects in molecular

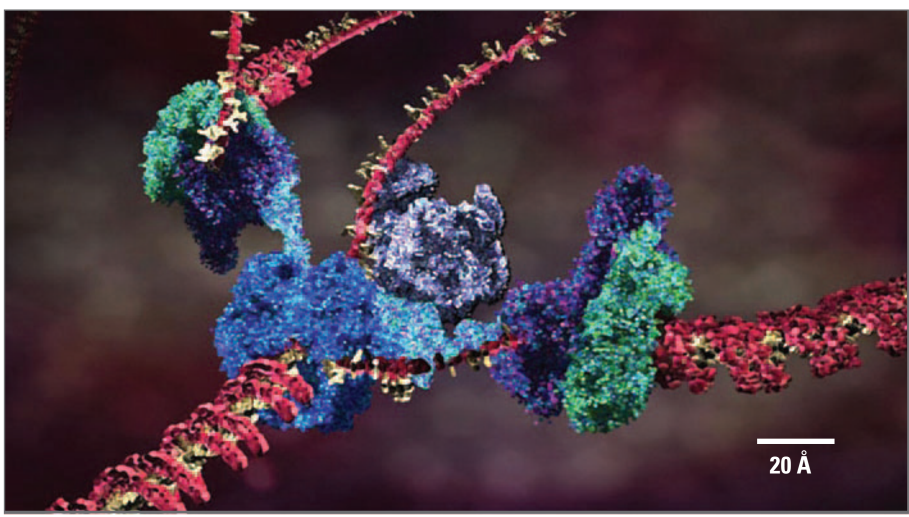


FIGURE 08.14

Animator Drew Berry used Paint Effects and particles to create striking visualizations of DNA replication and other biomolecular subjects. By painting particles onto globular models of proteins and DNA, he was able to achieve a high level of atomic detail and give a sense of thermal vibrations of molecules. Building, animating, and rendering at this level of detail using geometry instead of particles are computationally intractable with current desktop computers.

DNA Replication by Drew Berry, The Walter and Eliza Hall Institute.

interpretive visualization. He was named a Maya Master by Alias in 2005 for visualizations of DNA (Figure 08.14) that he created for a major multi-national project marking the 50th anniversary of the discovery of the DNA double helix.

Paint Effects overview

Help → Maya Help → Using Maya → General → Paint Effects and 3D Paint → Maya Paint Effects → What is Paint Effects?

Render View

The Render View is a window used to make preview renderings of single frames. Figure 08.15 shows the main UI Elements of the Render View window. The toolbar buttons are shortcuts to items available through the menus.

Previewing with IPR

IPR (**Interactive Photorealistic Rendering**) is an interactive rendering mode, used to **pre-visualize** your scene. It works in the Render View to update the rendered image automatically as you make changes to lights and materials. IPR is available when using the Software or mental ray for Maya renderer. While it doesn't support all of their features, it's a handy tool for tweaking your scene in preparation for a final rendering. You will use IPR for feedback on light positions in *Chapter 10*.

Render previewing with IPR

Maya Help → Using Maya → Rendering and Render Setup → Rendering → Visualize and render images → Rendering methods → Interactive Photorealistic Rendering (IPR)

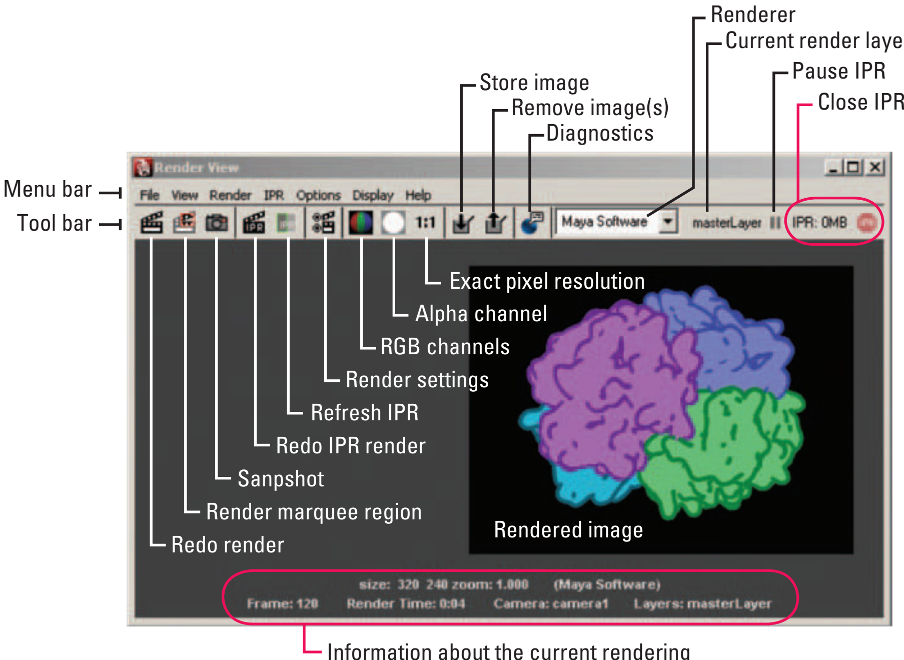


FIGURE 08.15

The Render View is an interactive rendering window. It is primarily for previewing, but can be used to make and save final single-frame renderings as well.

Keeping and removing images in the Render View

The `Keep Image in Render View` menu command (or `Keep Image` button) stores the current render preview in memory. When multiple images are stored, you can switch between them using the slider bar at the bottom of the Render View. You can also add a text comment to a stored frame in order to keep track of Render Settings. To add a comment:

RMB+click on the Keep Image button and choose Keep Image with Comment

Keeping images is a great way to see the effects of changes you make to your scene and it works for all renderers, unlike IPR which works only for Maya Software and mental ray for Maya. The `Remove Image` button (or `Remove Image in Render View` menu command) deletes from memory the image that is currently displayed.

Render View rendering

Maya Help → Using Maya → Rendering and Render Setup → Rendering → Visualize and render images → Rendering methods → Render View rendering

Tutorial 08.01: Shading

In this tutorial, you will create four basic shading networks, each with a different material color, and assign each one to a piece of polygonal geometry in a ready-made

We will explore molecular modeling and visualization and Protein Data Bank structure data files in more depth in *Chapter 14*.

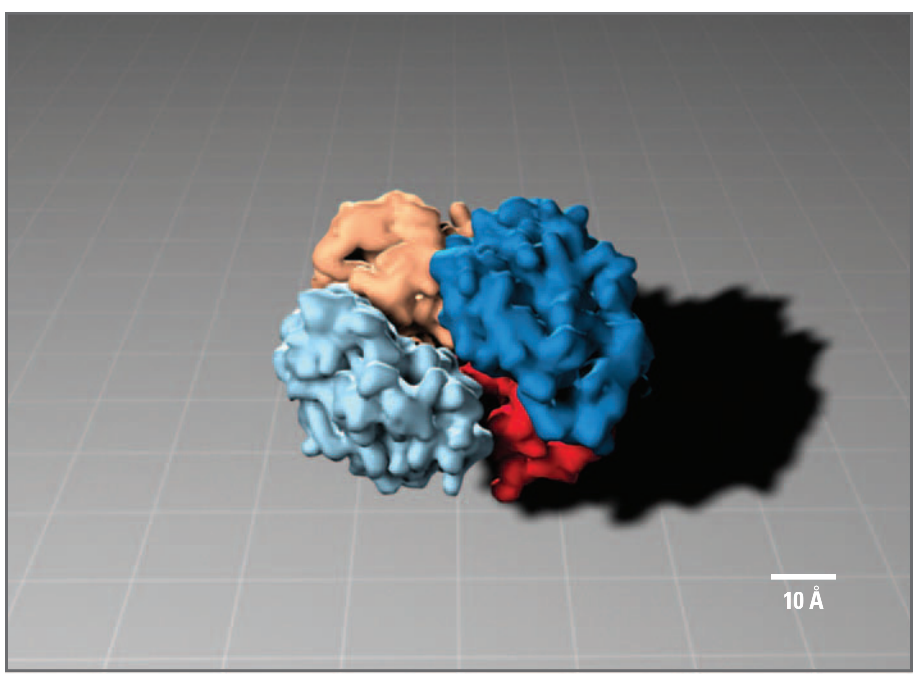


FIGURE 08.16

A final rendering of the scene you will shade in Tutorial 08.01. A lambert shader was applied to each object. The objects are the four **chains**, or subunits of a hemoglobin molecule. We modeled them using the UCSF Chimera package from the Resource for Biocomputing, Visualization, and Informatics at the University of California, San Francisco (supported by NIH P41 RR-01081).[1] The molecular structure data file used, 1buw.pdb, was procured free of charge from the RSCG Protein Data Bank.[2]

scene. Figure 08.16 shows the rendered result. To start, copy the scene we've created for you from the CD-ROM to your scenes directory:

08_Shading/scenes/tutorial_08_01.ma

Next, start Maya, set your project if necessary, and set your working units to 30 fps, Playback Speed to Real-time (30 fps), and Looping to Once. Open the scene file, tutorial_08_01.ma. This is a simple animation of the four subunits of **hemoglobin**—the oxygen-transport molecule found in red blood cells—coming together to form the complete molecule. A complete hemoglobin molecule contains amino acids, along with four atoms of iron which give blood its red hue. Our animation is not meant to be a realistic depiction of hemoglobin formation, which is a complex ballet of protein folding and bonding, but rather a simple visual scenario that highlights the molecule's overall shape and its subunit organization. A discussion of different types of molecular models follows in *Part 3*, *Chapter 14*.

The word "chain" here refers to a **polypeptide** subunit of the complete hemoglobin molecule. For more information on biomolecules and their components, refer to the *Further reading* section under the heading *Cell Science, Fundamentals*.

Create a surface material: A lambert shader

The file tutorial_08_01.ma opens with a perspective view of the scene. The hemoglobin subunits are polygonal models named chain1, chain2, chain3, and chain4. They are gray, the color of the default shader, lambert1. You can scrub the timeline to see the animation. Your first step is to create a shading network. We will then duplicate it three times and change the material color for each duplicate. The colors will help differentiate the subunits.

1. **Choose Window → Rendering Editors → Hypershade.**
2. **Click on lambert swatch in the Surface panel under the Create Bar. This creates a new shader called lambert2. (Refer to the labeled diagram of the Hypershade in** Figure 08.06**.)**
3. **Make sure both the top and bottom tabs are showing. If not, press the Show Top and Bottom Tab button .**
4. **Select the new shader in either the Materials (top tab) or Work Area (bottom tab) palette in the Hypershade and press the button to view the input and output connections.**
5. **If the render nodes are not visible in the Work Area, click in the Work Area and press the Show All hotkey, A.**

You can navigate (zoom and pan) the Hypershade view panels using the same key and mouse combinations you use to navigate Maya's scene view.

So far, the only node `lambert2` is connected to the SG node, `lambert2SG`. Each time you create a material using the method described above, Maya automatically creates an SG node and connects the material to it.

Common material attributes

Next, set the material's Color and other attributes. The settings we suggest are values chosen after experimenting with the final look. Feel free to experiment with them.

1. **Select the material, lambert2, by clicking on its icon in the Work Area or Material palette in the Hypershade.**
2. **Hit Ctrl+A or double+click lambert2 to launch the Attribute Editor.**

Color

This is the base color of the material.

3. **Under Common Material Attributes, click on the swatch for the Color channel (attribute). This launches the Color Chooser window (**Figure 08.17**).**

The default color system in the Color Chooser is **HSV** (**Hue**, **Saturation**, and **Value**). This color model allows you to vary the color by altering the essential tint (hue), intensity (saturation), and relative lightness or darkness (value). You can change this to the RGB (red, green, blue) color model if you like by selecting it from the menu in the bottom left corner of the Color Chooser. For now, leave it set to HSV.

4. **Make the following settings using the sliders or by entering them in the H, S, and V fields:**

   H: 350.0

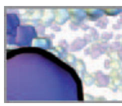
   S: 1.0

   V: 0.8

5. **Press the Accept button to close the Color Chooser.**

Using the Color Chooser

Maya Help → Using Maya → General→ Basics → Basics windows and editors → Color editor



The changes you make in the Color Chooser apply instantly. You don't need to hit Accept to implement them. The Color Chooser is like other windows in Maya; it displays content for the active item. If you select another color-type attribute, such as Transparency, the Color Chooser will load its current color setting.

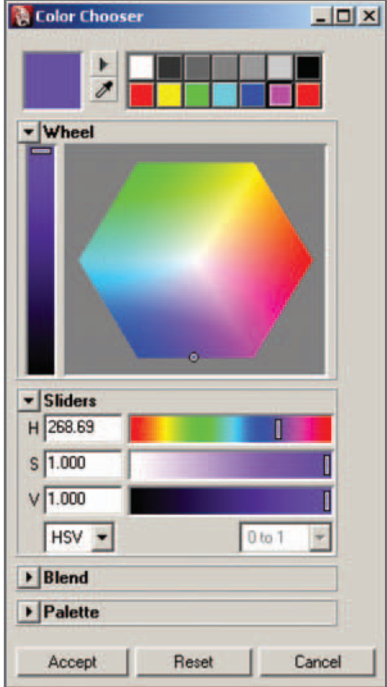


FIGURE 08.17

The Color Chooser allows you to specify colors in HSV or RGB mode.

Transparency

For the `Transparency` channel, black is equivalent to opaque and white to fully transparent.

Ambient Color

At its default setting of 0, Ambient Color has no effect on the material. If you increase its value toward one, its Ambient Color gets blended with the Color channel. Its contribution to the final color is influenced by the presence of Ambient lights in your scene.

Incandescence

Incandescence is self-emitted light. With non-zero incandescence values, it is possible to render a scene entirely without lights, using only the self-illuminated geometry. Incandescence affects only the object to which it's assigned. It doesn't cast light on other geometry in the scene.

Bump

The Bump channel uses a texture node to create the illusion of surface elevations and depressions at render time. Our geometry is quite bumpy to begin with, so we won't create a bump channel in this project.

Diffuse

The Diffuse channel controls a material's tendency to scatter light. In practical terms, it amounts to a brightness control. By default, it is set to 0.8.

Translucence

Translucence is the tendency of a material to absorb and scatter light beneath its surface. Skin, leaves, and wax are real-world materials that demonstrate this property. Translucence only works with a shadow-casting light and when raytraced shadows are turned on. In this exercise we're using depth map shadows instead, in order to keep render times down, so Translucence will have no effect.

The remaining lambert material attributes can be left at their defaults as well for now.

Attribute descriptions for all Maya surface materials

Maya Help → Using Maya → Rendering and Render Setup → Shading → Shading Nodes → Material nodes—Maya Software → Common surface material attributes

Assign the shading group to chain1

Below are two common ways to assign a shading group to an object. The simplest is to use the **material marking menu** in the Hypershade as follows:

1. **(a) Select chain1 by clicking on it in the scene view or on its name in the Outliner.**

   (b) With the Material palette visible in the Hypershade, RMB+click on the material you just set up, lambert2.

   (c) From the material marking menu, choose Assign Material to Selection (see Figure 08.18a**).**

or

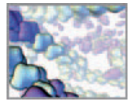
2. **(a) Position and/or resize the Hypershade so that both the Materials palette and chain1, in the scene view, are visible at once.**

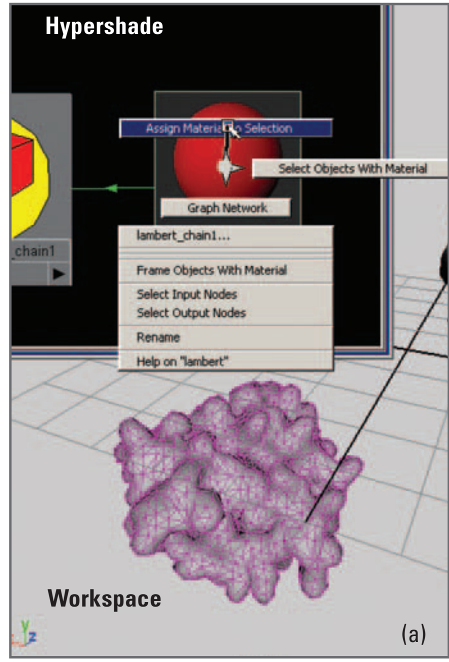
   (b) MMB+drag the icon for lambert2 onto chain1 in the scene view and then release the mouse button (see Figure 08.18b**).**

The last step connects chain1 to the shading group node, lambert2SG. View your scene in smooth shaded mode (type "5") to see the color applied to chain1.

Make and assign the remaining shaders

For this project, we want each hemoglobin subunit to have a different color. The audience then can see which is which after they've bound together. You have the choice of creating three new shaders from scratch, the way you did with lambert2, or duplicating lambert2 and with it any attribute settings you've already set up. Naturally, you will have to change the color attribute for each duplicate. Before you begin





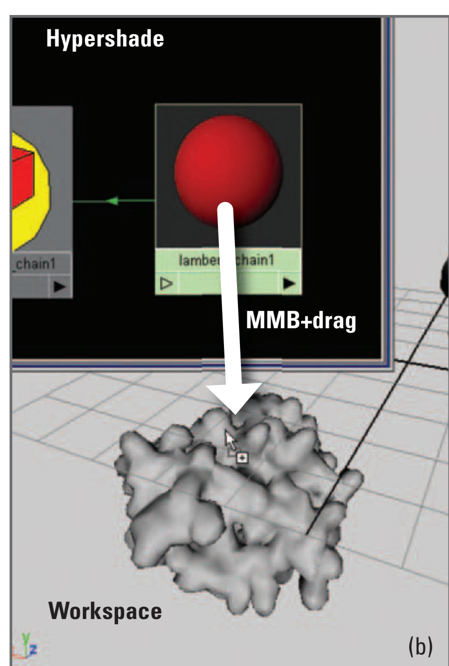


FIGURE 08.18

Two ways to assign a shading network to an object.

(a) Select the object, then use the material marking menu in the Hypershade.

(b) MMB+drag the material icon onto the object in the scene view.

duplicating or creating render nodes, let's consider the implications of how the nodes will be named.

Naming your render nodes

When you rename a material, the name of its shape node will update automatically, but not that of the shading group node—it must be renamed separately in the Channel Box or Attribute Editor.

When you duplicate a shader (by selecting the SG node and choosing Edit → Duplicate → Shading Network) all connected render nodes are duplicated. New node names are duplicates of the old, but incremented numerically. For instance, `lambert2SG` becomes `lambert2SG1`, then `lambert2SG2`, and so on. Likewise, the material name `lambert2` becomes `lambert3`. This default naming system can lead to confusion because the shading group name is different from the material name, which is completely different from the object you intend to shade. When working with many shaders, often the quickest way to select a render node according to the object it shades is to locate its tab in the object's Attribute Editor. However, being able to select render nodes by name in the Hypershade is preferable, since that is where you will be building and editing shading networks. For this reason, we find that setting up a logical scheme for node naming is a smart workflow decision. We suggest you avoid the default naming routine for all but the very simplest projects.

You may need to dolly and pan to adjust your view in the Hypergraph in order to read the names on the shading group swatches. If Popup Help is enabled in Preferences, a shading group name will appear when you place your mouse over its swatch.

Renaming a shading network correctly requires renaming both the material's transform node and the SG node. To rename the render nodes you created:

1. **Open the Attribute Editor for lambert2 by double-clicking on its icon in the Hypershade.**

2. **Enter lambert_chain1 in the name (lambert) field.**

3. **In the Attribute Editor, press the Output Connections button [icon] to reveal the lambert2SG tab.**

4. **Enter lambertSG_chain1 in the name (Shading Engine) field.**

The names we're using will take advantage of Maya's automatic naming strategy; as you duplicate nodes, their names will reflect the number of the hemoglobin chain they are to shade (i.e. chain2, chain3, and chain4).

Duplicate the shading network

It is advantageous to duplicate a shading network instead of creating it from scratch if you have made attribute settings for the original and you want them carried over to the new network. In the Hypershade:

1. **Choose the Shading Groups tab and select lambertSG_chain1.**

2. **Choose Edit → Duplicate → Shading Network.**

3. **Repeat step 2 two more times to create networks for chains 3 and 4.**

Adjust the color attributes

At this point all shaders have the same values in their color channels. We suggest you set these color settings, then experiment further once you've seen the final rendering:

lambert_chain2	lambert_chain3	lambert_chain4
H: 200.0	H: 20.0	H: 200.0
S: 1.0	S: 0.5	V: 1.0
V: 0.7	V: 1.0	V: 0.9

Apply the new shaders to the remaining three objects the same way you did for chain1. In smooth shaded mode, your scene view should look similar to Figure 08.19 (shown at frame 50 of the animation).

Add a textured background plane

In the next chapter and tutorial, you'll add shadow-casting lights to the scene. Cast shadows help greatly in the perception of spatial relationships. Currently there is no background to catch shadows in the scene, so let's add a polygon plane to do the job. To the plane, you can then add a repeating grid texture that gives a frame of reference for the viewer. Without it, there is nothing in the scene to indicate what, if anything, is standing still relative to the camera and geometry.

Create and position the plane

1. **Choose Create → Polygon Primitives → Plane ❐.** Turn off Interactive Creation if it isn't already off.

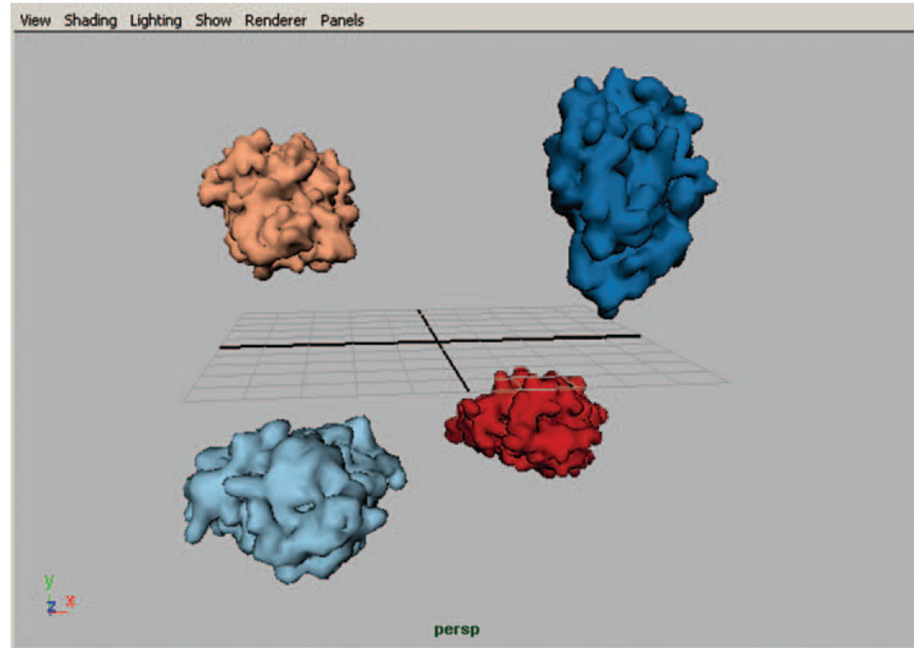


FIGURE 08.19

The scene at frame 50, after shaders have been assigned to each piece of geometry.

2. **Enter the following attributes:**

   Width: 1200

   Height: 1200

   Subdivisions Along Width/Height: 10

   Axis: Y

3. **Press Create.**

This makes a polygonal object called `pPlane1`. Next, move `pPlane1` below the other geometry to avoid intersections, and backwards in Z so that more of it is visible to `camera1`.

4. **Select the plane in the Outliner.**

5. **Enter the following attribute value in the Channel Box:**

   Translate Y: -100

Assign a material to the plane

The plane was just assigned the default shader when created, so add a new lambert material at this step in your project's workflow.

6. **Choose Window → Rendering Editors → Hypershade.**

7. **Click on lambert swatch [Lambert] in the Surface materials panel.**

8. **Open the Attribute Editor.**

9. **Rename the material: lambert_plane.**

10. **Set Diffuse to 0.9.**

11. **Press the output connections button to bring up the shading group node and rename it: lambertSG_plane.**

You can leave the Color attribute alone; in the next step you will assign a texture to it.

12. **Adjust the Hypershade so that the plane is visible beside it.**

13. **MMB+drag the lambert_plane material swatch from the Hypershade over top of the plane in the scene view and release the mouse button.**

Create the grid texture

When creating a shading network, you have the option of building it in the Hypershade, before assigning it to an object, or constructing it on the fly—adding to it after it's been assigned. Here you'll do the latter, connecting a procedural texture to the Color channel of lambert_plane, which is already connected to pPlane1. The texture is a grid pattern and one of the ready-made Maya procedural textures.

The quickest way to add a texture to a channel is by pressing the Create Render Node button next to the channel in the Attribute Editor. This launches the Create Render Node editor. If you then select a node, it gets automatically connected to the channel in question. We used a lengthier method to connect grid1 to lambert_plane in order to demonstrate how to make attribute connections in the Hypershade.

1. **In the Hypershade, choose Create → 2D Textures → Grid**

or **Click on the grid texture swatch in the 2D Textures panel under the Create Bar. This creates a new node called grid1.**

2. **Select grid1 in the Hypershade and hit Ctrl+A to open the Attribute Editor.**

The objective here is to create a subtle pattern—enough to provide a frame of reference for the viewer without being distracting.

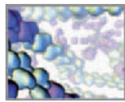
3. **Enter the following attribute values:**

Line Color	**H: 0** **S: 0** **V: 1.0 (white)**
Filler Color	**H: 0** **S: 0** **V: 0.8 (light gray)**
U Width	**0.01 (for thin lines)**
V Width	**0.01**

4. **Click on UV coordinates.**

5. **Click on the input connections button to bring up the 2D texture placement node.**

6. **Enter 40 in both of the Repeat UV fields.**



In the Hypershade, render node input and output connection arrows are either hollow ▷ to indicate that no connections exist, or solid ▶ when at least one attribute has a connection to another node.

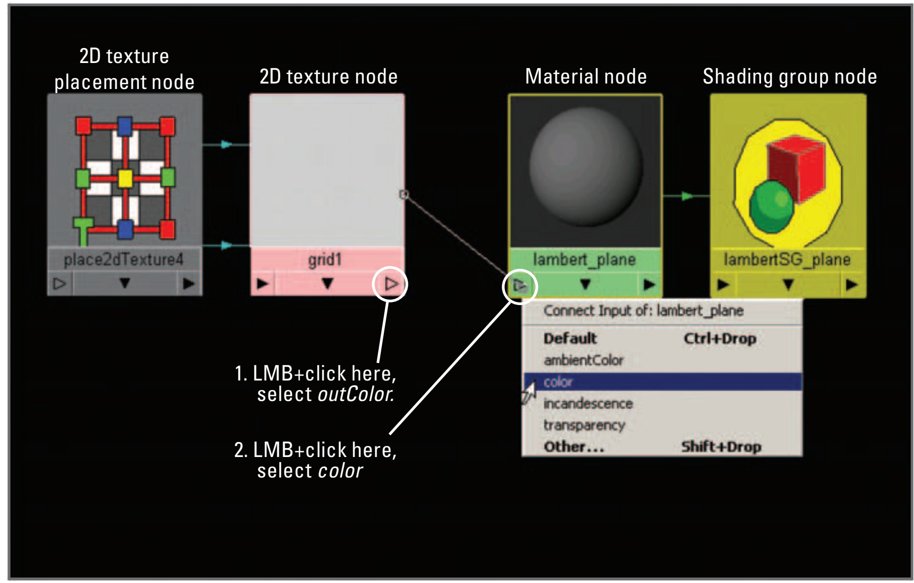


FIGURE 08.20

Attribute connections are made in the Hypershade by making a selection first in the output marking menu of one node and then in the input marking menu of a second node.

Increasing the Repeat attributes results in a finer grid (or more lines). Even with Hardware Texturing turned on in your scene view, you won't get a decent preview of the grid. The best way to see results is with a software rendering using the Render View. You'll get to that shortly.

Connect the texture to the material node

The first step here is to get the swatches for the two nodes, `lambert_plane` and `grid1`, lined up and visible in the Work Area of the Hypershade. After that, you will make the connection using marking menus that are accessed through output and input connections buttons on the node swatches.

1. **Shift+select lambert_plane and grid1 in the Work Area or in the Materials and Textures tabs, respectively.**
2. **Press the Graph Input and Output Connections button .**

   The Rearrange Graph button can be used to tidy up the graph view in the Work Area of the Hypershade.
3. **Rearrange the nodes to somewhat resemble the setup in** Figure 08.20**.**
4. **RMB+ or LMB+click the output connections button ▷ at the bottom right of grid1 and select outColor → outColor from the marking menu. Hold the mouse button down. A leader line now will follow your mouse pointer as you move it.**

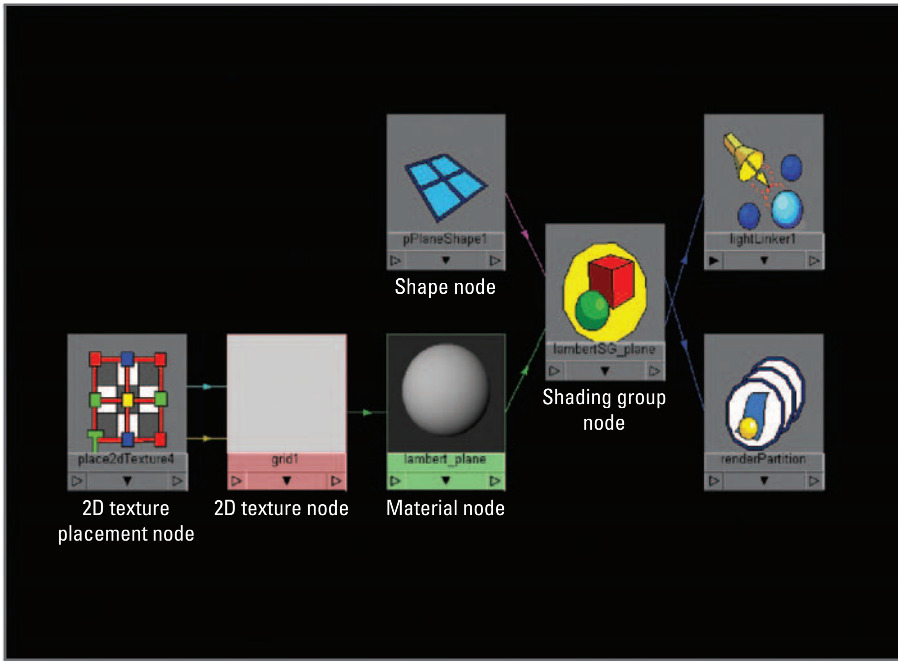


FIGURE 08.21

The finished shading network for the plane. The shading group (SG) node evaluates the other render nodes, along with lights and surface topography (via the pPlaneShape1 node), to determine the rendered appearance of the plane.

5. **RMB+** *or* **LMB+click anywhere on the lambert_plane swatch and select Color from the marking menu then release the mouse button (**Figure 08.19**).**

Now graph the complete shading network (Figure 08.21) to see the connections you have made since creating the material lambert_plane:

1. **In the Hypershade, select the lambert_plane material and press the Graph Input and Output Connections button .**

2. **Select lambertSG_plane and press .**

Make a preview render

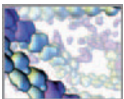
Before adding lights and a rendering camera, you can use the Render View to preview the effect of the shaders you created. Start by setting the render resolution in the Render Settings.

We will take a closer look at the Render Settings in Tutorial 11.01: Rendering.

1. **Choose Window → Rendering Editors → Render Settings (Render Globals in releases prior to Maya 7.0).**

or **Press the Render Settings button in the Status Line of the main window.**

2. **Press the Common tab and choose Presets → 640 × 480, under Image Size.**



3. **Press the Maya Software tab. If it isn't visible, select Maya Software from the Render Using menu.**
4. **Choose Quality → Production Quality, under Anti-aliasing Quality.**
5. **Press the Close button.**
6. **Move the Time Slider to a frame that you want to preview and adjust your scene view to a view you want to render.**
7. **Press the Render current frame button . This launches the Render View and starts a preview rendering.**

The result is a rendering of your scene using the default light and perspective camera. If you are not getting the results you want, you can compare your scene file with the completed tutorial file on the CD-ROM:

08_Shading/tutorial_08_01_done.ma

If you're at all like us, by this point in a project you can't wait to see a final complete render at production quality. For this you'll need to place the lights and camera of your virtual film studio before we can shout, "Action!" Let's get those set up!

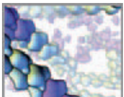
Summary

In Maya, shading is the creation of a network of render nodes. From a user's point of view, it is the application of colors, textures, and other surface attributes to elements in a scene. Render nodes include materials, textures, bump and displacement nodes, and texture placement nodes. The Hypershade is the primary tool in Maya for creating, applying, and editing shading networks. A shading engine (or group) node takes various render nodes for input, along with geometry and lights to calculate the final rendered appearance of a surface. The variety of materials and textures, along with options for photorealistic and NPR techniques, make Maya's shading capabilities extremely well suited to interpretive visualization challenges in biology. You have at your fingertips a wealth of techniques and familiar visual conventions—drawn from photography and illustration—for presenting dynamic graphical information.

References

1. Petterson EF, Goddard TD, Huang CC, Couch GS, Greenblatt DM, Meng EC, Ferrin TE: UCSF Chimera – A visualization system for exploratory research and analysis. *Journal of Computational Chemistry* 25: 1605–1612, 2004.
2. Chan NL, Rogers PH, Arnone A: Crystal structure of the S-nitroso form of liganded human hemoglobin. *Biochemistry* 37: 16459–16464, 1998.

09 Cameras

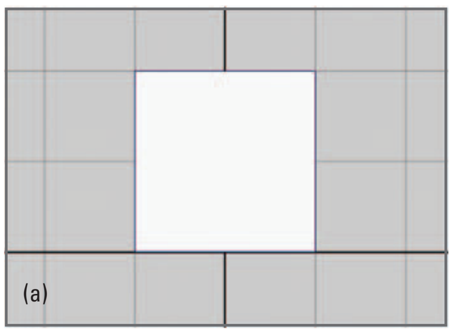


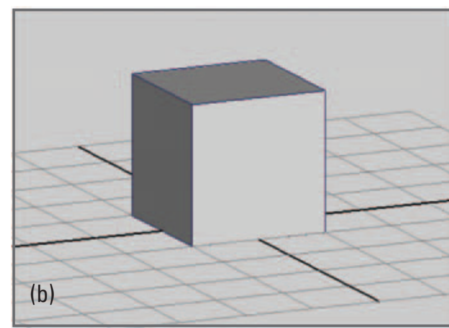
Introduction

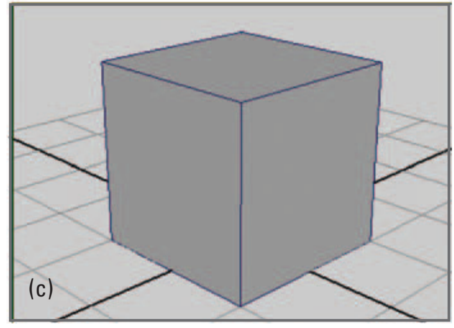
We have often heard the Hollywood refrain "Lights! Camera! Action!" Real life for the cinema pros, however, is a little different: that "Camera!", when it is all said and done, is to alert the camera operator to start the film running through the movie camera. Once it's rolling, the actors can get down to business. Before any of this can happen, though, the film's director must make a series of crucial decisions about where the camera is to be positioned for the shot and how it will move during the shot, along with lens effects such as **depth of field** and **field of view**. The camera supplies the audience its cinematic eye or the action; this eye must gaze adroitly if the audience is to experience the film's narrative and respond emotionally to the action. So really the refrain should be at least "Camera! Lights! Camera! Action!" Once the camera is planned for the shot, the cinematographer can decide how the scene is to be lit in order to achieve the visual style established for the shot.

As we have noted, Maya, like other high-end 3D animation packages, uses the simulacrum of virtual photography to let you approach your visual story using the language of cinema. All the choices a live action director confronts about camera placement, lenses, and lighting you must make too. There are now whole texts devoted to cinematography in live action and virtual (3D animation) worlds. Our objective in this chapter and the next one is to introduce you to the fascinating techniques and artistry of Maya cameras and Maya lighting. By the end of this chapter you will know what a Maya (virtual) camera is and how to set its basic properties. You will then apply this knowledge to replace Maya's default camera—which you used to preview the hemoglobin molecule in the last chapter—with a custom camera of your own design. We will guide you in its initial settings and placement, but you will quickly see the cinematic effects available to you as a Maya-based filmmaker. Unencumbered by gravity or physical risks, your Maya camera can move unconstrained by the limits within which live action cinematographers have to work. Since the properties of your cameras and lights are open to you through MEL for automated action, they become an essential part of your in silico visualization language as you formulate complex models: your cameras are your roving eyes on the results and predictions of your simulations, expressed in the universal language of cinema. Lighting is the other essential creative dimension of approaching your MEL simulation in cinematic terms. Having mastered the basics of the Maya camera in this chapter, you'll advance to custom light your hemoglobin in the next chapter.

Although you are free to experiment with camera placements and lighting arrangements using Maya as your prototyping environment, most of us find that—given the time, effort, and expense of designing and rendering 3D animation shots—many key decisions about cameras and lighting are best made in the preproduction stage of the visualization. From *Chapter 03* you'll recall that the storyboarding process of the preproduction stage allows you to plan your view of the 3D action shot by shot and establish in detail how you want the camera to move and the lighting design to apply to the scene. Unfortunately the art of cinema storyboarding is beyond the scope of this book, although two of us (CJL and NW) have a book on the subject in development for science filmmakers, based on our years of teaching (and using) the method with biomedical communications. In the meantime there are excellent texts available that will take you further with the art of storyboarding for film.







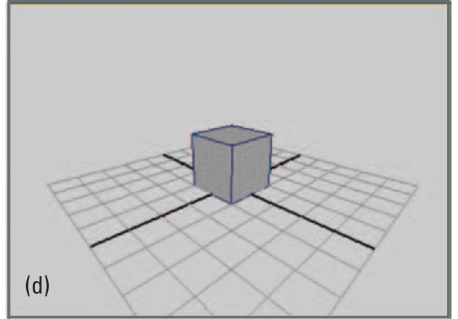


FIGURE 09.01

Different camera views of a polygon cube.

(a) A default orthographic camera view.

(b) When unlocked (using the Tumble Tool options), an orthographic view can be tumbled just like a perspective camera.

Camera attributes, **Angle of View** and **Focal Length**, control the viewable area and the degree of perspective distortion. The two are reciprocal; the longer the focal length, the narrower the view angle, and vice versa. For the images in (c) and (d), the camera remained in the same location; only the focal length (and angle of view) changed.

(c) A narrow angle of view and low distortion result from a focal length of 70 mm.

(d) A short focal length, 20 mm, gives a wide Angle of View, distorting the image.

Maya Cameras

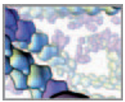
Like a real camera, a Maya camera defines the visible region of a scene—what will be captured in a rendering—and any image distortion due to the type of simulated lens used. You are by now no doubt familiar with maneuvering the default perspective camera, persp, to view a scene. Persp and the other default cameras—the orthographic front, top, and side—have all the same attributes as a camera you would create, and therefore can be used to render. It is advisable, however, to use the default cameras to view your scene as you set it up and create additional cameras specifically for rendering, that is, for "shooting" the virtual film of your Maya scene as established in your preproduction storyboarding plan, or its equivalent in your workflow. This will help you avoid accidentally moving a render camera, after setting it up, in order to view the scene.

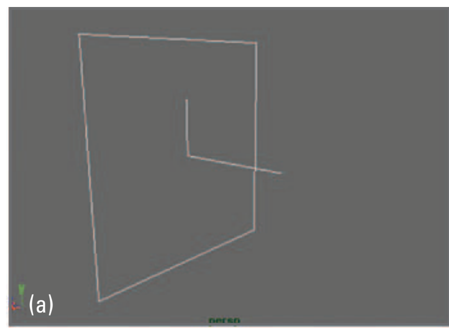
Essentially there is only one type of camera in Maya. When its Orthographic attribute is set to "1" or "yes" a camera becomes an orthographic camera. When a camera's Orthographic attribute is set to "0" or "no" we call it a perspective camera.

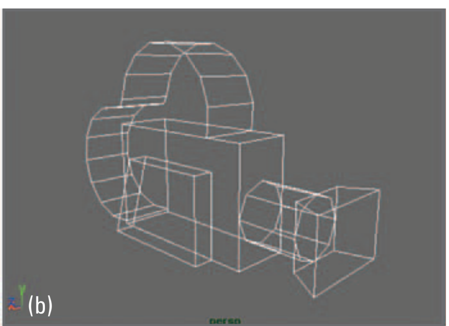
A perspective camera is like a real camera, with an **aperture**, **focal length**, and **angle of view**, but is more flexible. Far more flexible, as we hinted at above. Its simulated aperture can be of any size and aspect ratio (width : height), and works with a wide range of simulated focal length: 2.5–3500 mm. An orthographic camera has no distortion due to perspective; an object close the camera appears the same size as one that is far away. By default, an orthographic camera is locked to one of the major axis planes (XY, XZ, or YZ). You can unlock it using the Tumble Tool in order to tumble it to any point of view, as shown in Figure 09.01b.

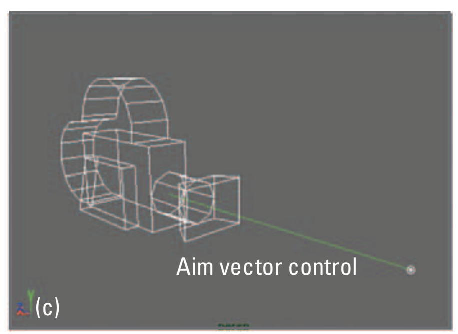
To avoid image artifacts related to depth perception, it is recommended that you use an orthographic rather than perspective camera for focal lengths greater than 400 mm.

You have three options when creating a perspective camera. These are **Camera**, **Camera and Aim**, and **Camera**, **Aim**, **and Up**, shown in Figure 09.02. The camera is the same in each









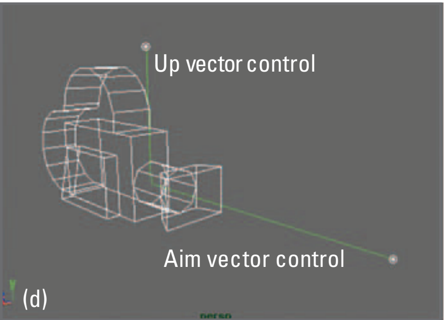


FIGURE 09.02

Maya's cameras all use the same transform and shape nodes but have different attribute or constraint settings. The different camera types are:

(a) Orthographic camera.

(b) Perspective camera.

(c) Perspective camera and Aim.

(d) Perspective camera, Aim, and Up.

case, but is constrained to locators, or null objects, in the latter two. The locators act as handles that drive the camera's `rotate` attributes. You aim a perspective camera interactively by tumbling, tracking, and dollying, or by transforming it using manipulator handles. With *Camera and Aim*, the camera will always point at the `camera_aim` locator, no matter where you translate it. As well, it will always be vertically upright, unable to tilt left or right (or rotate about its Z-axis). The additional locator with *Camera, Aim, and Up* is used to tilt the camera. At any time, you can change a camera from one type to another using the Attribute Editor. To create a camera:

Choose Create → Cameras → Camera
or Camera and Aim
or Camera, Aim, and Up

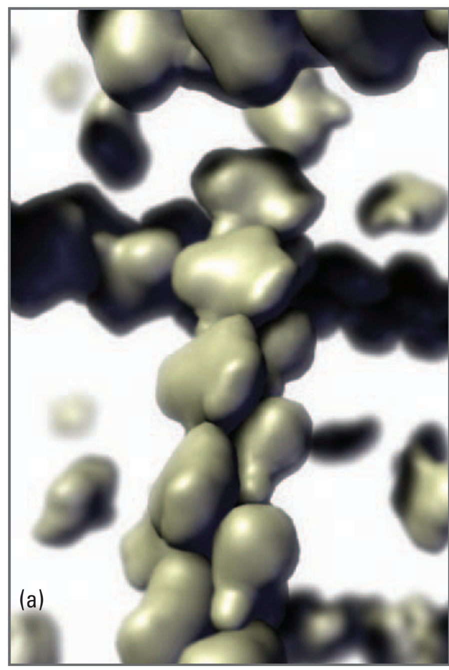
The create options for a camera can be left at their default settings and edited later in the Attribute Editor or Channel Box.

Camera Attribute Editor

Cameras clearly are essential to interacting with and rendering Maya scenes. They are the eyes you give your audience on what is happening in our model! Because of their importance we will discuss their attributes in depth.

Camera Attributes

The first camera attribute listed in the Camera Attribute Editor, Controls, allows you to switch between the different camera types listed above (regular, Aim, and



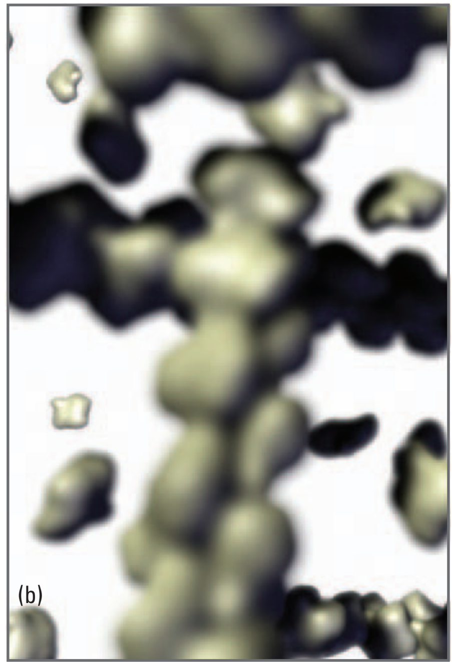


FIGURE 09.03

The depth-of-field effect varies the focus of a rendering based on the distance from the camera. This rendering is from a dynamic simulation of assembling actin protein filaments. The focal distance is set close to the camera in (a) and at the outer range of the simulation in (b). Depth of field can be useful device for directing the attention of your audience.

Aim and Up). `Angle of View` and `Focal Length` are inversely related. They control the relative width and lens distortion of the view. The greater the `Angle of View` value, the shorter the focal length and the greater the perspective distortion (Figure 09.01c and d) corresponding to a wide angle lens in conventional photography. Increasing `Camera Scale` has the effect of increasing the focal length—making images appear larger and decreasing the angle of view—and vice versa. **Clip** (or **clipping**) **planes** determine the visible range, perpendicular to the camera. If near or distant objects aren't visible in your scene, try decreasing the near clipping plane or increasing the far clipping plane.

The Camera Attribute Editor is simply the name given to the regular Maya Attribute Editor when a camera is selected.

Film Back

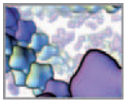
The Film Back attributes are used when you bring live action footage into Maya. Unless you are matching animation to live action, you can leave these settings at their default values.

mental ray

The mental ray attributes are for assigning shaders to your camera for special effects when rendering with the mental ray for Maya renderer.

Depth of Field

When activated, the `Depth of Field` attribute varies the focus of a rendering with distance from the camera (Figure 09.03). Because it mimics a familiar photographic effect—the blurring of objects outside of the focal range of the camera—depth of field can add the illusion of realism to renderings. However, this effect comes with a



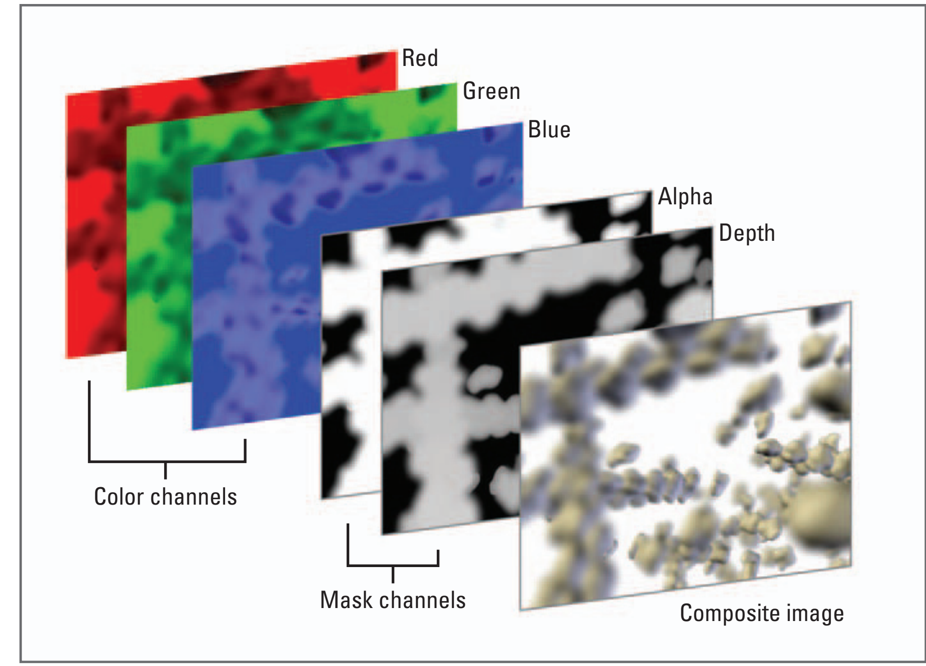


FIGURE 09.04

The rendering output channels are added together to create the final composite image. Channels are turned on or off in the camera Attribute Editor or in the Render Settings.

Depth-of-field effects are often created in the compositing stage of production using Maya's depth data channel, rather than in the actual Maya rendering. Depth of field is much quicker to render and easier to edit when applied in postproduction.

significant rendering time penalty. If you have access to a compositing program such as Adobe After Effects for animation or Adobe Photoshop for still images, you're better off rendering with `Depth of Field` turned off and then building the effect in the compositing stage using a Maya-rendered **depth channel** which contains the same depth data used to calculate depth of field. A depth channel is a bitmap image that represents distance from the camera using a grayscale value: the closest objects are white and furthest ones are black. Creating the depth-of-field effect in the compositing stage has the added benefit of allowing you to quickly fine-tune the degree of blur. In contrast, depth of field cannot be adjusted when it's been rendered into the primary footage. Regardless of the method used, depth of field is a powerful tool for isolating important visual information because the eye is naturally drawn to the point of sharpest focus. In crowded simulations of molecules or cells, it is one of our tools for directing the eye of the viewer.

Output Settings

Prior to Maya release 7.0, Render Settings was called Render Globals.

The Output Settings determine which standard channels will be rendered. When `Renderable` is checked, the camera will appear on the list of renderable cameras in the **Render Settings**. `Image` refers to the `RGB` color channels, `Mask` to the alpha channel, and `Depth` to the depth channel. These channels are illustrated in Figure 09.04. Alpha channels are used in compositing to mask out unwanted parts of an image so that other images can show through from behind. Black areas are masked out while white portions remain visible. A depth channel is used in the compositing stage of animation production to apply depth-of-field effects, and to achieve real-world distance effects, such as color saturation that decreases with distance from the viewer.

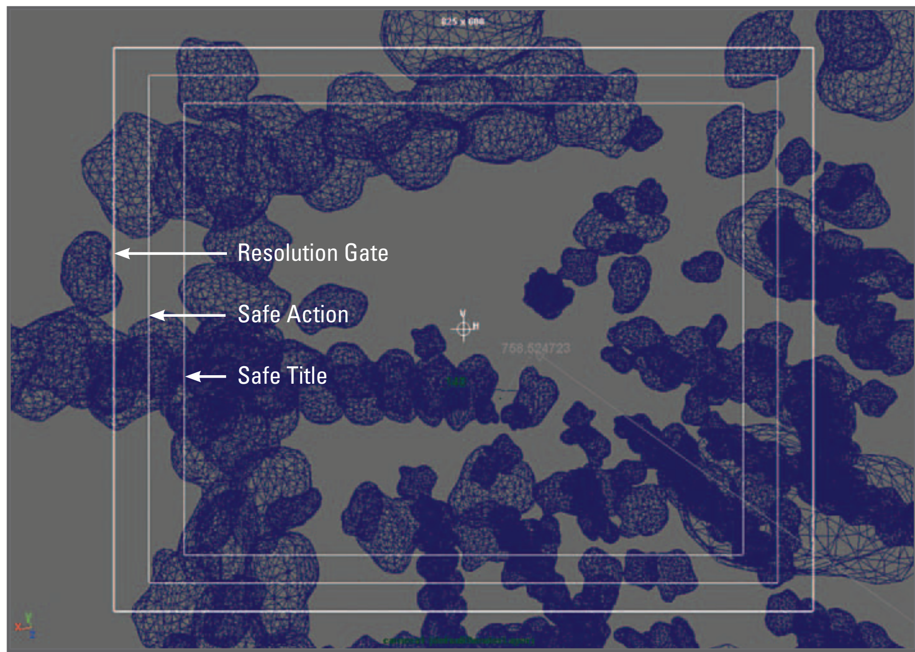


FIGURE 09.05

Camera Display Options. The Film Gate and Field Chart are not visible in this figure.

Environment

The Environment attributes determine the background, or what will appear in the spaces between the objects in your scene, once they're rendered in a flat image. `Background Color` sets a uniform flat color for the background. `Image Plane` lets you load bitmap image files or a procedural texture for the background. An image plane can be a template picture to use when building a model, or live footage to match animation to, as is often done in special effects work. In the image plane attributes, you can choose between an image file and a texture.

Display Options

The Display Options are labeled in Figure 09.05, with the exception of Field Chart which is a grid covering the renderable portion of the view. `Film Gate` shows the camera's aperture and applies to matching live footage. `Resolution` is the portion of the view that will be rendered. `Safe Action` and `Safe Title` apply if you plan to show your rendering on a television screen. Safe Action is 90% of the resolution region and is what will be fully visible on **NTSC** TV screen. On-screen type and logos should be kept within Safe Title, which is 80% of the resolution region. Overscan determines the viewable area outside of the renderable (resolution) region. It allows you to see what's just beyond the edges of your camera view.

Movement Options

Under Movement Options, Undoable Movements adds camera movements to the regular undo and redo commands that you access through the hotkeys Z and Shift+Z,

respectively. Center of Interest and Tumble Pivot set the pivot about which the camera tumbles. A camera can be set to tumble about itself, a defined Center of Interest, or a user-defined Tumble Pivot. You set which one is used in the Tumble Pivot Tool Options window. It uses the values specified for Center of Interest and Tumble Pivot defined in the Attribute Editor. To set the Tumble Tool options:

1. **Choose View → Camera Tools → Tumble Tools ❐.**
2. **Check one of Center of Interest *or* Tumble Pivot.**

If you check Tumble on Object, the camera will tumble about whichever object the tumble icon is over when you begin to tumble. The remaining tumble settings control the maneuverability of an orthographic view. With Locked unchecked, you can tumble an orthographic camera the way you would a perspective one.

Orthographic Views

Orthographic turns a perspective camera into an orthographic gaze. Orthographic Width is related directly to the distance of the camera from its origin plane (XY, XZ, or YZ). When you dolly an orthographic camera, you don't actually move it in the Z-direction, but rather increase or decrease its Width attribute.

The remaining camera attributes are common to all shape nodes and won't be covered here.

Camera Attributes

Maya Help → Using Maya → Rendering and Render Setup → Rendering → Rendering menus → Panel menus → View → View > Camera Attribute Editor

Tutorial 09.01: A camera on hemoglobin

One of the joys of working in 3D computer animation is composing shots using a virtual camera. You can mimic the techniques of experienced cinematographers seen in movies and on television. Well-executed cinematography, like good film editing, often goes unnoticed by the audience because it appears natural. Through CGI camera work, audiences are even becoming accustomed to quite unnatural, but striking viewing experiences. Examples include high-speed roller-coaster-like rides through cities and landscapes, and tours of tight, often microscopic spaces—all of which would be impossible with present-day photographic equipment. Three-dimensional in silico biology provides a host of opportunities for innovative camera work. Motion path animation, in which a camera animates along a track, can be used to move a camera on a complex trajectory through a scene dense with reacting molecules, for example. Likewise, a camera can be programmed to track specific events within a simulation. Such an "intelligent" camera could potentially remove much of the labor required in conventional camera setup techniques.

This exercise builds on our hemoglobin scene from the last chapter. Let's create a camera that sweeps around the animated hemoglobin subunits on a motion path. It's worth recalling that we chose to set up the camera before lighting the scene because what the camera sees will inform your lighting decisions. To start, you can either use

the scene you modified in *Tutorial 08.01* from the previous chapter or copy the following file from the CD-ROM to your scenes directory.

09_Cameras/tutorial_09_01.ma

Whether you got the file from the CD-ROM or from your progress from the previous tutorial, the file preferences already will be set—Maya stores settings, such as Working Units and Playback Speed in the scene file. You're working with a short animation of the four subunits of **hemoglobin**—the oxygen-transport molecule found in red blood cells—coming together to form the complete molecule. Each subunit has a shader applied to it and the intended renderer is Maya Software.

Create a camera

The first step is to create the render camera. It doesn't matter from the standpoint of options which camera type you choose at first because the creation options are the same for each, and you can change the type at any time using the Camera Attribute Editor so your options will stay flexible. Nor do you need to bother with the creation options since they can all be set in the Attribute Editor.

1. **Choose Create → Cameras → Camera ❐.**
2. **Press the Create button.**

or

3. **Press the Camera button on the Rendering shelf.**

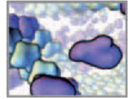
Leave the default settings as is, including the name, `camera1`. In projects like these, involving an overview of a potentially exotic object, we often set the `Focal Length` from 35 to 50 mm in order to get perspective familiar to that of the human eye. In this case, however, you want to enhance the illusion of perspective slightly for a more interesting view, so you'll leave it at 35 mm. Try to find time to come back and experiment with different settings. For instance, what would be the effect of locating a camera with an extreme wide angle lens close to the subunits of this molecule?

Next, you'll scale the camera in order to make it visible relative to the much larger polygonal objects in the scene. Scaling a camera using its transform node has no effect on the view it provides or the files it renders—it just makes it easier to see in the workspace.

4. **Select camera1 in the Outliner.**
5. **In the Channel Box, enter 20 in the Scale X, Y, and Z attributes.**

Set up a two-panel view

We find it helpful to use two scene views when setting up a camera (Figure 09.06). On the left is the default persp view, with which you'll navigate the scene. On the right is the render camera (`camera1`) view, in order to see what it sees. Different panel setups suit different workflows. For example, when working with keyframe animation rather than MEL simulation events, it's useful to dedicate at least one panel to the Graph Editor in order to have easy access to animation curves. Some users prefer to keep the



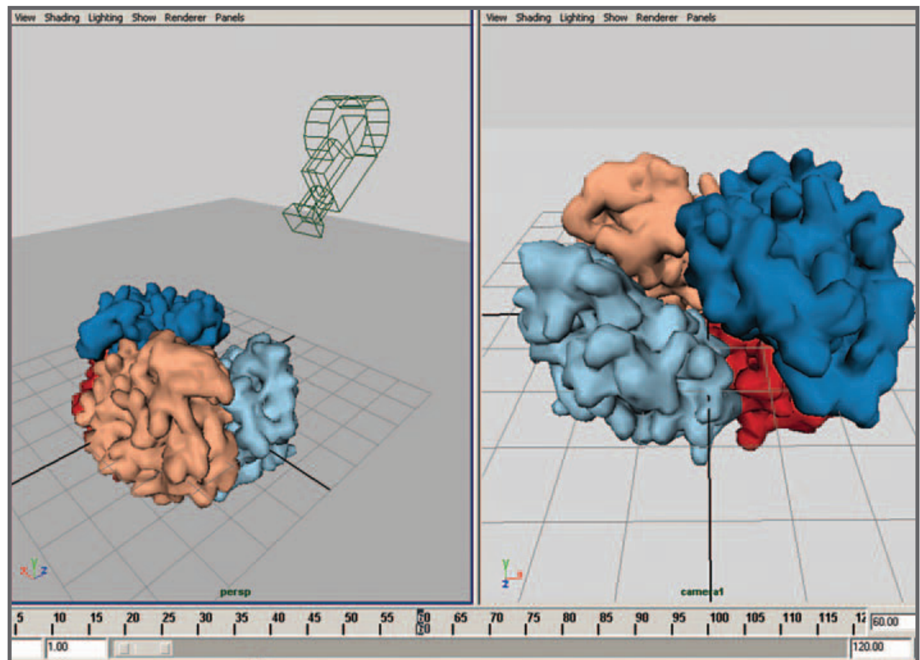


FIGURE 09.06

A two-panel layout is a good way to view your scene while setting up a render camera. In this image, the current time indicator is at frame 60—the point at which the hemoglobin subunits come together.

Outliner embedded as a panel for quick access. For the present exercise, you'll find it most efficient to allocate two panels to camera views and launch other elements, like the Outliner, as separate, floating windows.

1. **Choose Panels → Layouts → Two Panes Side by Side.**
2. **In the left-hand panel, choose Panels → Perspective → persp.**
3. **In the right-hand panel, choose Panels → Perspective → camera1.**

If you haven't yet moved `camera1`, it will be positioned at the world origin, facing in the negative Z-direction. Depending on which frame the time indicator is at, the camera may be inside one of the models, in which case you will see nothing recognizable. A quick way to get a view of your whole scene through the active panel is to choose Frame All, as follows:

1. **Click anywhere in the right (camera1) view panel to make it active.**
2. **Choose View → Frame All**

or

Hit the hotkey "A".

The cousin to Frame All is Frame Selection, which adjusts the active view to show one or more selected objects. It is accessed through the View menu or with the hotkey "F". If no objects are selected, "F" also frames all.

Take a minute to tumble, dolly, and pan `camera1` to get a good view of the four hemoglobin subunits. Then do the same with `persp` so that you can see the geometry and `camera1` together.

Adjust the camera attributes and Render Settings

Here you will add an aim locator, set the camera aperture (renderable width and height), and display the **resolution gate**. An **aim** is a target that the camera is constrained to look at. You will use this to keep the camera pointed at the world origin. The aperture will, by default, match the aspect ratio you set with Image Size in the Render Settings. The resolution gate shows you the renderable area of the camera.

Add the aim locator

1. **Select camera1 and open the Attribute Editor.**
2. **Under Camera Attributes, choose Control → Camera and Aim.**

This parents camera1 under a new transform node called camera1_group, along with camera1_aim, the locator node. Selecting camera1 now, you will see that the Rotate X, Y, and Z attribute fields are colored blue to indicate that they are constrained to the aim locator.

Unless the locator is constrained to a position, it will move as you pan the camera. You want it locked to the origin in this example so that the camera will always point at the center of your scene. This will keep the hemoglobin molecule centered up as you fly around it.

3. **Select camera1_aim in the Outliner.**
4. **In the Channel Box, enter 0 for Translate X, Y, and Z.**
5. **Select Translate X, Y, and Z by name in the Channel Box.**
6. **RMB+click to bring up the Channel Box Marking menu and select Lock Selected.**

The aim locator is now locked to the origin. If you pan camera1, it will snap back to its previous position, centered on the locator at the world origin.

Set the Image Size

1. **Choose Window → Rendering Editors → Render Settings (Render Globals in releases prior to Maya 7.0).**

or **Press the Render Settings button in the Status Line of the main window.**

2. **Press the Common tab and choose Presets → 320 x 240, under Image Size.**
3. **Press the Close button.**

You're using half the regular default resolution of 640 × 480 pixels for faster test rendering.

Unlike computer-image pixels, which are always square, high-end video formats often use the unintuitive notion of non-square pixels. As you can see in Table 09.01, standard definition NTSC video has pixels that are taller than they are wide, at a ratio of 0.9:1. This allows the video frame to fit into the 4:3 aspect ratio of NTSC video, even though its pixel dimensions (720 × 486) don't resolve to 4:3.

Display the resolution gate

The image size determines the resolution aspect ratio—1.333:1—which sets the shape of the resolution gate. Commonly used image formats are listed in Table 09.01.

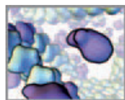
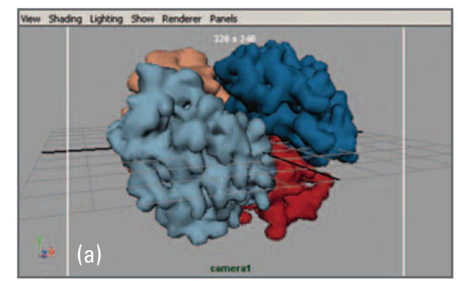


TABLE 09.01

Common rendering image sizes and the corresponding aspect ratios. The Device Aspect Ratio is the image width to height ratio in terms of the *number* of pixels. The Pixel Aspect Ratio (or PAR) is the width to height ratio of individual rectangular pixels. A PAR of 1 corresponds to square pixels.

Format	Resolution (pixels)	Device Aspect Ratio (width: height)	Pixel Aspect Ratio (width: height)
CCIR 601|Quantel NTSC (North American video)	720 × 486	4:3(= 1.333)	0.9:1
CCIR PAL|Quantel PAL (European video)	720 × 576	4:3(= 1.333)	1.066:1
HD 1080 (High Definition)	1920 × 1080	16:9 (= 1.777)	1:1



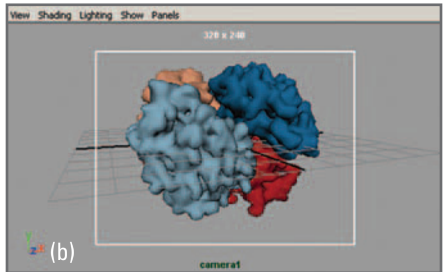


FIGURE 09.07

Overscan scales the resolution gate (white line) relative to the scene view, but doesn't alter the renderable image within the resolution gate.

(a) Overscan = 1.0; the render-able image fills the view in one dimension with the resolution gate.

(b) Overscan = 1.3; a value greater than 1 scales the gate down so that more of the scene is visible around the edges.

1. In the camera1 view panel, choose View → Camera Settings → Resolution Gate.

or **(a) Select camera1 and open the Attribute Editor.**

(b) Under Display Options, select Display Resolution.

Because the camera1 view panel is tall and narrow—what we call a "vertical layout"—it cuts off the sides of the resolution gate. To fix this, you will adjust the size of the resolution gate relative to the film gate.

2. Choose View → Camera Settings → Overscan.

or **(a) Select camera1 and open the Attribute Editor.**

(b) Under Film Back, select Fit Resolution Gate → Overscan.

3. In the Attribute Editor, enter 1.1 for the Overscan attribute.

In most cases, you can leave the Film Back attributes at their default values. Setting Fit Resolution Gate to Overscan will ensure that the renderable area is visible in the view panel.

Overscan controls how much of the scene you can see outside of the resolution gate. Values greater than 1 result in the view panel showing more of the scene than just what will render within the resolution gate. Figure 09.07 shows the results of different Overscan settings.

Animate camera1 on a path

A motion path is a spline curve used to guide the animation of an item's transform node. This can be anything in Maya with a transform node. Using a motion path for a Maya camera is similar to using a track to guide a real-world movie camera—it's a good way to get smooth, predictable movement of your viewpoint through a scene. Unlike the real world, however, in Maya a camera setup isn't restricted by the laws of physics and construction budgets; you can create elaborate roller-coaster-like trajectories, gravity-ignoring 3D orbits, and film a scene from any conceivable vantage point without concern for the complexities and cost of an equivalent real-world scenario.

Note: By default, the camera's overscan attribute is visible in the Attribute Editor but not the Channel Box. You can add overscan to the Channel Box attributes using the Channel Control Editor, available through the Channels menu at the top of the Channel Box.

Create the motion path

Your first step is to create the path, for which you'll use a circle to make the camera follow a smooth, horizontal arc around the moving geometry. So both the hemoglobin subunits and your camera will be moving.

1. **Choose Create → NURBS Primitives. If Interactive Creation is checked, select it to turn it off.**
2. **Choose Create → NURBS Primitives → Circle ❐.**
3. **Set Normal Axis to Y and Radius to 150.**
4. **Press Create.**

This makes a spline called `circle1` that is centered at the world origin.

5. **Hit the hotkey to bring up the Animation menu set.**
6. **Select camera1 and then circle1.**
7. **Choose → Animate → Motion Paths → Attach to Motion Paths ❐.**

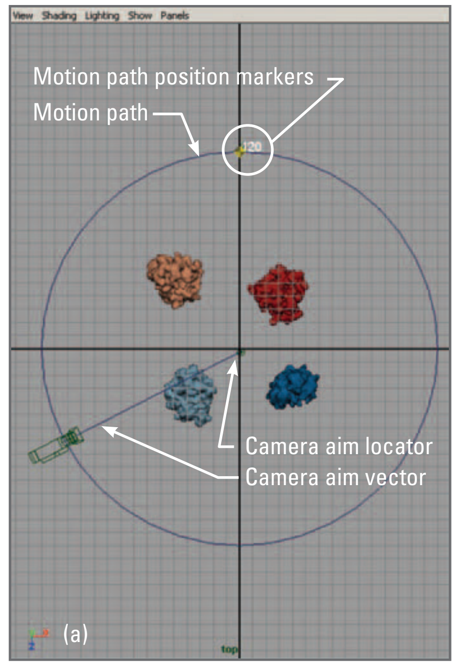
The `Time Range` attribute determines the start and end times for the motion path animation. Leave this set to `Time Slider`. This makes the start time equal to frame 1 and the end time equal to frame 120 (provided you haven't altered the start and end frames in the Timeline). The remaining attributes are fine at their default values except for `Follow`, which makes a connection involving the camera's rotation attributes. It must be unchecked, otherwise it will generate an error because the rotate attributes are already constrained to the aim locator.

The **follow** attribute for a motion path determines how the attached object changes orientation as it follows the path.

8. **Uncheck Follow then press the Attach button.**

Edit the animation

`camera1` will automatically be set to begin its motion at the start of the curve and finish moving at its end. The beginning and end of a motion path are marked by objects called **position markers**. Each marks a normalized (between 0 and 1) position along the curve. This position is displayed in the Channel Box when the marker is selected, under the attribute name Local Position X. For a circle, both markers appear to lie at the same point because the start and end of a circle are coincident. However, a glance at the `Local Position X` attribute for each will show that they lay at positions 0 and 1,



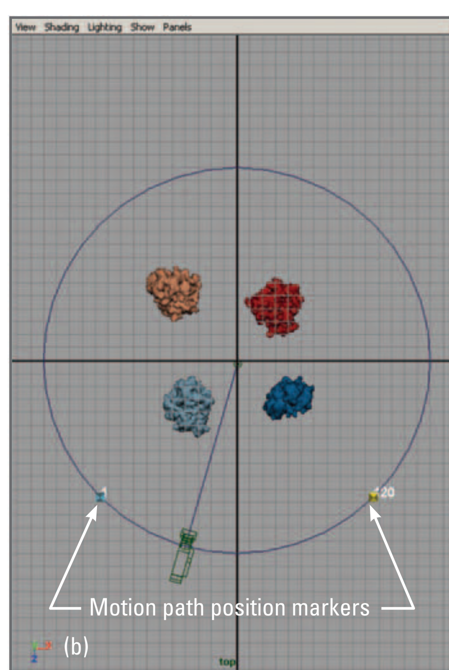


FIGURE 09.08

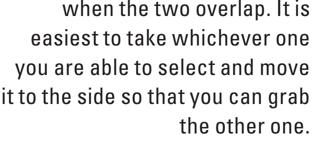
Motion path position markers:

(a) Positioned by default at the start and end of the motion path curve (top of the circle).

(b) Repositioned using the Move Tool.

These images were captured using the Top camera at frame 40, or one-third of the way through the hemoglobin animation.

respectively. You can drag a marker using the Move Tool to change its position on the curve and use the Attribute Editor to change its frame number. In this exercise you don't want a fast 360° rotation around the scene, but rather a slow 45° arc, with the camera facing primarily in the negative Z-direction. These choices are somewhat arbitrary and purely for the purpose of demonstration. Please take them further!


It is difficult to select one position marker over another when the two overlap. It is easiest to take whichever one you are able to select and move it to the side so that you can grab the other one.

9. **Hit the hotkey "W" to activate the Move Tool.**

10. **One at a time, select a position marker by clicking on its number beside the circle. If doesn't matter which one you grab first.**

11. **Drag positionMarker1 (with frame label "1") to roughly 0.375.**

    Drag positionMarker2 (with frame label "120") to 0.625.

    You can watch the Local Position X attribute update in the Channel Box until you're in the right spot.

Figure 09.08 shows the position markers before and after being moved. Play the animation to make sure the camera is working as it should. With the persp window active during playback you will see camera1 tracking along the motion path between frames 1 and 120.

Path animation

Maya Help → Using Maya → Animation, Character Setup, and Deformers → Animation → Path Animation

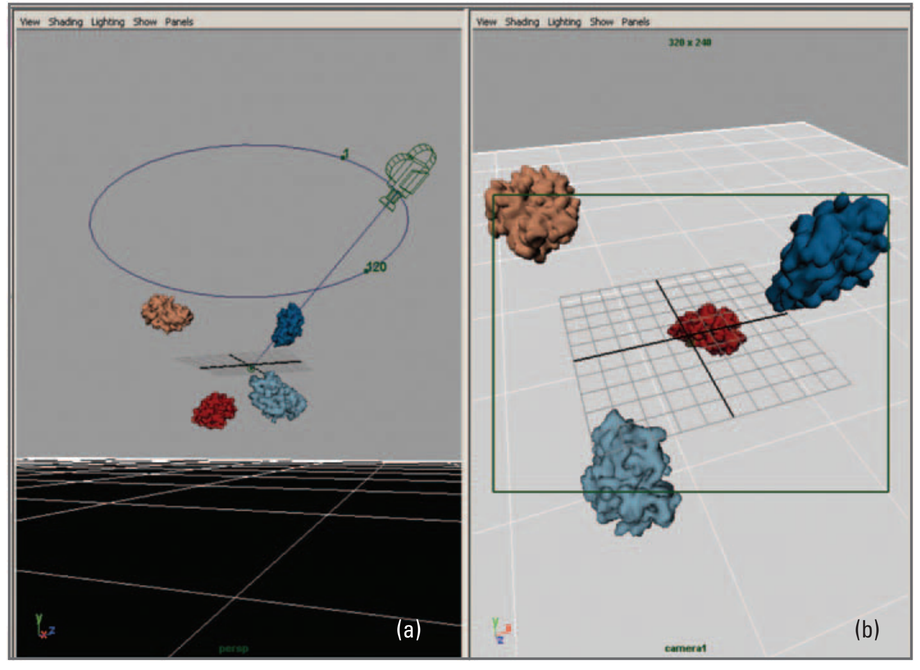


FIGURE 09.09

A view of the scene after translating the motion path:

(a) from the default perspective camera, persp.

(b) From camera1.

Adjust the camera

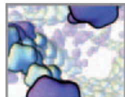
Currently, your camera is horizontally aligned with the origin. At this orientation, it doesn't see much of the plane you just created and neither will it see much of the cast shadows. Moving camera1 up will force it to look down toward its aim and therefore see more of the plane and shadows. This is done simply by translating circle1 in the Y-direction.

12. **Select circle1 in the scene view or the Outliner.**
13. **In the Channel Box, enter 120 for the Translate Y attribute.**

Your scene should now resemble Figure 09.09a when viewed from below the origin with the persp camera. Figure 09.09b shows the view through camera1. Make the camera1 view active and press Play to preview the animation from this new vantage point. While you haven't added a light or rendered a single frame, you now have a pretty good idea of how the action will appear when you render it. If you are unable to get relatively smooth playback in the scene view with the Playback Speed set to Real-time (30 fps), a playblast is a good idea at this point. Playblast preview renderings were discussed in *Chapter 06*.

Creating a playblast

Maya Help → Using Maya → Animation, Character Setup, and Deformers → Animation → Animation Basics → Playback animation → Playblast animation



The finished Maya scene file for this tutorial can be found on the CD-ROM:

09_Cameras/tutorial_09_01_done.ma

You're now ready to light your scene to bring out the information you want the audience following.

Summary

A Maya camera has many properties in common with a real-world movie camera but is more flexible. It can be of two types, orthographic or perspective. The latter is typically used for rendering since it emulates the perspective distortion we are used to seeing as a result of eyes and cameras with limited focal length (an orthographic camera is essentially a perspective camera with infinite focal length). In addition to its many shape attributes which control the image and special effects, like depth of field, a Maya camera can be animated to move and tumble using its transform node. This enables you to follow important action in a scene, much like a real camera on a track, dolly, or moving tripod. This in turn has significant implications for in silico simulation work: the inability to see inside and move around tiny, dynamic structures and systems continues to be a limiting factor with in vivo and in vitro methods of investigation. In contrast, 3D in silico methods potentially allow complete spatio-temporal transparency, and with it, new opportunities for discovery.

10 Lighting

Lighting

With your scene shaded and camera set, the natural next step is to add lights. The range of "looks" and visual styles you can achieve with lighting is nothing short of remarkable. You might set one project to be filled with even, mid-day illumination typical of a wildlife documentary. In another, you might fill your Maya scene with deep shadows and knife-edged pools of illumination to enhance a dramatic message or tightly focus your viewer's attention. As with camera work, cinema lighting is an art form in its own right and there are inspirational texts that will take you beyond the starting point we can accommodate here as we set the foundation for your explorations in MEL.

Lights come in six varieties in Maya. These are shown in Table 10.01, along with rendered examples. Before you add lights to your scene, Maya creates a directional light at render time, which illuminates objects evenly from the upper left and then deletes the light after rendering. This temporary light is sufficient for quick shader tests, but cannot be edited and should therefore be replaced by a custom lighting setup as soon as you're ready to start rendering. Custom lights are invoked through the Create menu or via the Hypershade.

Your scene will also contain a **default light** to make objects visible to your in Shaded display modes in the scene view. If you have created other lights, you can choose to use all of them or just select ones for scene view shading. Do this by selecting the appropriate light(s) in the scene view Lighting menu (Figure 10.01).

Master cinematographers spend their lifetimes perfecting their lighting artistry. Nonetheless, a good, basic lighting setup is well within your reach. In this chapter's tutorial, you will create a classic setup that uses three Point lights to illuminate objects in the scene in a way that accentuates their three-dimensional form. Maya Help provides general information on lighting concepts and basic instruction for working with lights in a Maya scene.

Lighting in Maya

Maya Help → Using Maya → Rendering and Render Setup → Lighting → Basics of lighting → Light and shadow in the real world

Shadows

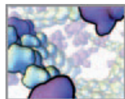
In the natural world, cast shadows tell us a lot about the spatial relationships of objects. When used correctly, they can do the same in 3D computer renderings. In Maya, a shadow is the lack of illumination on a surface or volume, caused by an object that blocks a shadow-casting light source. Surfaces facing away from the light are considered *not illuminated* versus being in shadow. You can specify whether or not an object casts and/or receives shadows under Render Stats in the Attribute Editor for its shape node.

When you create a light, by default it is set not to cast shadows. This saves processing time when the scene is rendered. Of the two shadow options available, **depth map** and **raytraced shadows**, the former is less accurate, but much quicker to render than the latter and usually sufficient. The two things raytraced shadows can do that depth

Light	Description	Sample rendering
Ambient	Simulates a combination of direct light (from its location) and diffuse light from all direction in the scene. This light does not cast depth map shadows.	
Area	Simulates a rectangular light source such as a window. The size (scale) of this light affects its intensity. Area lights can take longer to render than other types, but can generate higher-quality light and shadows.	
Directional	Simulates a very distant light source, such as the sun. The light rays are parallel and run one direction. This light is useful for lighting many objects in a scene evenly from a single source.	
Point	Simulates light emanating in all directions from a point in space. Point lights are quick to set up because their effect is independent of direction and scale.	
Spot	Simulates light emanating from a cone. The edges of the cone can be hard or soft, a feature controlled by the light's Penumbra attribute.	
Volume	Simulates light originating from a point and confined to a volume. The falloff of light from the point to the volume boundary is controlled by a color ramp.	

TABLE 10.01

Maya lights.



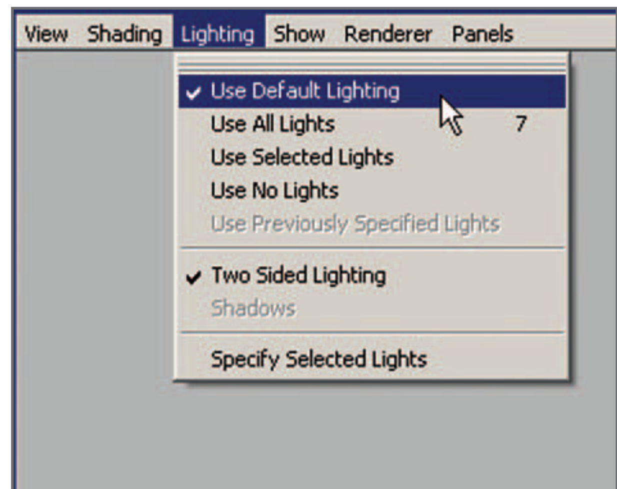
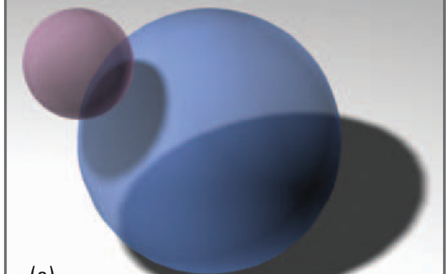


FIGURE 10.01

The Lighting menu in the Panel menus allows you to choose which light(s) will be used in Shaded mode in the scene view.



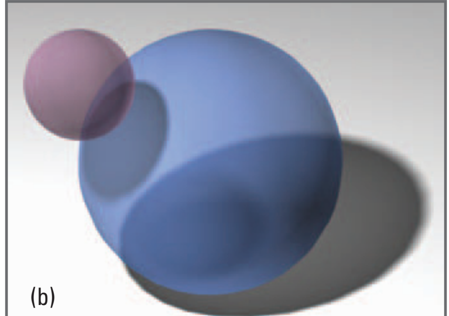


FIGURE 10.02

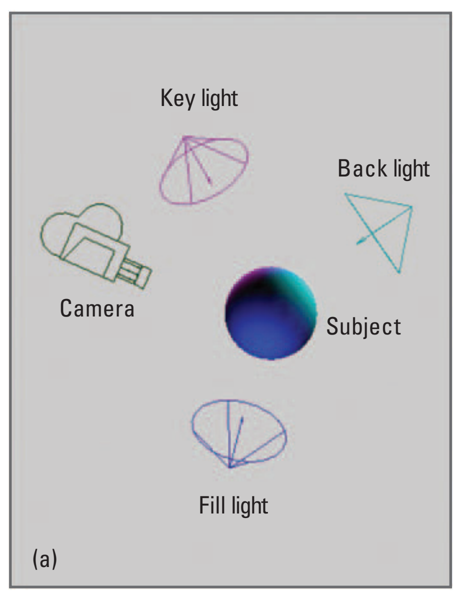
Depth map shadows (a) are fast to render but lack some of the realism possible with raytracing (b); namely, internal shadowing on transparent objects and edge blurring that progresses with distance from the source of the shadow.

map shadows cannot are cast shadows on the inside surfaces of transparent objects and produce soft shadows that are more physically realistic. Figure 10.02 shows the difference between depth map and raytraced shadows for the same objects, using the same light source.

Raytraced shadows and, to a lesser degree, depth map shadows increase render times, so discretion is warranted when deciding which lights will cast shadows and which objects will be affected. It is advisable to use depth map shadows whenever possible, cast from a single light, and to turn off Receive Shadows (in the Attribute Editor → Render Stats) on all objects for which shadows are unnecessary.

Shadows

Maya Help → Using Maya → Rendering and Render Setup → Lighting → Basics of lighting → Shadow → Shadow in Maya



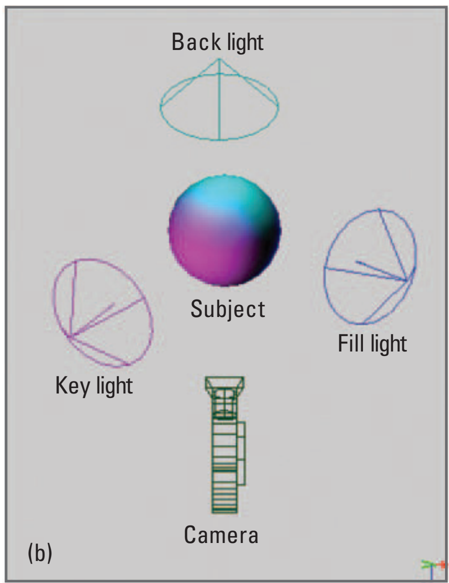
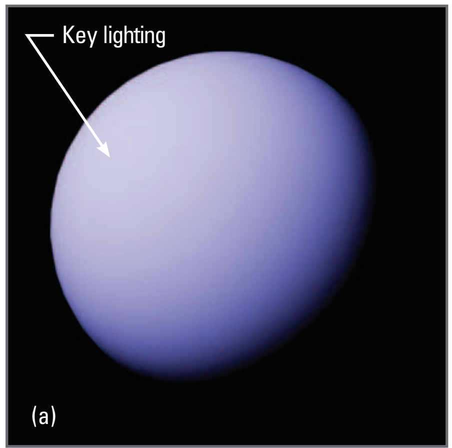


FIGURE 10.03

A standard 3-point lighting rig involves a key, a fill, and a back light. We used spot lights for this illustration to emphasize the directional nature of the illumination.

(a) Side view.

(b) Top view.



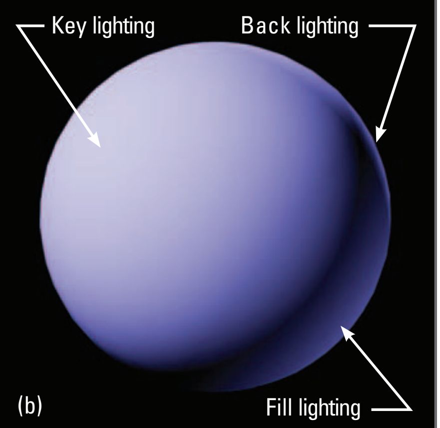


FIGURE 10.04

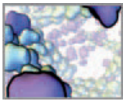
Multiple lights can be used to create naturally-occurring illumination effects such as lighting along the contour of an object and reflected (or fill) light. Such effects contribute to the illusion of 3D form and help to visually separate objects from one another and from the background. Notice how, without the back and fill lights, the sphere in (a) blends in with the dark background.

(a) Using a single key light.

(b) Using key, back, and fill lights.

Tutorial 10.01: Lighting the hemoglobin scene

Now that the rendering camera is ready, you can set up lights knowing where they need to be to provide proper illumination for the shot as the molecules move and your camera orbits through the 3D environment. In this tutorial, we want you to create a standard stationary **3-point lighting rig** (Figure 10.03). This involves a **key light**, supplying the main source of illumination, a **fill light** to balance the dark spots missed by the key, and a **back light**, to help separate the edges of objects from the background. A back light is useful at times, but is not always essential; its effect can be quite subtle. A fill light, on the other hand, contributes greatly to the illusion of 3D form. Figure 10.04



illustrates the contribution of back and fill lights to the basic illumination of a key light.

In some cases it may be desirable to animate your lights to compensate for a moving camera and/or geometry.

Lighting doesn't just define form; it also sets the mood and atmosphere of a picture or film. Lighting can be used to impart a sense of calm or urgency. It can reveal secrets and enhance mystery. A convention that you will use in this tutorial is to light from the upper left—a common practice in commercial photography and illustration.

To begin, start Maya and open the scene you created in the previous tutorial or copy the ready-made scene file from the CD-ROM.

10_Lighting/tutorial_10_01.ma

Create the lights

You will create three Point lights for this exercise and convert one to an Area light for the back lighting. Point lights are easy to use and their effects highly predictable, which makes them attractive choices for standard lighting setups. An Area light, which emits from a plane instead of a point, is more effective for back lighting because it illuminates a greater proportion of the geometry than a Point light, when shone from behind.

1. **(a) Choose Create → Lights → Point light.**

   (b) Hit the repeat last action hotkey G twice to create two more lights.
2. **Rename the lights key_light, fill_light, and back_light.**
3. **Select fill_light. In the Channel Box, enter 0.5 for Intensity.**
4. **Select back_light and open the Attribute Editor.**
5. **Choose Point light Attributes → Type → Area Light to change the light type.**
6. **Set the following attribute values:**

   Intensity: 0.5

   Scale X: 200

   Scale Y: 100

   Scale Z: 50

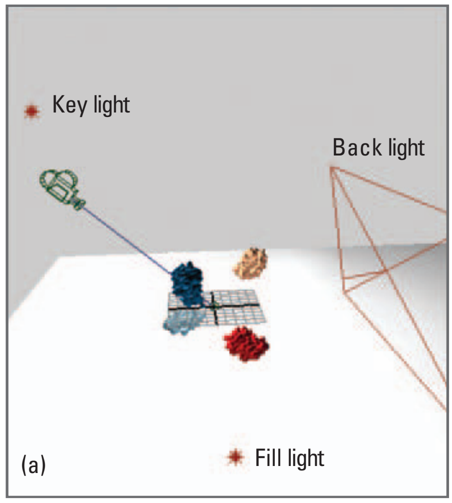
The length and width (X and Y) of an Area light determine its region of illumination. An Area light has a normal, much like a surface normal, that indicates the direction in which it shines. Increasing the Scale Z value simply makes the light normal bigger, and therefore easier to see (to you; it does not appear in the rendered scene!), but doesn't affect illumination.

7. **In the camera1 view panel:**

   (a) choose Shading → Smooth Shade All.

   (b) choose Lighting → Use All Lights.

The settings you made in step 7 will provide a rough preview of the lighting situation in the scene. With all lights in the same position, the scene will appear over-lit, or "blown out". This will be fixed when you move the lights to their proper positions.



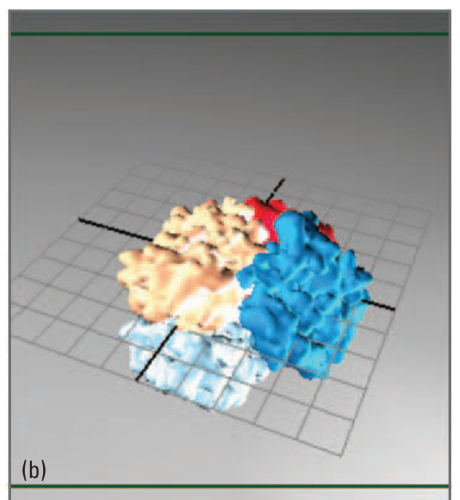


FIGURE 10.05

(a) The placement of lights prior to tweaking with IPR.

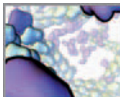
(b) Increasing the Intensity of the fill light makes its effect more obvious in the scene view, which makes the light easier to position.

Place the lights

Since the Point lights are multidirectional, you need only be concerned with their Translate values and not their Rotations. In contrast, the orientation of the Area light has everything to do with its illumination; it emits light in the direction of its normal. Furthermore, because in this project we have decided that the lights are stationary and the camera moves, the illumination will look different from different camera angles. In a rendering, this fact will enhance the illusion of 3D form and give the sense that the camera is moving relative to the objects. This also means that the lighting must be effective for the range of camera motion. We must therefore pick a representative frame at which to set up the lighting rig; we chose the half-way point in the animation, frame 60. Your goal is to get the lights working well at frame 60 and then make minor adjustments if needed at frames 1 and 120.

1. **Move the current time indicator to frame 60 (half-way through the animation).**
2. **Hit the hotkey to activate the Move Tool.**
3. **In the persp view:**
   (a) **Select key_light in the scene view or the Outliner and drag it to a position above and to the left of camera1.**
   (b) **Select fill_light and drag it to a position below and to the right of camera1.**
   (c) **Select back_light and drag it to a position behind the polygon models, relative to camera1.**
   (d) **Rotate back_light so that its normal points toward the polygon models.**

Figure 10.05a shows approximately where the three lights should be placed. Temporarily increase the Intensity for each light to 2 or 3 in order to observe its effect in the scene, as shown in Figure 10.05b. Fine-tuning of the lights will be done after you make the shadow settings.



Turn on shadows

Shadow casting is controlled by attributes in a light's shape node. It is, by default, turned off when you create a light. In this tutorial raytraced shadows are unnecessary since you have no need for shadows within transparent objects and you're not rendering translucent effects. Depth map shadows will be more than adequate. As well, you will designate the key light as the only shadow-caster, to keep shadow calculation time to a minimum.

1. **Select key_light and open the Attribute Editor.**
2. **Under Shadows → Depth Map Shadow Attributes, check Use Depth Map Shadows.**
3. **Enter 3 for the Filter Size.**

Depth map Filter Size controls the softness of shadow edges. A value of 0 gives hard edges. As the number increases, so does edge softness. The other shadow map attributes are okay at their default settings. You can preview shadows in the scene view as follows. In the Panel menus:

1. **Choose Renderer → High Quality Rendering.**
2. **Choose Lighting → Use All Lights.**
3. **Choose Lighting → Shadows.**

However, keep in mind that scene view shadows are extremely taxing on your computer system and will slow down interactivity considerably.

Light Linking

When you create a light, it automatically illuminates all visible objects in a scene. Conversely, when you add a new object to a scene, it receives illumination from all of the lights. With Light Linking you can specify which lights interact with which surfaces. In the real physical world, a cinematographer would have a difficult time achieving such a specific interaction between given lights and selected elements of the shot!

When using Render Layers, we avoid Light Linking. In our experience, the two features do not work well together.

In the present scene, you want only the key light to illuminate and cast shadows on the plane. The fill and back lights are meant for the hemoglobin subunits and not for the plane, so you will use Light Linking to disconnect them from the latter.

1. **Hit the hotkey F6 to activate the Render menu set.**
2. **Choose Lighting/Shading → Light Linking Editor → Object Centric. This launches the Light Linking window** (Figure 10.06).
3. **In the Illuminated Objects panel, select the polygon plane pPlane1.**
4. **The lights to which pPlane1 is linked are highlighted in the Light Sources panel.**
5. **LMB + click on fill_light and back_light in the Light Sources panel to unlink them from pPlane1.** Figure 10.06 **shows what the Light Linker should look like after you've taken this action.**

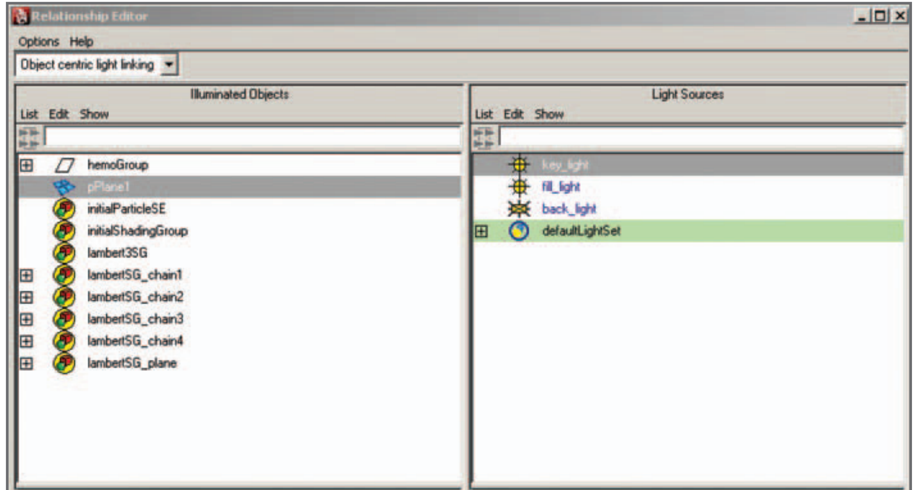


FIGURE 10.06

Light Linking is the practice of specifying which lights illuminate which objects in a scene. By default, all lights are linked to all objects. To unlink a light from an object, select the object in the left-hand pane, then click on a light in the right-hand pane; when it is unhighlighted, it is unlinked. In this example, fill_light and back_light have been unlinked from the plane, pPlane1. Light Linking will only show up in a rendered view, not in the interactive scene display.

Previewing with IPR

The roughly lit preview you see in the view panel for camera1 is actually hardware-rendered by your computer's graphics card. Interactive software previewing can be done using IPR, which renders a low-resolution image that updates automatically as you edit lights and shaders. Unlike your scene view, however, the IPR image will not update geometry or camera movements. You must therefore create a different IPR image for each point along the timeline you wish to preview. Let's start at frame 60, since that is your benchmark frame for lighting.

1. **Move the current time indicator to frame 60 and make the camera1 view active.**
2. **Press the IPR button in the Status Line to launch the Render View and start an IPR preview.**

This creates a group of temporary IPR render files. When you set up your Project, if you specified no directory for IPR images, they will be saved loosely in the Project directory. After the preview appears, which could take several seconds, a message appears at the bottom of the Render View window telling you to *Select a region to begin tuning*. By doing so, you tell Maya what region it should update as you adjust lights and shaders.

Back in *Chapter 09*, you set the Image Size to 320 × 240, or half-NTSC. This was done to make preview rendering fast—about one-quarter the time a 640 × 480 image takes. However, the smaller image size can make it difficult to see the subtleties of the lighting adjustments you're making. If this is indeed the case, set the Image Size to 640 × 480 in the Render Settings and see if the increase in preview render time is tolerable.

3. **LMB + Drag a selection box around the region you wish to see update (see our suggestion in** Figure 10.07**).**
4. **Press the Keep Image button to cache the current image for comparison later on with the edited lighting rig.**
5. **Take a few minutes to move the three lights around, adjust their Intensity values, and observe the effect in the Render View IPR preview. When you're satisfied with how the scene looks at frame 60, cache the most recent image using the Keep Image button.**

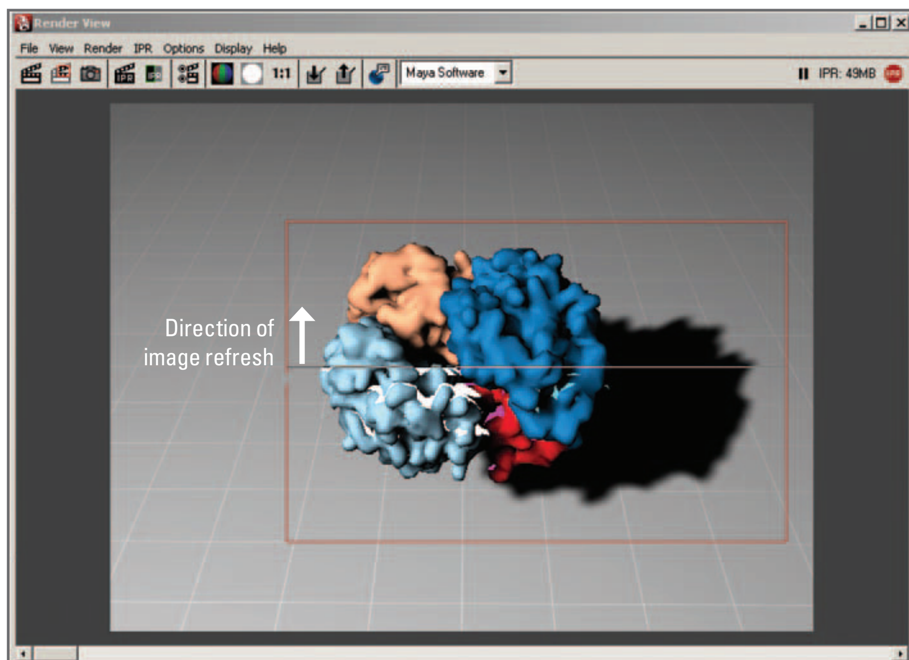


FIGURE 10.07

An IPR preview rendering. The red box indicates the "tuning" region—the area that gets updated each time you alter a light or shader. In this figure, the image is half-way refreshed after the **back_light** intensity was increased.

Before moving on to frames 1 and 120, make a software renderer preview to update the shaders and see what the final rendering will look like at this frame.

6. **In the Render view, press the Redo Previous Render button .**

If you are satisfied with the result at frame 60, it's time to see how things look at frames 1 and 120. Again, the goal is to get one lighting situation that works reasonably well from three vantage points. Repeat steps 1 through 5, above, with the current time indicator at frame 1 and then at frame 120. Figure 10.08 shows the final lighting setup we arrived at, software-rendered at frames 30, 60, and 120 (we omitted frame 1 because very little geometry is visible at that point). The light intensity values we used in the end were:

key_light: 1.25
fill_light: 0.5
back_light: 0.25

The right lights?

Because camera1 moves a considerable distance relative to the geometry, your standard 3-point lighting rig, which was set to work best at frame 60, is being stretched a little. Every scene requires some specialization of the lighting. Let 3-point lighting serves as a starting point for your own experiments with cinematic lighting principles; the key light provides the main source of directional illumination; the fill brings back some of the shadowy areas and really helps accentuate 3D—particularly curvilinear—form; the

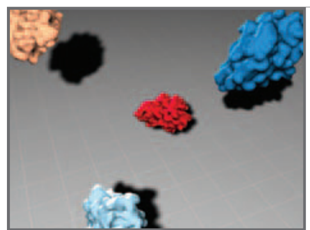
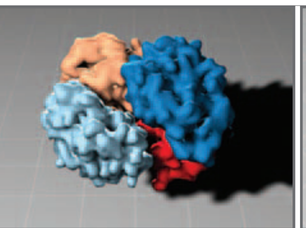
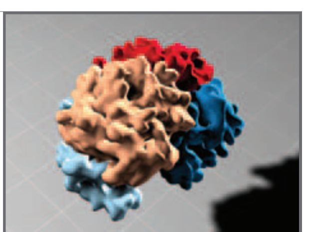




FIGURE 10.08

After tweaking the lights with the aid of IPR previewing, we arrived at a point where the key, fill, and back lights were all working well together to define 3D form. These images were rendered at (from left to right) frames 30, 60, and 120. We have enlarged frames 60 and 120 to make the geometry more visible on the printed page.

back light helps separate foreground geometry from the background and helps define contour.

There are several important aspects of shading, lighting, cameras, and rendering that we have intentionally skipped here. They are better left to later chapters where they can be discussed in the context of our MEL projects. Some notable mentions are shader **specularity** and **toon** (or **NPR) rendering** techniques. You'll work with them later in this book.

At this point your molecular actors are in place and their script—their animated activity—is set, your camera is in position, and your lights are showing off the spatial dance of molecules and lens to striking effect. It's time to shoot some virtual film footage. You'll find the finished Maya scene file for this tutorial on the CD-ROM:

10_Lighting/tutorial_10_01_done.ma

Summary

In this chapter we have discussed the manipulation of light and shadow in Maya. When used well, light and shadow in a 3D scene define form and spatial relationships the way they do in the real world; you can even amplify these effects in ways that are physically impossible outside of the computer. There are several types of light in Maya, each suited to a particular mode of real-world lighting. Though we haven't touched on it in this chapter, it should be mentioned that lights can also be assigned colors, which can further their expressive range. Three-point lighting is a technique used by studio photographers to define form and contour. It is easily emulated in Maya using two Point lights and an Area light, and makes a great departure point for your work with Maya cameras and lights.

This page intentionally left blank

11 Action! Maya rendering

Rendering

On a live-action film set, a shot is finally set in all its specifics of actors, action, costumes, camera, and lighting. Someone says, “Action!” and film rolls through the motion picture camera. (Or if it is a digital motion picture camera, the digital cinema frames are downloaded quickly from the camera to large-capacity disk storage.) The exposed film is sent to the processing lab where it is developed, color corrected, and printed, then returned to the studio for a look. Your production workflow in Maya has a step equivalent to the film’s exposure and developing; it is called rendering. When you have rendered your animation, Maya has produced the digital film frames (2D image files) of your computer animation at final quality. Your material can then at once be viewed and prepared for the postproduction steps like editing scenes together into a longer, complex film report of your work.

Maya creates 2D image files using a program called a **render engine**, or **renderer**. As of release Maya 8.5, Maya includes four **renderers**. These are **Maya Software**, **Maya Hardware**, **Maya Vector**, and **mental ray for Maya**. The first two are native to Maya and load automatically when you start the program. The last two are bundled with Maya as Plug-ins and are loaded by default when you first start Maya. If either of them is not included in the list of available renderers in the Render Settings editor, you can load it as follows:

1. **Choose Window → Settings/Preferences → Plug-in Manager.**
2. **To load mental ray for Maya, check "loaded" next to Mayatomr.mll. To load the Maya Vector renderer, check "loaded" next to VectorRender.mll.**
3. **Check auto load next to either Plug-in if you want it to load when you start Maya.**

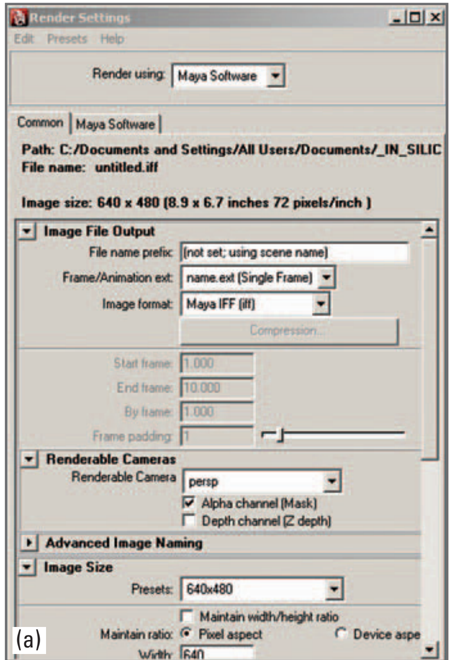
Render Settings

The Render Settings (part of which is shown in Figure 11.01) is an editor used to select a renderer and customize the output of rendered files. It displays two or more tabs, one for attributes that are common to all renderers and one for the active renderer. We will explore Render Settings in the upcoming tutorial.

Batch rendering

Batch rendering is done by an application external to Maya, enabling you to continue working on your project while images are created behind the scenes. You can batch render a still frame or an animation. If you start a batch render from within Maya, it uses the renderer and settings currently specified in the Render Settings.

A **command line render** is a batch render started by executing the render command from either the Command Line or the Script Editor in Maya, or from a **command prompt** external to Maya. A command line render external to Maya will use the Render Settings (including the specified renderer) saved in the file you’re rendering and the Project settings that were saved the last time you closed Maya. Alternately you can type in **flags**, or special instructions, with the render command in the Command Line in order to override the saved Render Settings.



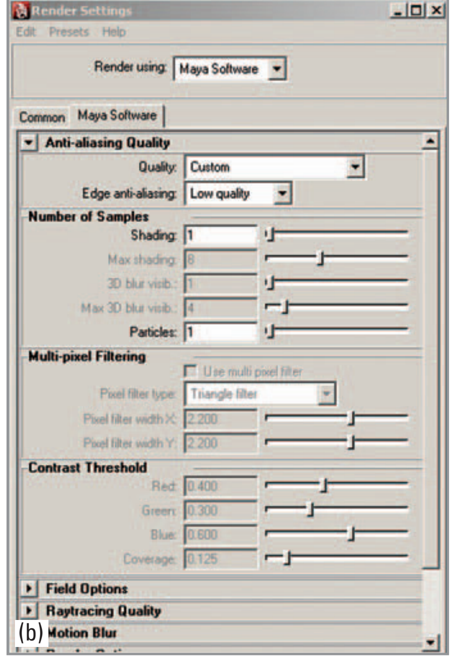


FIGURE 11.01

The Render Settings editor is used to select a renderer and to specify image size and quality. The Common tab (a) contains attributes common to all Maya renderers. Each renderer has its own tab (b) with settings unique to it.

Although you can execute a render while continuing to work in Maya proper, doing so will result in slower than normal performance due to the allocation of system resources to the render engine. The real advantage to the stand-alone batch renderer is the external, command line rendering capability; it allows you to execute renders without having to open the Maya application, which uses valuable memory. In general, batch renders that are executed from the Command Line run faster when Maya is closed.

Each operating system has its own name for the system Command Line (external to Maya). In Windows, it is known as the Command Prompt, In Mac OS, the Terminal Window, and in Linux, the Shell.

Batch rendering

Maya Help → Using Maya → Rendering and Render Setup → Rendering → Rendering menus → Render menu set → Render → Render → Batch Render

Command line renderer

Maya Help → Using Maya → Rendering and Render Setup → Rendering Utilities → Command line renderer → Command line renderer

The Maya Software renderer

This is the default renderer in Maya and is capable of both **scan-line rendering** and **raytracing**. The scan-line technique renders a scene in rows based on the intersection of a scan line with the scene geometry. It is faster than raytracing, which is a more accurate physical simulation that calculates the illumination of objects, their shadows, and reflections, based on the paths of imaginary photons emitted from light sources in the scene (it actually works backwards, emitting "reverse photons" from the scene

The name "Maya" precedes the native renderers to distinguish them from third-party renderers like mental ray for Maya.

viewpoint). Raytracing can create certain effects not available in a scan-line render. These include shadow casting inside transparent objects, physically realistic shadows, refractive distortions in transparent materials of different densities, and **caustics**, the illumination patterns created when light reflects from and refracts through objects. Raytracing is not turned on by default; it must be activated in the Render Settings.

The software renderer must be used to render effects created by Maya's 3D painting tool, Paint Effects.

Multi-processor support

The Maya Software renderer supports **multithreading**, meaning it can be set to use more than one CPU on a **multi-processor** machine. This feature can literally half the time required to render a scene. Maya automatically attempts to exploit hyperthreading as long as `"Use All Available CPUs"` is checked in the Multi Processing section of the Maya Software tab in the Render Settings.

The Maya Hardware renderer

The Maya Hardware renderer uses your computer's video graphics card to render images faster than the software renderer. Due to the way it processes images, the hardware renderer does not support raytracing or Paint Effects. Nonetheless it works with a wide variety of Maya shaders and lighting effects, including transparency and shadows. Furthermore, a person skilled in compositing can do a lot to create sophisticated visual effects using only hardware-rendered animation. Because compositing involves 2D footage rather than 3D scenes, it is often much faster to create visual effects in the compositing rather than 3D rendering stage. For example, depth-of-field effects are, in our opinion, best left for compositing.

Your computer must have a Maya-compatible graphics card in order to hardware render scenes. On the official Maya website, you will find links to lists of qualified hardware for each release of Maya. If your card is unable to handle Maya Hardware rendering, you may get the following message:

```
// Error: Graphics card capabilities are insufficient for rendering.
Render aborted
```

The Hardware Render Buffer

The Hardware Render Buffer is a handy item to add to your custom shelf.

Before the Maya Hardware renderer was brought up to current level of sophistication, we usually relied on the **Hardware Render Buffer** (**HRB** for short) to make fast hardware renders. The fact that it doesn't match the hardware renderer proper for image quality and effects capabilities makes it a less attractive choice for final renders. Nonetheless, it is still a regular item in our tool set. One advantage it offers over the other renderers is the ability to view an animation while it's being rendered. In contrast, when you render an animation (as opposed to a still frame) using one of the other four renderers, you can't see the images as they're created.

The Maya Vector renderer

A 2D picture on a computer screen is either a raster graphics image or a **vector graphics** image. A **raster graphics** image is a rectangular grid of pixels, each with its own color value. As you enlarge a raster image, the pixels become visible. Vector graphics images, on the other hand, are drawn using mathematical curves and polygons whose

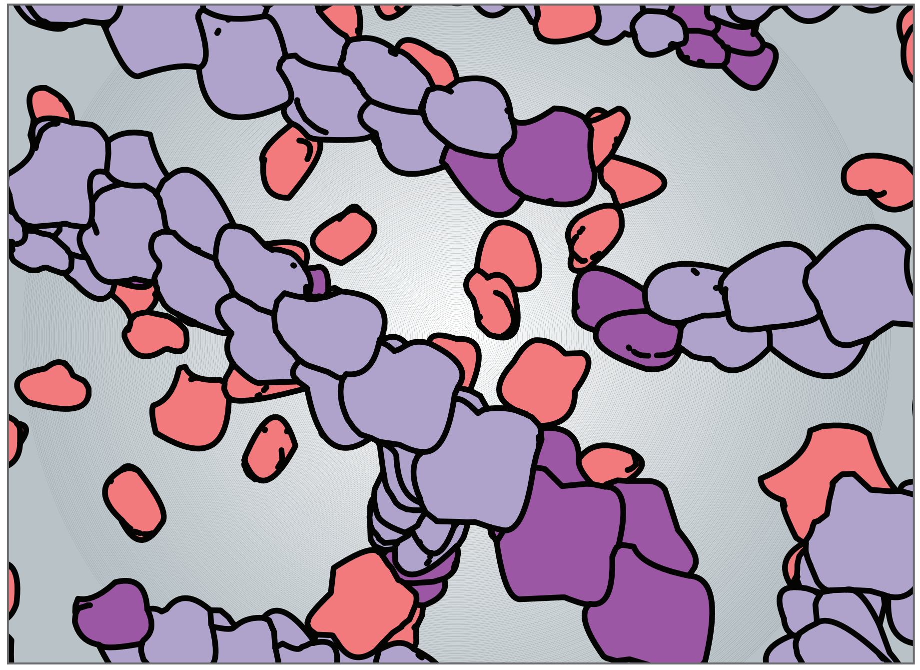


FIGURE 11.02

The Maya Vector renderer creates vector images that are defined by colored strokes and fills. This image, from a simulation of actin protein filament (or polymer) assembly, uses different shaders for different sub-populations of molecule: pre-existing polymers (mauve), newly added polymer subunits (purple), and unassociated monomers (red) (the individual building blocks of a polymer).

shapes are defined by directed line segments, or vectors running between points in space. They are resolution-independent and will always look sharp when enlarged.

The subject of vector graphics will be familiar to users of illustration and drafting applications like Adobe Illustrator, Autodesk AutoCAD, and that ubiquitous 2D animation and web design software, Adobe Flash (formerly by Macromedia). Vector images are characterized by fills and strokes. Fills are solid colors or gradients applied to enclosed areas. Strokes are colored lines that follow the boundary curves that define those areas. When you render with the Maya Vector renderer, the effects of lights and shaders in your scene are converted to strokes and fills. In Render Settings you will find options for how fills and strokes are rendered and for the output file format. If you choose one of the vector file formats, like **.swf** or **.eps**, Maya will output a resolution-independent vector file. If you select a raster file format, such as **.tga** or **.tif**, the strokes and fills will be converted back into pixels and saved in a raster file. Most render file formats produce individual image files. To be viewed as animation, the images must be opened in a movie player like fCheck or QuickTime. In contrast a **.swf** file is a self-contained Flash movie that can be opened and played directly in **Flash Player** software which is available free of charge from Adobe's Website: http://www.adobe.com/products/flashplayer/.

The Maya Vector renderer is the obvious choice if you want to output files for use in a vector graphics program like Flash or Illustrator. It also makes it very easy to get NPR toon shading features (see Figure 11.02) without the need for custom shading networks or Maya Paint Effects. You can specify fill and stroke colors, and stroke weights, for different materials in your scene. This feature supports one of the key strengths of **toon shading** for scientific interpretive visualization: the use of line weight and color to highlight different properties of data visually. If you wish to draw on this strength

in your projects but also want the smooth-shaded appearance of software rendering, an approach using either a custom mental ray shading network or Maya Paint Effects is preferable to vector rendering.

Loading the Vector renderer plug-in

If Maya Vector does not appear on the list of available renderers in the Render Using menu of the Render Settings, you can load it using the Plug-in Manager as follows:

1. **Choose Window → Settings/Preferences → Plug-in Manager.**
2. **Check the loaded box beside VectorRender.mll, the Maya Vector plug-in.**
3. **Check auto load if you want the plug-in to load each time you start Maya.**

The mental ray for Maya renderer

mental ray for Maya is a third-party renderer which uses **raytracing**, a physical simulation that calculates the illumination of objects, their shadows, and reflections, based on the paths of imaginary photons emitted from light sources in the scene. The developers have improved the integration of mental ray with each subsequent release of Maya. The result, in addition to making mental ray easier to use, is that some shading and rendering techniques that are used in recent versions of Maya may have to be approached differently in earlier versions of the software.

Maya includes a host of render nodes specific to mental ray for Maya. As DG nodes, they connect one to another to form shading networks in the same way as Maya render nodes. mental ray nodes greatly extend Maya's rendering effects capabilities, with features like **subsurface scattering** which is an advanced simulation of translucence (Figure 11.03). Unfortunately the documentation on mental ray for Maya isn't nearly as extensive as that for the other Maya renderers. For this reason, we recommend getting comfortable with Maya shading networks and the Maya Software renderer first before diving into mental ray. To list the mental ray render nodes:

1. **In the Hypershade:**
   (a) **RMB+click on the Create Bar.**
   (b) **From the Marking menu, select mental ray nodes.**

or

2. **In the Create Render Node window:**
   (a) **In the Hypershade menu bar, choose Create → Create Render Node.**
   (b) **Click on the mental ray tab.**

or

3. **Open the Create menu. The mental ray nodes are listed by category under the regular Maya render nodes.**

Like the Maya Vector renderer, mental ray for Maya is a plug-in. By default it is autoloaded when you start Maya. If it does not appear on the list of available renderers in the Render Settings window, you can load it by checking `load` next to `Mayatomr.mll` in the Plug-in Manager.

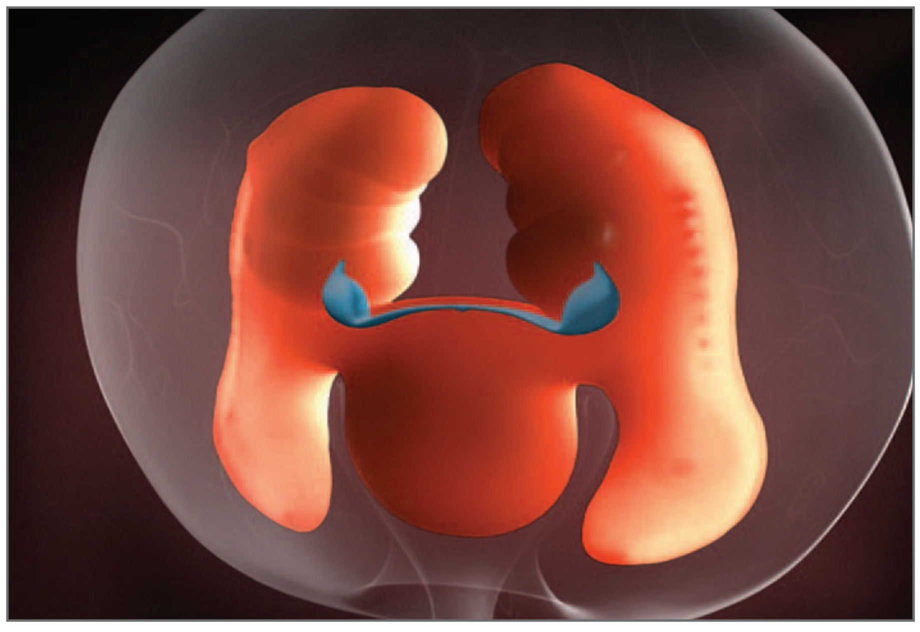


FIGURE 11.03

mental ray for Maya extends Maya's rendering effects capabilities with features like subsurface scattering (pictured here), which simulates the absorption and scattering of light beneath the surface of an object.

Courtesy Ellis Entertainment and AXS Biomedical Animation Studio. Copyright © 2006 Ellis Entertainment.

Descriptions of the different renderers

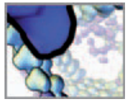
Maya Help → Using Maya → Rendering and Render Setup → Rendering → About rendering and renderers → Renderers → Maya Software renderer
Maya Hardware render
mental ray for Maya renderer
Maya Vector renderer

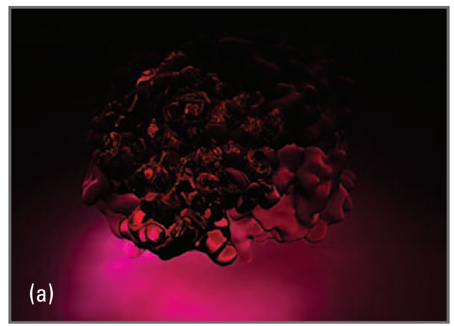
Renderer Settings for the Maya Software, Hardware, and Vector renderers

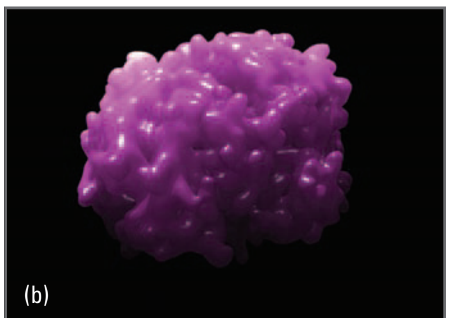
Maya Help → Using Maya → Rendering and Render Setup → Rendering → Rendering Windows and Editors → Render Settings →
Render Settings: Maya Software tab
Render Settings: mental ray tab
Render Settings: Maya Hardware tab
Render Settings: Maya Vector tab

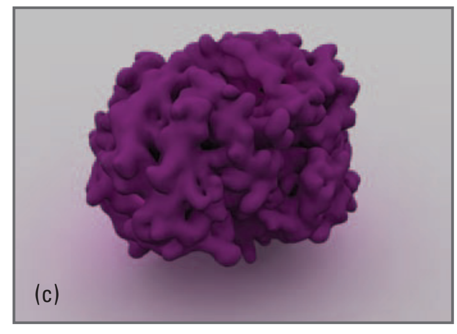
Advanced rendering techniques with the mental ray for Maya renderer

The developers of mental ray for Maya have incorporated numerous advances in 3D rendering that mimic real-world lighting and material situations. While these can improve the photorealism of a rendering, they can add substantially to your render times. Figure 11.04 shows examples of several of these rendering features.









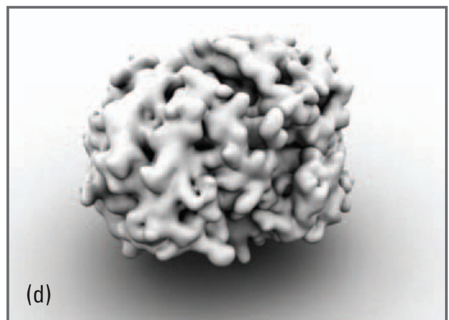


FIGURE 11.04

Examples of advance rendering techniques using the mental ray for Maya renderer. The molecule pictured here is hemoglobin.

(a) Caustics.

(b) Subsurface scattering.

(c) Global illumination.

(d) Ambient occlusion.

Courtesy of Eddy Xuan.

Caustics

Caustics refers to the light patterns created by specular refraction through materials such as glass and water, or reflection from materials like chrome. See the example in Figure 11.04a.

Caustics

Maya Help → Using Maya → Rendering and Render Setup → Lighting → mental ray for Maya lighting → Global illumination and caustics → Caustics

Subsurface scattering

Translucence is an attribute common to all Maya material nodes. It is the same in principle as subsurface scattering but works with the Maya Software renderer rather than mental ray for Maya. The choice between Translucence and subsurface scattering depends on which renderer you use.

The absorption and scattering of light beneath an object's surface is a property of many real-world materials, including organic tissues like skin. Maya can simulate this phenomenon with two mental ray for Maya **subsurface scattering** shading networks: a fast, non-physically correct one (Figure 11.04b) and a slow, physically accurate one. Subsurface scattering is available in Maya 6.0 (mental ray 3.3) and later.

Subsurface scattering (Maya 6.0 or later)

Help → Maya Help → Using Maya → mental ray → mental ray Shaders Guide → Subsurface Scattering Shaders

Global illumination

Global illumination (**GI**) (Figure 11.04c) simulates real-world lighting by accounting for the indirect light that has been reflected off of all objects in a scene. This technique can make for long rendering times, so it should be used selectively when rendering animations. However, it is well suited to creating realistic lighting effects in still images, where the wait time is less of a concern than for animations where hundreds or thousands of images are often rendered.

Global illumination

Maya Help → Using Maya → Rendering and Render Setup → Lighting → mental ray for Maya lighting → Global illumination and caustics → Global illumination

Ambient occlusion

Ambient occlusion (**AO**) (Figure 11.04d) is similar to GI but quicker to render and cruder. It calculates the attenuation of indirect light by nearby objects, in order to produce the realistic shadow effects resulting from even, ambient light. The resulting image is typically monotone and used in compositing to enhance the illusion of 3D. Since Maya 7, ambient occlusion has been incorporated into Render Layers (more on this shortly) as a Preset called **occlusion.** When activated, occlusion creates a mental ray for Maya shading network using the `mib_amb_occlusion` node, and connects it to the geometry in the layer.

Render Layer presets: occlusion

Maya Help → Using Maya → Rendering and Render Setup → Rendering → Visualize and render images → Visualize scenes and render images → Work with Render Layers → Work with layer presets

mental ray for Maya mib_amb_occlusion node

Maya Help → Using Maya → mental ray → mental ray Shaders Guide → Base Shaders → Illumination

Search: **mib_amb_occlusion** on the Illumination Help page

Image-based lighting

Image-based lighting (**IBL**) uses an image file as a source of illumination or reflection in order to create natural-looking lighting conditions in a scene. Traditionally, the image is applied to a sphere which encloses the scene. However, this technique is slow to render because the sphere geometry is factored into the raytracing calculations. A more efficient approach, introduced Maya 7.0, uses a special `IBL` node, which controls IBL with attributes and eliminates the need for geometry.

IBL works best with **high dynamic range imaging** (**HDRI**) files. In addition to color data, each pixel in an HDRI image stores a luminance value. IBL uses both the color and luminance data to calculate the lighting situation in a scene.

Maya Help → Using Maya → Rendering and Render Setup → Lighting → mental ray for Maya lighting → Final gather and HDRI → Image-based lighting (sky-like illumination)

Maya Help → Using Maya → Rendering and Render Setup → Lighting → mental ray for Maya lighting → Final gather and HDRI → High dynamic range imaging (HDRI)

Realism about photorealism

Before using one or more of these photorealistic techniques, you may want to ask if this sort of *realism* is a worthwhile rendering goal. Certainly, on cellular and molecular scales, photorealism has no meaning other than a projection of macro-world ideas about light and materials onto micro- and nano-world entities. We're not suggesting that you dismiss Maya's capabilities for photography-like rendering complex real-world lighting scenarios. Instead, we propose that they be considered for their merits in relation to the communication goals of each project.

For example, caustics could be used to give a gel-like appearance to the cytoplasm of a cell; not because we think the cytoplasm would appear to distort and reflect light in such a way, but because the visual association of caustics with macro-scale gelatinous materials could help suggest the mechanical properties of the cytoplasm, which is a gelatinous combination of water and protein. Similarly, ambient occlusion may lend itself well to visualizing an intricate nano-structure, such as a bioengineering scaffold. Again, not because it gives a *realistic* picture of nano-scale light and shadow, but it gives a beautiful sense of 3D form.

Render Layers

Maya's Render Layers features let you organize items in your scene (objects, cameras, lights, even shaders) into different groups or layers. Each layer is then rendered individually. Render Layers have a number of presets such as ambient occlusion (discussed above), specular, and shadow with which you can render those effects individually as well.

Using Maya → Rendering and Render Setup → Rendering → Visualize and render images → Layers and passes → Render Layer overview

Tutorial 11.01: Batch rendering

Batch rendering is used to render an animation into a sequence of image files. In this tutorial, you will adjust the Render Settings and then batch render the hemoglobin

FIGURE 11.05 Render file naming.

animation using the Maya Software renderer. Begin by opening the scene you created in the last chapter or copy the ready-made scene file from the CD-ROM.

11_Rendering/scenes/tutorial_11_01.ma

In the next two sections we'll explore the Render Settings window:

1. **Choose Window → Rendering Editors → Render Settings.**
2. **From the Render Using menu, choose Maya Software and make the Common tab active.**

Common Render Settings

The following attributes are common to all renderers, including mental ray for Maya.

Image File Output

In this section, you will specify the file name, image format, and rendering frame (or time) range. When you render an image sequence, the file names must include an image number so that the files will be recognized as a sequence by a movie player or editing application, and be read in the correct order. Figure 11.05 shows a sequence of image file names and labels their components, which you will set below.

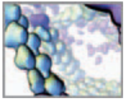
3. **Make the following settings:**

File Name Prefix	**hemoTest**
Frame/Animation Ext	**name.#.ext**

This last setting, **name.#.ext**, determines the order of file name components. The **name** component will be assigned your entry for `File Name` Prefix: **hemoTest**. The # component will be assigned the number of the frame being rendered. And finally, the **ext** component will be assigned the appropriate file extension, **.tif** for a TIFF file, for the file format, which you will specify in step 4 below.

Maya supports a large number of image file formats. The one you choose will depend on how you plan to use the images. Most compositing applications used in the postproduction phase support a wide range of file types too. If you're creating an animation for an online journal publication or for broadcast at a conference, the publisher or media coordinator may request a specific file type.

Furthermore, there are functional differences between certain image formats. For example, Wavefront RLA (file extension .rla) supports five image channels: R, G, B, alpha, and depth all in one image file. Targa (file extension .tga) format, which is popular in production for television and movies, produces two separate files for each



animation frame: one containing the R, G, B, and alpha channels, and the other the depth channel. The popular TIFF format has R, G, B, and alpha, but no depth channel. Maya has its own native format, IFF, which is popular with commercial animation studios. Nonetheless, we tend to avoid IFF files due to compatibility issues with the compositing software we use. As a general rule, we render RLA files when we need the depth channel for compositing (depth-of-field effects), and TIFF files when depth is unnecessary. In this tutorial you will render a TIFF image sequence, which you will then view using Maya's movie player fCheck.

4. **Enter the following settings:**

Image Format	**TIFF (.tif)**
Start Frame	**1.000**
End Frame	**120.000**
By Frame	**1.0**
Frame Padding	**4**
Camera	**camera1**
RGB Channel (Color)	✓
Alpha Channel (Mask)	✓

A depth channel can be rendered in a separate TIFF file when using Render Layers. Render Layers are used to produce separate image passes, such as color, depth, and shadow. When you render the depth pass, it will render as a separate single-channel file, not as a channel in a multichannel file.

You can leave Depth Channel unchecked since it won't be rendered in the TIFF format. Frame Padding sets the number of decimal places reserved for the frame number. A setting of 4 is good for sequences less than 10,000 frames in length. You have no need presently for `Custom File Extension or Renumber Frames`, so you can skip those sections.

Image Size

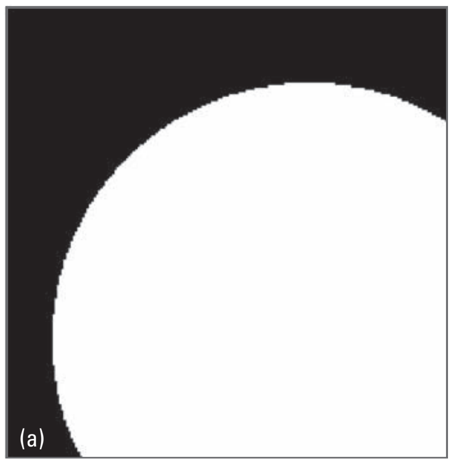
There are a number of standard video formats you can choose from in the Preset menu. When producing animation for video playback on a television, it is important to render in the required broadcast format. Widely used formats are NTSC for video in North America and PAL in Europe. DV (for Digital Video) format, which applies to miniDV video and DVD-Video, uses non-square pixels. In this case, when a frame is rendered, each pixel in the image is compressed to 90% of its width. On playback, it is stretched back to its original square shape. Non-square pixels are denoted in the Render Settings by a Pixel Aspect Ratio of 0.9. When you choose one of the non-square Image Size presets, this ratio is automatically set.

If your animation is likely only to be seen on a computer screen, you can avoid the broadcast presets and choose a standard format like 640 × 480. The first time you render this animation you may want to use a small image size for quicker results. On our benchmark computer system, a single frame rendered as follows, for three different image sizes:

The shorthand, "px", denotes "pixels".

Small (1×)	**320 × 240 px:**	**2 seconds**
Medium (2×)	**640 × 480 px:**	**7 seconds**
Large (3×)	**1280 × 960 px:**	**24 seconds**

You can see that, for this example, the increase in rendering time is roughly in direct proportion to the increase in image dimensions. Guided by these results, you can forecast that your batch render of 120 frames will take approximately 14 minutes for a 640 × 480 px rendering versus 4 minutes for one that's 320 × 240 px in size. It's much quicker to render the smaller size, which can be helpful if you discover



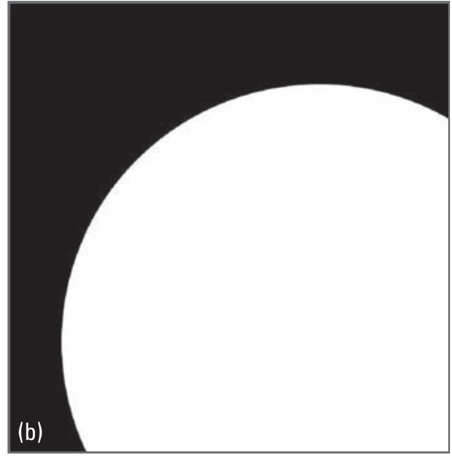


FIGURE 11.06

Edge aliasing (or "jaggies") is one of the rendering artifacts affected by the anti-aliasing attributes in the Render Settings. Without anti-aliasing, edges appear jagged.

(a) Without edge anti-aliasing.

(b) With edge anti-aliasing.

problems with your rendered animation that you wish to fix. Assuming that you're using the smaller render size:

5. **Enter the following settings:**

Presets	**320 × 240**
Size	**Units pixels**
Resolution	**72**
Resolution	**Units pixels/inch**

When you choose a preset, the Width and Height fields are set automatically.

The Pixel and Device Aspect Ratios can be left as is. The former is set by Width and Height and the latter by the Preset choice; 320 × 240 is a square pixel preset, therefore the ratio will be 1. If you are rendering for print and know the image resolution requirements, you can choose units other than pixels, including inches and centimeters, and specify a resolution in pixels per unit. For example, suppose a journal requested that you supply a single frame from your animation as a 4 inch × 6 inch picture at 300 **dpi** (for **dots per inch**). In Maya, you would set Width and Height to 4 and 6, respectively, set Size Units to inches, and resolution to 300.

The remaining settings—those under Render Options—don't apply to this exercise and can be ignored for the time being.

Maya Software Render Settings

Most of the settings particular to the Maya Software renderer will be fine at their default values for this exercise. We will focus on those that most affect image quality.

1. **Click on the Maya Software tab in the Render Settings to make it active.**

Anti-aliasing Quality

Anti-aliasing diminishes unwanted visual artifacts typical of low-resolution computer displays. One type of artifact that it resolves is edge aliasing (Figure 11.06), also known as the "jaggies". Quality is an over-arching setting that influences the remaining attributes in this section. Until you desire a more in-depth understanding

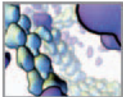


FIGURE 11.07

The render file path is displayed along the top of the Common tab in the Render Settings.

of these attributes, the Quality settings of Preview and Production will suit your requirements for fast, low-quality and slower, high-quality renderings, respectively. The Quality setting automatically adjusts Edge Anti-aliasing.

2. **Choose Quality → Production Quality.**

Leave the Render Settings window open for the time being. For a complete description of Render Settings for any of the renderers, see the Maya Help section on the Render Settings window.

Render Settings

Maya Help → Using Maya → Rendering and Render Setup → Rendering → Rendering Windows and Editors → Render Settings → Render Settings window

Hit *Render*!

Wait! Before you hit Render, check to make sure the render files will go where you want them to in your file system. If you haven't set your project as described back in *Chapter 04*, do so now. The render file destination directory (usually called "images") is displayed at the top of the Render Settings window (Figure 11.07). You may have to widen the window to see the path name on one line.

Now you can hit *Render*:

1. **(a) With the Render menu set active, choose Render → Batch Render ❐.**

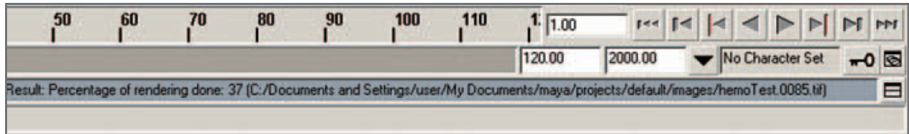


FIGURE 11.08

The Command Line at the bottom of the main window displays information about each frame as it's being rendered.

(b) If you are rendering on a multi-processor machine, check Use All Available Processors. If not, leave the box unchecked.

(c) Press Batch Render.

or

2. (a) Activate the Render shelf.

(b) Press the Batch Render button in the Render shelf.

Information about the render will appear in the result field of the Command Line (Figure 11.08). This tells you what frame is currently being rendered, the percentage of the frame that is done rendering, and the path to the directory where it's being saved. When rendering is complete, the Command Line will display the following text:

```
Result: Rendering Completed. See mayaRenderLog.txt for information.
```

The **Render Log** is a text file to which Maya writes information such as the time taken to render each frame. The file is named `mayaRenderLog.txt` and is located in your Maya user account directory.

On the CD-ROM, we have included a copy of the completed scene file, along with the rendered sequence of rendered TIFF files.

11_Rendering/scenes/tutorial_11_01_done.ma

11_Rendering/images/hemoTest.0001.tif *etc.*

Software versus hardware rendering

The hemoglobin scene you've been working on is considered *light* in the world of Maya scene files. In other words, it didn't tax your system resources heavily and was well suited to rendering with the Maya Software renderer. As the complexity of your work in Maya increases, it is worthwhile becoming familiar with the Maya Hardware renderer. In many cases, it delivers results as good as the software renderer but in a fraction of the time.

Tutorial 11.02: Playback using fCheck

Now, after all that work—shading, camera setup, lighting, and render settings—comes the moment you've been waiting for: seeing your rendered animation in action. In this tutorial, you will view the rendering of hemoglobin, using Maya's own **fCheck** (short for "file check") previewing application. fCheck was installed when you installed Maya. In Windows, a shortcut was placed in the Start menu. The application itself is located in Maya(version)/extras/bin within your applications directory. If you did *not* render the animation sequence in the previous tutorial, copy the 120 TIFF files from the CD-ROM to the image directory within your Project directory.

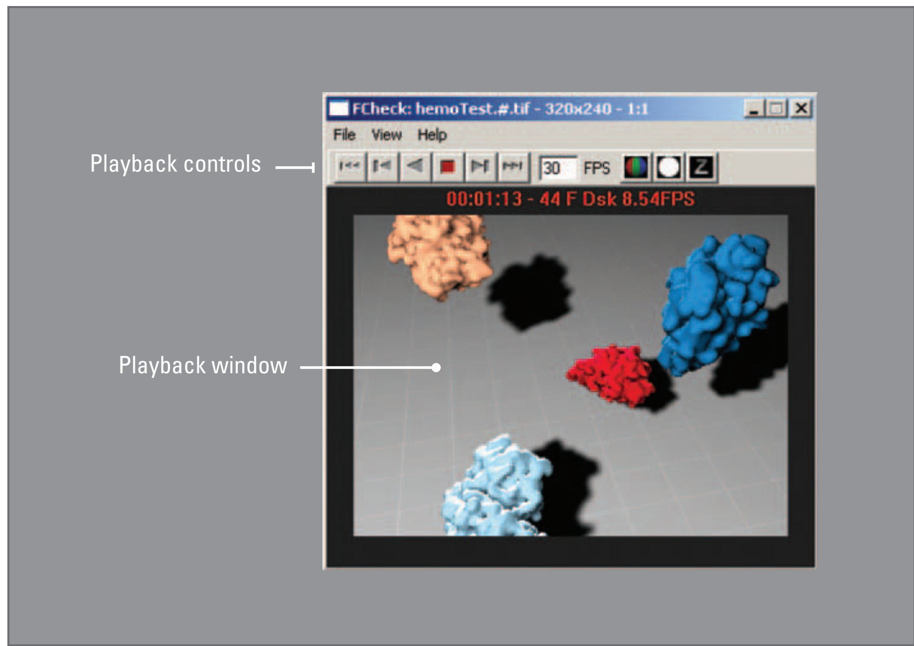


FIGURE 11.09

fCheck is an application external to Maya that is used for viewing rendered animations and still images. It has an extensive suite of functions for adjusting and resizing the displayed image and for controlling animation playback. Many of these are accessed through hotkeys.

11_Rendering/images/hemoTest.0001.tif *etc.*

Playback in fCheck

1. **Launch fCheck by locating it on your hard drive and**

   (a) double-clicking its icon

   or **(b) RMB+click on its icon or name and selecting Open.**

2. **In the fCheck menu bar, choose File → Open Animation.**
3. **Navigate to the image file sequence in your <project>/images directory.**
4. **Select the first file in the list, hemoTest.0001.tif.**
5. **Press Open.**
6. **Enter "30" in the FPS field.**

Playback will likely be slow until all frames have been loaded, at which point it will speed up. Basic playback controls are available in the Control Bar for the fCheck window (Figure 11.09). Additional control over playback and to adjust the display (zoom, channels, etc.) and image (luminance, saturation, etc.), are available through hotkeys and mouse controls. You can enter any FPS rate you like in order to see your animation playback at slower- and fast-than-normal speeds. The actual playback rate is displayed in red type at the top of the Playback window. If your computer is unable to play the animation at the rate you specify, hitting the hotkey "T" forces to fCheck to skip frames in order to meet the prescribed rate. Another handy fCheck feature is the

ability to scrub back and forth through the animation interactively by LMB+dragging in the Playback window.

Saving from fCheck

If you make image adjustments in fCheck, you can save your animation under a different name and the changes will be saved as well. If you're using Maya in Mac OS X you have the option to save out a self-contained QuickTime movie file. In either case, to save out of fCheck:

1. **Choose File → Save Animation.**
2. **In the Save As window, navigate to an appropriate directory or create a new one.**
3. **Enter a file name and select the Image type.**
4. **Press Save.**

Third-party applications

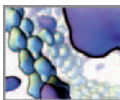
It is often desirable to produce a self-contained movie file of an animation sequence. Unless you're using Maya for Mac OS X—in which case you can export a QuickTime file from fCheck—you can use a third-party application such as Apple QuickTime Pro or Adobe After Effects to save out your image sequence in a popular video format like QuickTime (file extension **.mov**), Windows Media Player (file extension **.wmv**), or Audio Video Interleave (file extension **.avi**). Such formats can be viewed using free movie player applications like Apple QuickTime Player and Windows Media Player, making it feasible to distribute your movies to colleagues, clients, or students over the Internet or on portable media, such as CD-ROM/DVD-ROM.

Summary

Rendering is the creation of image files from a Maya scene. It is the final stage in the 3D production workflow and includes a series of creative steps covered in the last four chapters:

- **Shade/texture**
- **Camera setup**
- **Lighting**
- **Rendering**

The final step of the rendering workflow involves customizing the Render Settings and then batch rendering your animation. The choice of renderer is usually made early on, before making Render View previews to test shaders, lights, and camera setup. A batch render can be started from within Maya, which allows you to continue working on a scene while rendering happens in the background. Alternately, you can execute a batch render from your computer's Command Line, without opening Maya. We concluded our discussion of rendering with a brief summary of some of the advanced rendering techniques available through mental ray for Maya.



The hemoglobin tutorials let you try out these workflow steps on a small but meaningful project. Effective visualization of big, complex biomolecules is a frequently requested deliverable in computational cell biology. The hemoglobin scene presented a challenge typical of rendering in silico cellular environments: how to light and shade multiple, simultaneously moving objects for a moving camera view. You learned that materials can be easily created, applied, and edited using the Hypershade and the Attribute Editor.

In *Chapter 09*, you saw that a camera to view the virtual molecule can be animated in the same way as the molecule's geometry. Motion paths, which are commonly used to control the animation of geometry, are an easy way to animate a camera along a predictable trajectory. With the camera animation set, *in Chapter 10* you lit the scene with a classic 3-point lighting rig. IPR previewing in the Render View helped you adjust the lights for optimal effect at the beginning, middle, and end of the animation. When you were happy with the lighting, in this chapter you adjusted the Render Settings to prepare for batch rendering the animation. In the Common render attributes, which apply to all Maya renderers, you set the file name, frame range, and image size. Each renderer has its own tab in the Render Settings. You used the Maya Software tab to set the image quality. When the settings were done, you batch-rendered the animation to the image folder in your current project.

In this chapter's second tutorial, you used the stand-alone application, fCheck, to preview your finished animation. If you work in Maya for Mac OS X, you may have also used fCheck to export a self-contained QuickTime movie file.

Rendering is an essential part of the in silico workflow. If you're producing an animation for a client, the rendering is what you deliver; it's the product. If you're using Maya to run simulations and generate data, rendering is how you report your results as visual images. While not everyone has access to Maya in order to open and review your scene file, anyone with a computer and an Internet connection can view a rendering of your scene file using freely available software.

The rendering needs of a biomedical communicator creating content for a big-budget pharmaceutical video may differ substantially from those of a scientist publishing results on a Maya-based protein folding simulation. Your new skills with materials, cameras, lights, and renderers equip you for initial projects and, we hope, will set you on an exciting journey to learn more. The learning resources available on the subject are substantial. Further exploration will no doubt guide you to the tools and techniques suited to your specific needs, and enable you to innovate further in the visualization of molecules, cells, and tissues.

Astonishingly, most of the attributes involved in Maya materials, cameras, and lights are open to automated control through the MEL language, just as MEL enables access to the geometry, dynamics, and physics of your scene elements. Now that you have established a foundation in Maya, it is time to take command of your virtual world with MEL.

```
move -a 0 ($H/2) 0 $name1;
move -a 0 (-$H/2) 0 $name2;
$groupName = `group $name1 $name2`;

   if ($j==0){

      // Create the first peptides.
      $locatorName1 = `spaceLocator
      move $W 0 0 $locatorName1;
      parent $locatorName1 $groupName

      move -r $x $y $z $groupName;
      rotate -r $rx $ry $rz $groupNa

      // Increment the helix rotatio
      $rx = ($rx + $helix);
   }
   else { // Create the next peptide

      // Store the translate values
      $xyz1 = `xform -q -t -ws $loca
      $x = $xyz1[0]; $y = $xyz1[1];

      // delete the previous locator
      delete $locatorName1;
      $locatorName1 = `spaceLocator
```

This model of a "nano-trellis" composed of self-assembling peptides was built and animated entirely using MEL.

Image courtesy Shaftesbury Films and AXS Biomedical Animation Studio. Copyright Shaftesbury ReGenesis III Inc.

12 MEL scripting

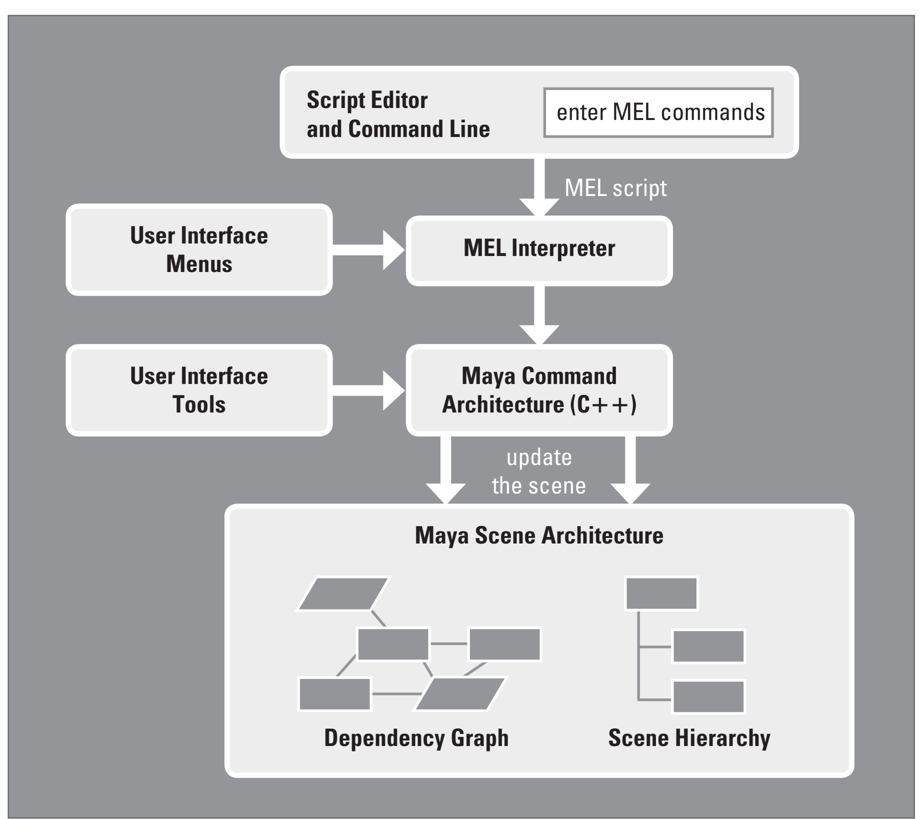


FIGURE 12.01

Maya's command architecture receives MEL commands from UI menu selections and tools, and from user input via the **MEL interpreter**. The result is a change to the DG, the Scene Hierarchy, or both. The compiled commands in the Command Architecture, were written in the C++ programming language. Users can add custom commands to the command architecture using the application programming interface (**API**) available through Maya's Developer's Toolkit. Furthermore, although MEL is used to implement much of Maya's functionality, including the UI, MEL is not the source of the commands per se. Therefore, the role of MEL in Maya could conceivably be filled by another language that can interact with Maya's command architecture via its own interpreter. Early in the development of Maya, this role was filled by the Tcl scripting language—before MEL came into being.

Introduction

If you're new to programming, some of the terminology in this section may sound foreign. Not to worry: terms like variables, function arguments, strings, and back quotes will be explained shortly.

Up to this point in this book, you've done most things in Maya using menus and tools available through the user interface (**UI**). In this chapter you will learn about Maya's powerful built-in scripting language called MEL (for Maya Embedded Language). With MEL you can perform just about any task in Maya using computer code (typed instructions) that you would otherwise do through the UI. Such tasks range from one-off actions such as creating a polygon or NURBS primitive using a MEL command, right up to procedural animations in which MEL code is used to animate attributes, producing complex results that would be difficult or impossible with keyframing alone. It's in this latter role that MEL scripting shines as powerful tool for in silico biology.

MEL scripting is not another layer of functionality built *on top* of Maya's core modeling, animation, and rendering systems. It is integral to the program's operation. When you create an object, set a keyframe or change the color of a shader through the UI, Maya executes a **MEL statement** (a command and text that modifies it) to do the job. In fact, the UI itself is constructed by MEL scripts each time you start Maya. A MEL script is a computer program composed of MEL commands and other supporting information. Figure 12.01 illustrates how MEL fits into Maya's program architecture.

The developers of Maya made its inner workings easy to see and interact with via MEL. Many of the statements you execute through UI tools are echoed (displayed as they are executed) in the Script Editor as MEL statements. As well, dependency and scene hierarchy relationships are plainly displayed in the Hypergraph. The ability to see what goes on behind the scenes when you select a menu item or use a tool is a tremendous advantage in learning how to script in Maya: you can see the statements Maya runs in response to actions you perform through the UI and then use those statements to build a script. You can subsequently enter the script to perform a number of tasks at once, saving the time you would otherwise spend working with menus and tools. If the tasks involve a repetitive action—such as building and distributing many similar objects throughout your scene—using a MEL script can be a significant time saver. You'll see examples of this shortly as you work through the chapter. The projects in *Part 03* of this book take advantage of MEL scripting to build complex models and to automate animation.

In this chapter you will learn how to work with MEL commands and scripts. Even if you are brand new to computer programming, set your worries aside. MEL is an elegant, powerful language and you'll quickly meet the basics in preparation for the projects in *Part 03*. By the end of this chapter you'll know how to create objects and animate attributes without relying on the common UI menus and tools.

The origins of MEL

Early in the development of Maya, prior to the Alias|Wavefront merger, it was deemed that the program's command architecture was to be integrated with the Dependency Graph **(DG)** through a scripting interface. This would allow users to execute commands and customize the Maya UI via typed instructions. The development team at Alias Research surveyed a number of scripting languages for the job, including PERL, Scheme, Tcl, and Python (which was still in early development). Tcl was used at first, in order to leverage its similarity to the Unix shell scripting language for the benefit of users; at the time, PowerAnimator (one of Maya's predecessors) ran on SGI computers, for which users had varying degrees of familiarity with the **Unix shell** scripting interface. In the 1995 merger under SGI, Wavefront Technologies brought with it Sophia, the scripting language embedded in Dynamation (see page 15 in *Chapter 01*).

In the Unix operating system (**OS**), the "shell" is a scripting interface allowing users to interact with the OS through typed instructions.

Originally written in 1990–1991 by Jim Hourihan—then of Santa Barbara Studios and, as of this writing, co-founder and R&D Director at Tweak Films studios—Sophia had many of the features the Maya developers were looking for in a scripting language: it was simple to use and fast to execute—a must for computer-generated imagery **(CGI)** artists; it was by design suited to 3D algebra and UI development; and it resembled Unix shell scripting. Sophia began its transformation into MEL at the hands of Alias|Wavefront programmer and IBM alumnus Joyce Janczyn. As it evolved, MEL took on certain traits of shell scripting languages, including dollar signs ($) before variable names and back quotes to call functions. Nonetheless, MEL retained many of the original Dynamation Sophia constructs like noise functions, vector algebra, and string manipulation.

Since the release of Maya 1 in 1998, there have been numerous additions to Maya's capabilities, but little has changed with MEL—save for the obvious addition of new commands. The syntax and data structures have remained steady in order to support customer workflows built on MEL. Remarkably, despite its supposed simplicity,

animators and **technical directors** continually use MEL to achieve sophisticated results in Maya that far exceed what MEL's developers originally had in mind for the language. "We intended MEL to enable users to script basic tasks and customize the UI in Maya" says Janczyn. "I've been surprised and delighted to see how users have embraced MEL and used it to do things in Maya we never imagined.[1]"

In a word: *Scripting*

"Scripting", when deployed as a put-down, is used to hint that a piece of software is facile or trivial compared to what the critic imagines it could have been, if only a "real" programming language had been used—one loaded with an ultimate range of data types—from booleans to objects, trees, and beyond—and packed with low-level operations to peek and poke the hardware. Languages with a trimmer range of data types, or with little to say about pokes or pointers, need not apply. Attitudes that champion this false dichotomy are semantic trash—dim echoes of the earliest debates over the merits of writing software in anything but machine language. As we saw in *Chapter 02*, these debates were in full swing long ago, in the 1950s following the appearance of the first popular languages with human-friendly design, like Fortran and COBOL.

You'll notice we're saying "MEL scripting" at this point, not "MEL programming". We've even used "MEL scripting" as our chapter's title! This deserves a short comment on terminology before we look closely at the nut-and-bolts of MEL's elegant syntax and command structure.

You may have encountered the verb "script" in other software books and come across it in on-line forums. Occasionally you might even have seen "**script**" and "scripting" used as put-downs among people with different attitudes about what makes a good programming language. In this book, we intend no such negative connotation. In fact you will see that we use the terms "scripting" and "programming" interchangeably—deliberately so, to make clear that scripting is a practice of software authoring capable of delivering sophisticated code for complex applications.

The modern notion of scripting is based on the insight that there are at least two interesting ways to get a computer to give you the outcomes you want. The first way is to write a program and let the computer run it. The second is to take control of the programs already running on the machine and somehow direct their operation to achieve the ends you seek. This insight is not new and dates from the early years of computer programming, when the first programming language compilers and language interpreters were developed. We noted in *Chapter 02* that interpreters for processing higher-level languages, such as IBM's Speedcoding, actually predated the invention of efficient compilers and the rise of Fortran, COBOL, and their peers in software history.

Chapter 02 also introduced the notion that an interpreter is a program simulating a "virtual computer", which runs as software inside the hardware of the actual physical machine. The interpreter's virtual computer is useful because it is not limited to the instruction set and operations wired into the hardware. It can go beyond them to include anything an ingenious programmer can devise. The MEL interpreter running inside Maya takes commands in the MEL language, which by comparison with modern-day assembly languages is very high level. With a single MEL command you can invoke entire math operations (like vector cross products) and computer graphics procedures.

In a program written in or compiled to machine code, you can with some accuracy say your software is running the hardware, since the hardware circuits are wired to react automatically to each machine language instruction in just the right way. When you are inputting code to an interpreter, however, you are not intervening on the machine hardware in quite the same way. Your software file—your MEL script—is directing the activity of the interpreter to produce the results we need.

With humble origins in the early interpreters and the job control commands of the early operating systems, "scripting" has come to mean writing software that organizes or modifies the activity of other, pre-existing computer programs. A "script", then, is a program that modifies or guides the activity of another program or programs.

That is why script programs are sometimes called "glue" programs: they bring together existing processes. "Scripting" is the creative act of programming in the language understood by an interpreter that runs as part of the target program. These programming languages are usually referred to as scripting languages.

One interesting trait of modern scripting languages is concision: it is not unusual to see a script that is just a fraction of the length (and is written in a fraction of the time) of a file for the same job written in a general-purpose language like C++. Scripting languages achieve their concision by drawing on the pre-existing tools and resources of the application program they direct. Thus while the creation of a specific 3D pattern of noisy driving forces for an animation might take many lines of mathematics and flow control expressions in C or C++, a single MEL command call to the Maya physics engine invokes an entire toolset of such procedures.

The quest for concision is of course not the private territory of scripting codes. This is an exciting era in which programming language designers of all stripes are finding ways to let you write correct software, in the fewest possible lines of crystal-clear code, making the fewest mistakes along the way.

Trends to concision are nothing new to the evolution of programming languages. We saw in *Chapter 02* that while the earliest assemblers did little more that transcribe abstruse binary instructions on a 1-for-1 basis, they were quickly enhanced by assembler macro commands, which unrolled into whole sets of machine instructions or triggered an entire subroutine to automatically load from a library. Compilers took this a step further with parsers that unpacked scientist-friendly expressions like `y(k+j) = log(x(k)) + sin(x(k)*x(j))` into the lengthy list of assembly instructions needed to drive the math through the processor circuitry. Scripting languages, in their turn, transform objects, methods, and procedures of their host application into building blocks for task automation.

The scripting language programmer is also the happy beneficiary of everything learned so far from the mistakes and oversights of the earlier language designers, on whose shoulders we stand. Since scripting languages are in vogue, you have many flavors of command syntax, data type, and program flow control to explore as your skills and application needs develop—sometimes even for the same host application program! For example, as this book goes to press Autodesk has announced that, starting with Version 8.5, Maya can be scripted not only with MEL but also with Python, a general-purpose, object-oriented language (http://www.python.org) with legions of admirers. Since MEL remains the language of the Maya UI and the bedrock of Maya procedural animation we shall have just a little to say about Python in this book.

The diversity of software equipped with scripting languages and interpreters today is truly impressive: web browsers, image and video editing and compositing software, computer games, spreadsheets, operating systems (the "shell scripting" patois of Unix has been hugely influential), and of course 3D computer modeling and animation packages like Maya with its MEL language and MEL interpreter.

MEL is a beautiful instance of a **domain-oriented programming language**. Since it leverages on the high-level objects of its host application, Maya, it has a special aptitude for certain kinds of problems. Thus in addition to its general-purpose facilities for arithmetic, string manipulation, and file input/output, MEL has a huge vocabulary of commands for 3D computer graphics and animation operations. It is designed expressly for solving problems in the domain of 3D animation.

A language like C++, by comparison to a domain-oriented language like MEL, certainly will provide you general-purpose facilities—very powerful general-purpose facilities supporting diverse data constructs and multiple styles of programming. The domain orientation you build yourself with the aid of whatever libraries you find helpful. A domain-general language doesn't nudge you in one application direction or another. You could sit down with C++ to write a tax accounting program as readily as you would write to a game engine or a 3D modeling application. But it's your responsibility to build the domain-relevant tools and capabilities for your application. The general-purpose language definition does not supply them from the get-go.

MEL also is (for now at least) tied to the application Maya; it is a case par excellence of an **application-oriented programming language**: a scripting language intended to be used in association with particular application software. For the time being, there is no sign that other top-tier software packages for 3D modeling and animation will swing to MEL as a de facto scripting standard. They have their own scripting language tools. MEL is likely to remain at home in its established Maya environment. Fortunately most programming languages—including those invented for scripting—have strong family resemblances. That is because all, ultimately, are about letting you micro-manage the activity of von Neumann-style computer hardware (see page 32 in *Chapter 02*). The benefit of this common foundation is that experience you get from MEL and Maya will accelerate your progress with other learning curves.

With this comparative vocabulary in place we can begin to explore MEL in depth, recognizing it as a modern, domain-oriented, application-specific programming language for scripting the Maya animation package. Let's get started!

Getting started

Prepare your scene file

As you learn MEL, you'll be typing statements and sending them to Maya for processing. For this you'll want to have Maya running.

1. Start Maya.
2. Choose Window → Settings/Preferences → Preferences.
3. Choose Categories → Settings and make the following settings:
   Under Working Units → Linear: **centimeter**.
   → Angular: **degrees**.
   → Time: **NTSC**.
4. Choose Categories → Timeline and make the following settings:
   Under Timeline → Playback Start: **1**.
   → Playback End: **300**.
   → Time, select **NTSC**.
   Under Playback → Looping: **once**.
   → Playback Speed: **Play every frame**.
   → Playback by **1**.
5. Press the Save button to set your preferences.
6. Select the Perspective view of your scene by pressing the button in the Toolbox.

MEL input

There are three primary ways to enter MEL statements and scripts in Maya:

1. Through the **Command Line**.
2. Through the **Script Editor**.
3. By **sourcing** a MEL script.

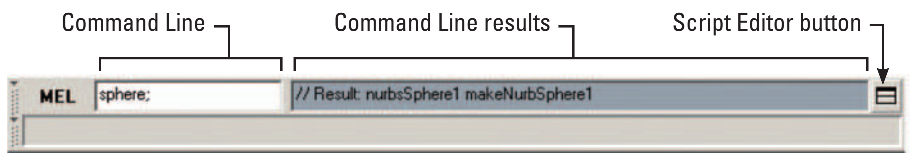


FIGURE 12.02

The Command Line is useful for entering single MEL commands. It also displays the result of the most recent scripting action—in this example, the creation of a NURBS sphere object. The button at the far right of the Command Line launches the Script Editor.

The Command Line

For entering single MEL statements, the Command Line (Figure 12.02) comes in handy. A statement is entered by typing it in the input field and then pressing Enter on your numeric keypad.

The Script Editor

The Script Editor (Figure 12.03) is used both for single statements and multiple lines of MEL script. Launch the Script Editor in one of two ways:

1. **Choose Window → General Editors → Script Editor.**

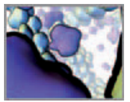
or

2. **Press the ⊟ icon at the far right of the Command Line.**

Because the Script Editor behaves in some ways like a text editing application, you can use it to compose MEL scripts. However, we recommend writing your MEL scripts in a text editor external to Maya and saving them as plain text files separate from your Maya scene file. On page 302 we've listed several text editors that work well for composing computer code.

We use the Script Editor for running and **tracing** (reporting the goings-on of a script at run-time) externally written MEL scripts and for testing short bits of MEL code that don't necessarily warrant creating an external text file. Tracing refers to the process of displaying certain script results in the history panel of the Script Editor to help you **debug** (locate and fix errors within) your code.

When composing MEL statements and scripts in an external text editor, you can get the code into Maya by copying and pasting it into the command input panel of the Script Editor. After the code has been pasted, pressing Enter *runs* (or executes) it in Maya. The two Enter keys on your keyboard perform different functions in the Script Editor. The alphanumeric Enter key (located next to the letter and punctuation keys) causes a line break in the command input panel; it doesn't send any code to Maya. The numeric keypad Enter key executes the code in the command input panel. From here on, when you see an instruction to *enter* a command or script in the Script Editor, it means type or paste the code into the input panel and press the numeric keypad Enter key. When you select code within the input panel and press Enter, only the selected text is sent to Maya for processing. When Maya is finished, the selected text remains in the input panel. You can delete text in the input panel by selecting it and pressing Delete on your keyboard.



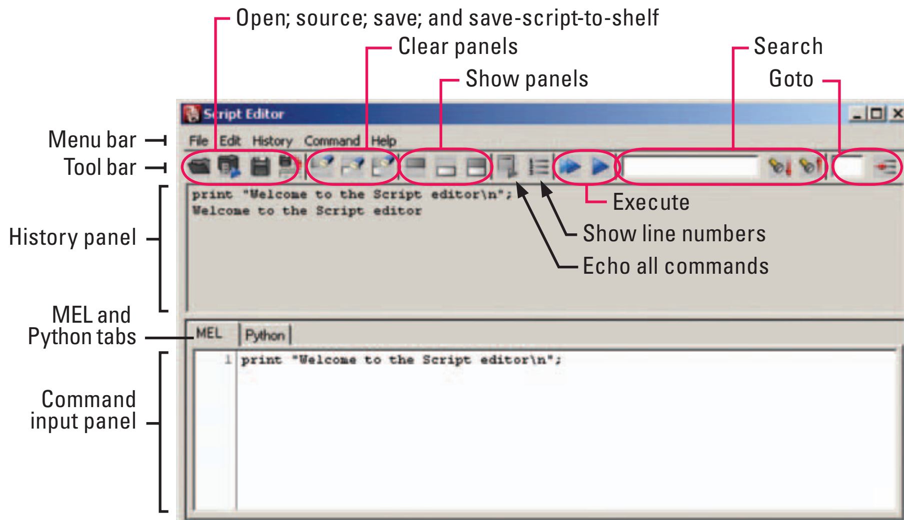


FIGURE 12.03

The Script Editor is an essential tool for testing, sourcing, and debugging MEL commands and scripts. Code that is selected (highlighted in blue) remains in the command input panel after you hit Enter to run it.

The History panel in the Script Editor automatically logs MEL command results. You can clear the History or command input panel at any time as follows:

From the Script Editor menu bar, choose Edit → Clear History
or → **Clear Input**

We will explore other features of the Script Editor as they pertain to specific examples in the chapter. The Maya Help Library is a good one-stop source of additional information as you work along:

The Script Editor

Maya Help → Using Maya → General → Basics → Basic Windows and Editors → Script Editor

Sourcing an external MEL script

Depending on the settings/preferences in your text editor, a long line of text may reflow to the following line when it exceeds the page margins. Maya ignores these "line feeds". Carriage returns or "line breaks" (pressing Enter), however, are not ignored; each line ending in a carriage return must be terminated with a semi-colon before Maya will accept it.

When you **source** a script, you are loading it from a text file that is external to your Maya scene. You can source a script through the Script Editor as follows:

1. **Choose Window → General Editors → Script Editor.**
2. **From the Script Editor menu bar, choose File → Source Script.**
3. **In the Source Script window, navigate to your script file, select it, and press Open.**

   This automatically loads the script into memory. If it contains errors, Maya will display error messages in the History panel of the Script Editor and in the Results field next to the Command Line.

You can also source a script using the MEL command, source. For example, you might enter the following line of code in the Command Line to load a MEL script you'd created and saved in a file called myFirstScript.mel.

```
source myFirstScript.mel
```

However, for this to work, your script file must be located on Maya's **search path** and be listed in the **search path contents**.

Maya's search path

When you installed Maya on your computer, a default **Scripts directory** was created for your user account. The path to this directory is Maya's search path. Every time you start Maya, the program queries the contents (file names) of its search path. When you use the source command, Maya scans its search path contents for the file. If the file exists on the search path, Maya sources it (loads it into memory). If you add files to the Scripts directory when Maya is running, you will have to refresh the search path contents using the rehash command in order to have access to those files and their contents.

To refresh the search path contents, enter the following in the Script Editor:

```
rehash;
```

You can get the search path by querying Maya's internal variables as follows:

Enter the following in the Script Editor:

```
internalVar -userScriptDir;
```

From this point on, when you see an instruction to ***enter*** a command or script in the Script editor, it means type or paste the code into the input panel and press the numeric keypad *Enter* key.

The result displayed in the Script Editor should look something like:

```
// Result: C:/Documents and Settings/User/My Documents/maya/8.5/
  scripts/ //
```

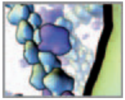
We'll return to sourcing scripts toward the end of the chapter. For now, you will run MEL commands and short scripts by entering them directly in the command input panel of the Script Editor.

MEL syntax

Before dealing with MEL commands explicitly, let's look at some of the elements that are fundamental to writing MEL scripts in Maya.

Terminate your MEL statements;

For our purposes in this chapter, a MEL statement is a single instruction to be executed by Maya. This may be a MEL command to return the value of an attribute or a declaration of a variable used to store a number. Regardless of its purpose, each MEL statement must be terminated with a semi-colon. This tells Maya where one instruction ends and the next one begins. Strictly speaking, when only one MEL statement is executed on its own, it doesn't require a semi-colon. However, when more that one statement is passed to Maya at a time, it is essential to end each with a semi-colon. There is one notable exception: between the "if" and "else" components in a **conditional statement**—which we'll discuss later in the chapter.



Not all "quotation marks" are "the same"

There will be many occasions where your MEL statements use double quotation marks (") to enclose character **strings**. As in other computer programming languages, Maya will accept only *straight* (" ") and not curved (" ") or slanted quotation marks (or quotes). The latter will generate an error in Maya. When you type code in Maya's Script Editor, the double quotes are straight. However, when you compose a MEL script in an external text editing application, you may have to set the application to use straight quotes instead of curved quotes (which are sometimes called *smart quotes*); consult the documentation of your text editing program for instructions on how to specify the type of quotation marks used.

MEL is CASE sensitive

To Maya, the following two statements are distinct from one another

1. `ScriptEditor;`
2. `scriptEditor;`

#1 will launch Maya's Script Editor whereas #2 will generate an error message because there is no such command as `scriptEditor`. To your computer, upper and lower case versions of the same letter are distinct characters, and Maya treats them as such.

Comments

Comments are statements used to document your code. They are ignored by Maya during the execution of a script. Commenting your code is an important part of scripting because it communicates information about variables and other elements of your script in plain language. Such communication is helpful not just for other users who are trying to understand your code but also for keeping track yourself of what does what. Single line comments are denoted with two forward slashes, //, and ended with a line break (by pressing the Return key), as in the following example:

```
print "This line will print in the Script editor.";
//print "This line will NOT print in the Script editor."
```
(press Enter here)

The MEL command **print** has nothing to do with creating a paper ("hard copy") output. It instructs Maya to display a value on your computer screen in the Script editor and the Command Line. In the next chapter we will discuss the important topic of moving data between your Maya models and your computer's file system (data input/output).

Multi-line, or **block comments**, are denoted with a forward slash and asterisk, as follows:

```
/* Commenting your code is an important part of scripting because
  it communicates information about variables and other elements of
  your script in plain language. */
```

Values

A value is what is stored in an attribute or a variable. The number 11 and the word "eleven" are values. Much of what a MEL script does is manipulate values. Every value in Maya has a specific type. For example, the transform attributes (translate, rotate, and scale) of a typical transform node all use **floating point**, or *decimal* values. The different value (or data) types in Maya are as follows:

Data type	Example
integer	5
floating point (float)	5.25
string	"Henry, Alex, Aaron", "3.14"
boolean	"yes", "no", 1, 0

Within a MEL script, values are often stored in variables, which we'll discuss next.

Variables

A variable is a container for data in a computer program. After being created—or "declared"—a variable exists for the duration of the program although its value typically "varies", that is takes on different values as the program runs. For example, you may create a variable to store a chemical concentration, a force, or the speed of a crawling cell. As the program executes, changes in the concentration, force, or speed are reflected in the changing value of the variable. In MEL, variables store one of the four types of data listed above. In addition to single-value variables, MEL supports compound variables in the form of vectors, arrays, and matrices. Table 12.01 lists the different variable types available in MEL and examples of their use.

The variable types listed in Table 12.01 also apply to node attributes. That is to say, every attribute has a particular data type. In fact, variables and attributes are very similar, in that they both store values and the same types of data structure (vectors, arrays, and matrices). The difference is that attributes belong to nodes in the Hypergraph whereas variables are not connected in any permanent way to the scene—each time you start Maya, a variable must be **declared** and assigned a value before being used. In contrast, an attribute and its value is saved within your Maya scene file.

Naming variables

Every variable name must begin with the dollar sign ($) symbol, followed immediately by a letter or an underscore (_), but not a number. The remaining characters can be any combination of numbers, letters, and underscores and can be as long or as short as you like. The following are examples of variable name "dos" and "don'ts":

`$position`	**Correct**
`$position5`	**Correct**
`$position_5`	**Correct**
`$5position`	**Incorrect** (a number cannot immediately follow $)
`$_5position`	**Correct**

If you leave the $ off the front a variable name—"position", for instance—Maya will interpret the word as a procedure. If it can't locate a procedure named "position" it will generate an error message. In addition to proper syntax, it is good practice to give your variables intuitive names that relate to their functions within a script. This will not only benefit others who use your scripts, but will help you recall how your code works when you haven't seen it for a while.

Declaring and assigning variables

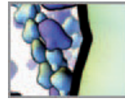
A variable must be *declared* before being used. The following statement declares `$title` as a variable of type, string.

```
string $title;
```

Assigning a variable means storing a value in it. Continuing on, the following statement assigns a value to `$title`:

```
$title = "cellInvasions";
```

Variable declarations for the different data types, along with sample assignments, are shown in Table 12.01. Note that an array is not declared as "array" but rather according to the type of values it stores.



Variable type	Declaration	Assignment
int	int $myInt;	$myInt = 20;
float	float $myFlt;	$myFlt = 3.14159265;
vector	vector $myVct;	$myVct = << 5.5, 1.1, 6.6 >>;
string	string $myStr;	$myStr = "henry";
vector array	vector $myVctArray[];	$myVctArray = {<< 1, 0, 0 >>, << 0, 1, 0 >>, << 0, 0, 1 >>}; *or* vector $myVectArray[0] = << 1, 0, 0 >>; vector $myVectArray[1] = << 0, 1, 0 >>; vector $myVectArray[2] = << 0, 0, 1 >>;
float array	float $myFltArray[];	$myFltArray = { 4.5, 12, 6.2 }; *or* $myFltArray[0] = 4.5; $myFltArray[1] = 12; $myFltArray[2] = 6.2;

matrix	matrix $myMatr[2][2];	matrix $myMatr2[2][3]
		= << 4.5, 12, 6.2; 5.4, 21, 2.6 >>;
		or
		float $myMatr[0][0] = 4.5;
		float $myMatr[0][1] = 12;
		float $myMatr[0][2] = 6.2;
		float $myMatr[1][0] = 5.4;
		float $myMatr[1][1] = 2.1;
		float $myMatr[1][2] = 2.6;

TABLE 12.01

Variable types. An array can be of type integer, float, vector, or string. All matrices are of type float. The size of a matrix must be declared explicitly and, unlike an array, the size of a matrix cannot change once it's been declared.

If you use a variable before declaring it, the MEL interpreter will generate an error. The following code attempts to perform an operation with the variable $edition which hasn't yet been declared.

In addition to alphabetic characters, a string variable can contain numbers.

```
string $title;
string $filmName;
$title = "cellInvasions";
$filmName = $title + $edition;
// Error: $filmName = $title + $edition; //
// Error: "$edition" is an undeclared variable. //
```

To conserve space in your scripts, you can declare multiple variables of the same type together on the same line, as follows:

```
int $counter1, $counter2, $counter3, $counter4;
float $cellX, $cellY, $cellZ;
```

Dynamic typing

MEL is a **dynamically typed** language, meaning you can assign a value to a variable without explicitly declaring its data type. For example, the following variable assignment:

```
$myVar = "cellInvasions";
```

is interpreted the same as the explicitly typed statement:

```
string $myVar = "cellInvasions";
```

MEL's author and long-time Maya developer Joyce Janczyn incorporated dynamic typing for the sake of 3D artists who, early in the development of Maya, rallied against excessive formality in its scripting language. Although dynamic typing demands less attention to detail than does requiring users to explicitly type every variable, Janczyn cautions that "explicit typing helps you stay organized and get better diagnostics when there are bugs in your script.[1]" When scanning your MEL code, you can tell at a glance the data type of each variable.

So, you can assign a value to a variable either when you declare it or afterward, as in the following example:

```
string $myStr;
$myStr = "cellInvasions";
```

or

```
string $myStr = "cellInvasions";
```

The choice is one of organizational style; for the MEL scripts in this book we separate variable declaration and assignment—as in the first example above—because we find the code clearer to follow this way.

Data conversion

If you attempt to assign a value of one type to a variable of another type, Maya will convert the value in order to complete the assignment. When assigning a string value to an integer or float variable, Maya behaves as shown in the following examples:

```
string $str = "Hello world.";
int $int = $str;
// Warning: line 3: Converting string "Hello" to an int value of 0. //
// Result: 0 //

float $flt = $str;
// Warning: line 1: Converting string "Hello" to a float value
  of 0. //
// Result: 0 //

$str = "3.1459";
$int = $str;
// Warning: line 2: Converting string "3.1459" to an int value
  of 3. //
// Result: 3 //

$flt = $str;
// Result: 3.1459 //
```

Maya applies different rules depending on what type of data mismatch you give it. The Maya Help Library contains information on data type conversion:

Data type conversion in Maya

Maya Help → Using Maya → General → MEL and Expressions → Advanced → Advanced programming topics → Automatic type conversion

Maya Help → Using Maya → General → MEL and Expressions → Advanced → Advanced animation expressions topics → Data type conversions

Type casting

In certain instances it's desirable to cast data from one type to another. In the following example, dividing one integer by another returns a somewhat unexpected result:

```
int $a = 1;
float $b = $a/5;
// Result: 0 //
```

While $b was declared a float, dividing an integer by an integer returns an integer. To get the desired result, cast the value of $a to type float as follows:

```
int $a = 1;
float $b = (float) $a/5;
// Result: 0.2 //
```

You will use this technique in the next chapter to convert data before assigning it to variables.

Strings

Certain characters perform special functions when used with character strings in Maya (Table 12.02). They apply to string variables and to the MEL commands, `print` and `expression`.

Vectors

Vectors are triple floating point variables or attributes, and are used often in Maya to describe 3D transform values and RGB colors. Vectors are denoted by angled brackets and must be assigned all three elements at once, as follows:

```
$myVect = << 2.3, 4.6, 7.5 >>;
```

Although they must all be *assigned* at once, individual elements can be *queried* one at a time using the .x, .y, and .z **accessors**, as in the example below:

```
float $x, $y, $z;

$x = $myVect.x; // Result: 2.3 //
$y = $myVect.y; // Result: 4.6 //
$z = $myVect.z; // Result: 7.5 //
```

Character	Description	Sample use
\"	**Quotation mark character**	`$myStr = "She said \"eureka!\"";` `// Result: She said "eureka!" //`
\t	**Tab**	`$myStr = "She\tsaid\t\"eureka!\"";` `// Result: She  said  "eureka!" //`

\n	**New line**	`$myStr = "She said:\neureka!";`
		`// Result: She said:`
		`eureka! //`

\r	**Carriage return**	`$myStr = "She said:\reureka!";`
		`// Result: She said:`
		`eureka! //`

\\	**Back slash character**	`$myVar = "Escape characters with a back slash: \\";` `// Result: Escape characters with a back slash: \ //`

TABLE 12.02

Special characters are used to modify strings in Maya. The backslash character “escapes” the following character so that it will be included in the string and not treated as computer code. The Script Editor treats line breaks (\n) and carriage returns (\r) the same.

Arrays

An array is a list of integers, floats, strings, or vectors. Square brackets following a variable name mark it as an array. Arrays are declared according to the types of values they contain.

```
float $myFirstArray[ ]; // Size = 0
float $mySecondArray[ ] = {5.1, 6.2, 7.3, 8.4}; // Declaring size
  is optional.
string $myThirdArray[ ] = {"cell1", "cell2", "cell3"};
float $myFourthArray[4]; // Size = 4. // Allocate memory for
  4 values.
int $myFifthArray[4] = {5, 9, 2, 12}; // The size (4) is redundant
  here.
```

The size of an array is the number of elements it contains. You can set the size explicitly when you declare an array. Alternately, you can leave it blank (a “zero-element” array) and let Maya increase it automatically as you add elements. The first three arrays in the example above are declared as zero-element arrays. The fourth and fifth arrays are declared each with a size of 4.

The term **index** is used to refer to a specific element of an array. In Maya, indices start at zero. Therefore a four-element array has indices numbering from 0 to 3. This is important to remember when you begin using arrays in your MEL scripts since it’s easy to confuse the first element of an array with index #1, when in fact the first element corresponds to index #0.

```
float $myVar = $mySecondArray[1]; // Result: 6.2 //
```

You can assign array elements together, using curly brackets, or individually using index numbers within the square brackets as follows:

```
float $myFirstArray[] = {5.1, 6.2, 7.3, 8.4};
```

or

```
float $myFirstArray[];
$myFirstArray[0] = 5.1;
$myFirstArray[1] = 6.2; // etc.
```

Arrays of vectors come in handy for storing position data in the migrating cell simulations you'll undertake in *Part 03* of this book. A vector array is declared and assigned as follows:

```
vector $myVectArray[] = { << 5.1, 6.2, 7.3 >>, << 1.5, 2.6, 3.7 >> };
```

or

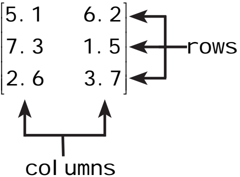
```
vector $myVectArray[];
$myVectArray[0] = << 5.1, 6.2, 7.3 >>;
$myVectArray[1] = << 1.5, 2.6, 3.7 >>;
```

Matrices

A matrix is a 2D array of floating point values. The size of a matrix variable—the number of rows and columns—must be stated explicitly when it is declared. Like 1D arrays, matrices use square brackets. Furthermore, once you've declared a matrix, as in the following example, its size cannot be changed.

```
matrix $myMatrix[3][2] = << 5.1, 6.2; 7.3, 1.5; 2.6, 3.7 >>;
                  /   \
              rows     columns
```

The first square-bracketed index specifies the number of rows and the second to columns. In conventional mathematical notation, the matrix above would be written as follows:



To query or set a specific element in an array, use its row and column numbers, as follows:

```
float $tmpFlt = $myMatrix[1][0];
// Result: 7.3 //

$myMatrix[1][0] = 75;
// Result: 75 //
```

Global variables

A variable is either local or global, reflecting its **scope** in Maya. A local variable operates only within the animation expression or procedure in which it's declared. In contrast, a global variable can be declared and assigned in one procedure, and then queried and reassigned in any other procedure or expression available to your Maya

scene. The significance of a variable's scope will become apparent as we explore the scripting structures, procedures and expressions, later in this chapter.

All variables declared in the Script Editor—that is, entered via the Command input panel—are automatically global variables. However, this does not apply to variables contained within procedures or animation expressions that are in turn entered via the Script Editor. In procedures and animation expressions, variables can be either global or local. A variable is local if declared as follows:

```
string $myString;
```

A variable is declared global by preceding it with the word "global":

```
global string $yourString;
```

A global variable need only be assigned once, but it must be declared within each procedure or expression that uses it.

You cannot re-type a global variable

Once you declare a global variable, you cannot change its data type until you restart Maya. For example, enter the following code in the Script Editor:

```
string $myStr = "cellInvasions";
$myStr = "tumor progression"; // Okay.
int $myStr = 5; // Declare $myStr as a different type.

// Error: int $myStr = 5; //
// Error: Invalid redeclaration of variable "$myStr" as a different
  type. //
```

You made `$myStr` *global* by entering it in the Script Editor. The code then attempts to retype it as an integer. You cannot use the name `$myStr` with a different variable type until you quit and restart Maya.

Global variables

Maya Help → Using Maya → General → MEL and Expressions → Debugging, optimizing, and troubleshooting → Troubleshooting → Accessing global variables

Values and variables in Maya

Maya Help → Using Maya → General → MEL and Expressions → Values and variables

Arrays, vectors, and matrices

Maya Help → Using Maya → General → MEL and Expressions → Arrays, vectors, and matrices

Mathematical and logical expressions

The word **expression** refers to two things in Maya. The first is a mathematical or logical statement composed of one or more **operands** (or values) and one or more **operators**. For example:

```
5 + 6
```

or

```
size($someArrayVariable)
```

Expressions may contain MEL commands, as in the second example above (size is a MEL command), and MEL command statements may contain expressions as follows:

```
float $var1 = 5.5;
setAttr mySphere.translateX ($var1 + 6);
```

setAttr is a MEL command used to set the value of an attribute.

Blocks

Local variables declared within a block operate only inside that block. In other words, the variable's scope is limited to the block. For example, moving the print $message command outside the code block to the right generates an error message.

A block is several expressions grouped together in curly brackets. Blocks are used frequently in conditional statements—a programming structure you'll meet more formally shortly. For example:

```
float $x, $y;
$x = 5;
$y = 4;
if ($x > $y)
{
   float $biggest = $x;
   string $message = "\n$x is bigger than $y\n";
   print $message;
}
```

For legibility, blocks are usually indented (as shown above). Also, each statement within a block must end with a semi-colon (an uncommon requirement in many programming languages). However, the end of a block—the second curly bracket—does not require a semi-colon.

The second type of expression in Maya is an **animation expression**, which is a statement or script that you typically attach to an attribute to animate it. We will explore animation expressions in detail beginning on page 292 .

Operators

Operators are used in arithmetic and logical expressions and in conditional statements. Table 12.03 lists the operators available for use in MEL. The order in which they are evaluated in an expression—their order of precedence—is as follows:

Highest	() [ ]
	! ++ --
	* / % ^
	+ -
	< <= > >=
	&&
	||
	? :
Lowest	= += -= *= /=

Operators on the same row of the above list have equal precedence. When operators from the same row are used together, the left-most one in the expression is evaluated first. The following two expressions demonstrate how operator precedence causes two

Operator	Name or meaning	Sample use
Arithmetic operators		
()	Round brackets	$var1 = 2 * (2 + 3); // Result: 10.
[]	Square brackets	$var1 = $array1[4]; // 5th element of $array1.
!	Not	if ($var != 0) // If $var not equal to 0, follow with a function or a command.
++, --	Increment, Decrement by 1	$var1 ++; // Increases the value of $var by 1.
*	Multiply	$var1 = 2 * 5; // Result: 10
/	Divide	$var2 = (float) 2/5; // Result: 0.4 (float) forces 2/5 to return a floating point value.
%	Modulo	$mod = 8% 3; // Result: 2
^	Vector cross product	$vect = <<1, 3, 5>> ^ <<-2, 4, -5>>; // Result: <<-35, -5, 10>>
+, -	Plus, minus	$var2 = $var2 + 5 – $var1; // Result: -4.6
Logical operators		
<	Less than	if ($var1 < 5) …
<=	Less than or equal to	if ($var1 <= 5) …
>	Greater than	if ($var1 > 5) …
>=	Greater than or equal to	if ($var1 >= 5) …
==	Equal to	if ($var1 == 5) …
!=	Not equal to	if ($var1 != 5) …
&&	Logical AND	if ($var1 == 5 && $var2 == 6) …
||	Logical OR	if ($var1 == 5 || $var2 == 6) …
?:	If-else shorthand	($var2 < 0) ? $var2 : $var2/2; // If $var2 < 0, return its value, else return its value divided by 2.

TABLE 12.03

MEL operators listed in order of precedence from the highest at the top of the table to the lowest at the bottom.

In some programming languages the circumflex (**^**) character is used for the exponentiation operation. Still other languages use a double asterisk (**) for the same operation. In Maya, this operation is performed with the **pow(a,n)** command, where **a** is the base and **n**, the exponent.

similar expressions to evaluate differently due to the order in which the operators are applied.

```
float $myFloat = 3 + 6 * 2;
// Result: 15 //
```

```
float $myFloat = 2 * 3 + 6;
// Result: 12 //
```

Expressions and operators

Maya Help → Using Maya → General → MEL and Expressions → Syntax → Expressions, operators and statements

Operator overloading

In a programming language with operator overloading, certain operator symbols can have more than one meaning. A familiar example is the use of the symbol +. It would not be unusual to write code today in which the "plus" sign could be invoked legitimately to "add" together pairs of data of diverse kinds: pairs of integers, or pairs of decimal numbers, or two strings or vectors or matrices—all with one little "plus" sign. (To get a feel for the old days, just try this in a vintage language of the 1950s or 1960s, like Fortran!)

Operator overloading can bring significant concision to your coding. Why use a plethora of symbols or procedure invocations when the right one can be triggered in context with just one symbol? Be careful though: a downside to overloading, and it can become a significant issue as the size and complexity of your program grows, is the problem future readers of your code (including yourself) may have deciphering what you intend the overloaded operator to do at each place it is used in your code. For example, seeing what you mean by two integers added may be clear, but what might a reader make of a line in which your plus sign links an integer and a decimal number, or a decimal number and a string? Error or genius? Good coding style in an overloaded programming language demands very careful attention to how the compiler or interpreter handles all the possible permutations of data types you could throw at it.

The MEL command

Let's look at a MEL command that runs when you choose a typical menu item in Maya. You'll then use that command to perform the same action without making the menu selection.

1. **Type the following in the Command Line and press Enter:**

   ```
   ScriptEditor;
   ```

You've just run a MEL command!

2. **Adjust the Script Editor so that you can see both the history and command input panels. You can change the relative size of the panels by LMB+dragging the horizontal bar dividing them.**

3. **From the Script Editor menu bar, choose Edit → Clear All.**

4. **From the main window menu bar choose Create → Polygon Primitives → Sphere ❐.**

5. **In the Polygon Sphere Options window choose Edit → Reset Settings.**

6. **Press Create.**

As you may have expected, your actions above created a sphere centered at the world origin of your scene. Now let's look at the two lines that appeared in the Script editor History panel immediately after you pressed Create.

```
polySphere -r 1 -sx 20 -sy 20 -ax 0 1 0 -tx 1 -ch 1;
// Result: pSphere1 polySphere1 //
```

The first line is a statement containing the MEL command and various modifiers. The second line states the **return value** of the command: the names of the transform node (pSphere1) and the creation (or history) node (polySphere1). By default, the shape node name is not reported. The forward slash "//" characters indicate non-executable (commented) code—that is, information intended only for the user and not for Maya to process in any way. Let's take a closer look at the first line:

```
polySphere  -r  1 -sx  20 -sy  20 -ax  0 1  0 -tx  1  -ch  1;
```

MEL command flag flag argument

The MEL command performs a task—manipulates the UI or the scene graph in some way. In the above example, the polySphere command creates a polygonal sphere. A flag is preceded with a hyphen (or dash) and modifies the MEL command according to the flag arguments given to it. The arguments are values of a certain type: float, integer, string, and so on. Flags often correspond to attribute settings. The first flag, -r, specifies the sphere's radius: the default argument value for this flag is 1. When Maya traces a command to the Script Editor, it displays the short names for the flags: "-r" is short for "-radius". You can use long or short names when you type MEL commands.

Next you'll create another sphere by reusing the statement that Maya traced in the History panel of the Script Editor.

1. **Select the following line in the history panel of the Script Editor:**

   ```
   polySphere -r 1 -sx 20 -sy 20 -ax 0 1 0 -tx 1 -ch 1;
   ```

2. **LMB+drag the selection from the history panel to the command input panel.**
3. **With the text still selected (highlighted), hit Enter on your numeric keypad.**

This should create a second sphere, automatically named pSphere2.

Flags and default tool settings

Specifying flags and arguments is essential if you want a command to do something other than what its default settings dictate. By extension, you need only to include flags whose values you wish to alter from the default settings; those that aren't specified in your MEL command line will be used at their default values. For example, let's create a third polygon sphere with the same dimensions and other basic settings, but with a unique name:

1. **Type the following line into the Script Editor:**

   ```
   polySphere -name mySphere;
   ```

2. **Press Enter.**

If you take a minute to inspect the sphere in your perspective scene view and look at its nodes and attribute settings in the Attribute Editor, you'll see that it is the same as the previous two spheres except for its name, which is "mySphere" instead of

pSphere3. The other flags—radius, subdivisionsX, subdivisionsY, and so on—were taken at their saved settings since you didn't specify alternate values.

MEL command mode: Create, edit, and query

In the examples above, you used the polySphere command in its default **create mode**. Here you'll explore the other two MEL command modes: **edit** mode **query**. Edit mode makes changes to the attributes of an existing item. For example, change the radius of mySphere as follows:

1. **From the Script Editor menu bar, choose Edit → Clear All.**
2. **Type the following code in the input panel and press Enter:**

```
polySphere -edit -radius 5 mySphere;
```

The -edit flag specifies the command mode and requires no arguments. The -radius flag sets the sphere's radius to the flag argument value of 5. The final term in the statement, mySphere, is a **command argument**; it has no flag and tells the polySphere command what is being edited.

Query mode is used to query information about a node whose name is specified by the command argument. For example, suppose you wanted to know if mySphere has construction history:

1. **From the Script Editor menu bar, choose Edit → Clear All.**
2. **Type the following code in the input panel and press Enter:**

```
polySphere -query -ch mySphere;
```

If you highlight text (by selecting it) before entering it in the Script editor, it will remain in the command input panel, ready to be run again. On the other hand, if you enter a statement without selecting it first, it will run, but the text will be deleted from the command input line.

The flag, -ch, is short for the rather cumbersome word: -constructionHistory. After entering the command, you should see the following result in the History panel of the Script Editor:

```
// Result: 1 //
```

The return value in this case is **boolean**: 1 corresponds to "yes" and 0 to "no". Therefore the sphere does indeed have construction history. Naturally, a quick way to query an attribute value is to select the item in question and inspect the Channel Box or the Attribute Editor. However, when you begin using MEL scripts to build complex models and run simulations, you'll need to query attribute values for multiple objects in rapid succession—the position and orientation of colliding molecules for example. You'll learn additional methods for querying and setting attributes shortly.

MEL command syntax

Imperative and function syntax

The MEL command example above (polySphere -query -ch mySphere;) is written in **imperative syntax** which is used in Unix shell and DOS commands. Maya also supports

function syntax, in which commands resemble a standard function call in computer language. The following statements do exactly the same thing when executed in Maya:

1. **Imperative syntax:** `polySphere -name mySphere;`

2. **Function syntax:** `polySphere ("-name", "mySphere");`

In #2 the flag and flag arguments are passed as *function arguments* to the command polySphere. Note that the arguments must be enclosed in quotation marks, unlike with imperative syntax, for which quotation marks around character strings (e.g. "mySphere") are optional.

As you work with MEL, occasions will arise in which you want to pass the return value of a MEL command to an attribute or to a variable. In the following example, $rad is an empty variable used to store the radius attribute value for a polygon sphere named `mySphere`.

1. **Imperative syntax:**

```
float $rad = `polySphere -query -radius mySphere`;
// Result: 5 //
```

2. **Function syntax:**

```
float $rad = polySphere ("-query", "-radius",
  "mySphere");
// Result: 5 //
```

When imperative syntax is used, the command has no return value unless you force one by surrounding the command in back quotes as shown in #1 above. Function syntax returns a value without the need for modifying characters. With few exceptions, we use imperative syntax throughout this book. It generally requires fewer characters than function syntax and is therefore easier to **debug** (analyze and correct for errors).

Blank spaces and lines

Blank spaces in MEL statements are, for the most part, ignored by the MEL interpreter. The following two lines of code are interpreted in the same way:

```
vector $myVect=<<1.2,2.3,3.4>>+<<2.1,3.2,4.3>>;
// Result: <<3.3, 5.5, 7.7>> //

vector $myVect = << 1.2, 2.3, 3.4 >> + << 2.1, 3.2, 4.3 >>;
// Result: <<3.3, 5.5, 7.7>> //
```

Used wisely, blank space can improve the legibility of your MEL code. Similarly blank lines are ignored and can therefore enhance legibility.

Functions

Functions are MEL commands used with values and variables. Most functions perform mathematical operations as in the following example featuring the trigonometric (periodic) function, cosine:

```
float $pi = 3.14159265;
float $y = `cos $pi`;
// Result: -1 //
```

Functions can be written using imperative or function syntax. For example, the cosine function above could also be written as:

```
float $y = cos($pi);
// Result: -1 //
```

Maya has over 50 functions which are well documented in the Help Library:

MEL functions

Maya Help → Using Maya → General → MEL and Expressions → Useful functions

MEL command reference library

As of Maya release 8.5 there were several thousand MEL commands spanning a broad range of functionality in Maya—from system utilities to modeling. In this book we will introduce many of the commands relevant to learning the in silico biology Maya workflow. A complete list of MEL commands, their flags, and arguments is available in the MEL command reference in the Maya Help Library.

MEL command reference

Maya Help → Commands

MEL commands we know and love

When asked to relate best practices for MEL scripting, Joyce Janczyn (MEL's author) stressed the importance of learning the commands so that you'll be able to use them quickly and efficiently.[1] Table 12.04 lists our 10 most frequently used MEL commands and the ones we have committed to memory.

Make a shelf button from a MEL command

Shelf buttons are handy for launching windows and editors. What a button actually does when you press it is run a MEL statement or script. You can attach any script you like to a button; each time you press the button, the script executes. The following example demonstrates how to capture a script from the Script Editor and turn it into a button. The script in this case is a single MEL command used to launch the Expression Editor.

1. **Enter the following text in the Command Line:**

   ```
   ExpressionEditor;
   ```

2. **If your shelves are not visible, choose Display → UI Element → Shelves.**
3. **Click on your Custom shelf tab. If you do not have a custom shelf, create one now:**
   - (a) **Window → Settings/Preferences → Shelves.**
   - (b) **Under the Shelves tab press the New Shelf button and name it "Custom"**

MEL command	Sample use	Result
Select	`select mySphere;`	Selects the object called mySphere.
getAttr	`getAttr mySphere.translateX;`	Returns the value of the specified attribute, translateX.
setAttr	`setAttr polySphere1.radius 5;`	Sets the value of the specified attribute.
connectAttr	`connectAttr myGlobe.tx myCube.rz;`	Connects the output value of the first attribute to the input of the second.
addAttr	`addAttr -longName newAttr myGlobe;`	Adds a custom attribute called "newAttr" to the object myGlobe.
rand	`rand 0 1;`	Returns a pseudorandom number between 0 and 1.
ls	`$var = ` ` `ls-transforms "poly*"` ` `;`	Returns the names of items in your scene—in this example, objects with a transform node name starting with "poly".
size	`size $var;`	Returns the length of an array.
clear	`clear $var;`	Clears the memory being used by an array variable and resets the array length to zero.
print	`print "Hello world";`	Prints the argument ("hello world") to the Script Editor and Command Line Result field.

TABLE 12.04

Our **MEL Top 10**: MEL commands that we use most regularly in our Maya-based in silico biology work. The commands are displayed in imperative syntax.

(c) **Press the Save All Shelves button.**

(d) **Select your Custom shelf tab.**

4. **Open the Script Editor and select the command in the History field:**

```
ExpressionEditor;
```

5. **LMB+ or MMB+drag the command to your Custom shelf and release the mouse button.**
6. **Open the Shelf editor: choose Window → Settings/Preferences → Shelves.**
7. **Choose the Shelf Contents tab, then locate and select the newly added** `Expression Editor` **command.**
8. **In the Icon Name field at the bottom of the editor, type the short name** `EE` **(**Figure 12.04**).**
9. **Press the Save All Shelves button.**

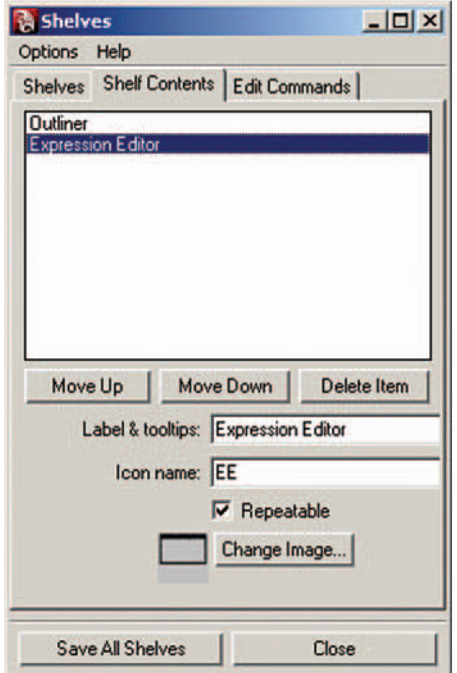


FIGURE 12.04
Use the Shelf Editor to arrange and rename shelf buttons.

Now anytime you want to launch the Expression Editor you can do so easily by pressing the EE button on your Custom shelf.

Attributes in MEL

You have done a lot with attributes in the previous chapters—reading and setting their values in the Channel Box and Attribute Editor. When used in MEL statements, attribute names are slightly different than they appear in the Channel Box and Attribute Editor; they use **dot notation** as follows:

```
objectName.attributeName
```

Getting, setting, and connecting attributes

By now you've learned that working in Maya boils down to manipulating attribute values; models, shaders, and animation are all characterized by attributes. Of the thousands of MEL commands, arguably the most useful are those that query, set, and connect attributes because they directly affect the DG, modifying and animating your models. The following code listing demonstrates these three important command types. Note our use of comments to document each step.

```
// Create a polygonal sphere.
polySphere -name myGlobe;

// Set translateY to a random value less than or equal to 20.
setAttr myGlobe.translateY `rand 20`;

// Create a polygonal cube.
polyCube -name myCube;
```

Attribute type	Sample attribute	Sample data
float	mySphere.translateX	3.14159265
boolean	mySphere.visibility	"yes" or "no"; 1 or 0
vector	mySphere.translate	<< 5.5, 1.1, 6.6 >>
int	polySphere1.subdivisionsX	20
string	polySphere1.customAttribute	"concentration"
vector array	swarmShape.rgbPP	{ << 1, 0, 0 >>, << 0, 1, 0 >>, << 0, 0, 1 >> }
float array	swarmShape.mass	{ 4.5, 12, 6.2 }

TABLE 12.05

Attribute data types. The last two types pertain to **per particle attributes**, for which each element pertains to a specific particle within a particle object.

```
// Get the translateY value of myGlobe and store it in a variable.
float $transY =`getAttr myGlobe.translateY`;

// Set the translateY value of myCube using the variable $transY.
setAttr myCube.translateY $transY;

// Use the sphere's X position to drive the cube's Z rotation.
connectAttr myGlobe.translateX myCube.rotateZ;
```

In the example above, dot notation was used for attribute names. Note the difference between the following two statements. In the first, the `rotateY` attribute is written as it appears in the Channel Box. The second statement uses the correct dot notation to refer to the attribute and executes without error.

```
setAttr myGlobe Rotate Y 45;                    (incorrect)
// Error: line 1: No attribute was specified. //

setAttr myGlobe.rotateY 45;                     (correct)
```

Like variables, each attribute in Maya has data type. Table 12.05 lists attribute data types with examples of each.

Setting attributes with the "type" flag

Unlike single-value, numerical attributes—like translateX, visibility, and so on—some attributes must be set using the type flag and an appropriate argument. Two common examples include compound transform and string attributes:

```
setAttr myGlobe.translate -type double3 5 10 15;
setAttr someObject.customAttribute -type "string" "someValue";
```

String attributes are rare in Maya and usually take the form of a custom attribute in which you want to store textual information. Unlike numerical attributes like `translate` and `scale`, string attributes cannot be keyframed.

Conditional statements

Conditional statements choose a course of action in a computer program by testing one or more conditions and take the following basic form:

if (some condition is met)

do something

else if (some other condition is met)

do something else

Maya has two types of conditional statement: **if...else** and **switch...case**. The following is an example of a typical if...else statement. It tests a random number against several conditions then assigns a variable and prints a message base on the result.

```
string $myColor;
float $rnd = `rand 9`; // Pick a random number between 0 and 9.
if ($rnd <= 3) {
    $myColor = "red";
    print "Win!";
}
else if ($rnd > 3 && $rnd <= 6) {
    $myColor = "blue";
    print "Lose!";
}
else {
    $myColor = "green";
    print "Draw!";
}
```

The final else statement is a catch-all if none of the previous conditions are met. Curly brackets are used to enclose the contents of each if, else...if, and else statement. Note that semi-colons are not used to terminate the statements. If the contents of a condition statement don't exceed one line, you can omit the curly brackets and shorten each statement to a single line, as in the following example:

```
float $rnd = `rand 3`;
if ($rnd <= 1) print "Win!";
else if ($rnd > 1 && $rnd <= 2) print "Lose!";
else print "Draw!";
```

The switch...case statement evaluates an expression against several predetermined cases. When it matches the expression to the value of a case it executes the corresponding code. A break statement exits the switch...case statement once a condition has been met. switch...case looks for an *equality* between an expression and a case and cannot match them based on an *inequality* as we did in the if...else examples above. In the following example the ceil (short for ceiling) function is used to return next highest integer value above the randomly generated number.

The **ceil** function returns the next highest integer above its argument—a float value.

```
int $rnd2 = ceil(`rand 3`); // Function syntax wrapping imperative
  syntax.
switch ($rnd2){
   case 1:
         print "Win!";
         break;
```

```
    case 2:
          print "Lose!";
          break;
    default:
          print "Draw!";
          break;
}
```

Note that the imperative command, `rand 3`, is wrapped within function syntax, ceil(). The expression could be written as it appears above or as:

```
int $rnd2 = ceil(rand(3)) // Function syntax.
```

but not as:

```
int $rnd2 =`ceil `rand 3``; // Imperative syntax wrapping imperative
  syntax.

// Error: int $rnd2 =`ceil `rand 3``; //
// Error: Line 1.23: Syntax error //
```

Indenting lines

Indenting lines of code is not a syntactic requirement of MEL but is a style choice that makes your scripts easier for humans to read. We highly recommend it. We typically indent lines using tab characters, which are ignored by the MEL interpreter. Many text editors designed for computer coding indent lines automatically.

Loops

A loop is a block of code that repeats for as long as some condition is met. Loops are essential to in silico simulations involving multiple objects like cells and molecules, and for constructing models out of repeated subunits like the ones you'll build beginning in *Chapter 14*. MEL supports four kinds of loop: for, for-in, while, and do…while.

The for loop

The following is a typical example of a for loop, involving a counter integer used to increment through the code block.

```
int $i;
for ($i = 1; $i <= 5; $i++ ) {
   print $i;
   print "\n";
}
```

The first term within the brackets states the starting condition, the second term states the ending condition, and the third increments the counter by 1. Upon running the above for loop, the printed result in the Script Editor is:

```
1
2
3
4
5
```

Note the use of the "new line" notation, "\n", to print each new value of $i on a separate line. The above for loop could also be written more concisely as:

```
for ( $i = 1; $i <= 5; $i++ ) print( $i + "\n" );
```

The for-in loop

The for-in loop is a tidy way to increment through the elements of an array, as in the following example:

```
string $cell;
string $cellType[] = {"tCell", "fibroblast", "keratinocyte",
  "monocyte"};
for ( $cell in $cellType ) print( $cell + "\n" );
```

The printed result in the Script Editor is:

```
tCell
fibroblast
keratinocyte
monocyte
```

The while loop

A while loop executes for a long as the condition in brackets is true. For example:

```
int $i = 0;
while ( $i <= 10 ) {

    $i += 2; // The += assignment operator adds 2 to the current
      value of $i.
    print( $i + "\n" );
}
```

The printed result in the Script Editor is:

```
2
4
6
8
10
12
```

The do...while loop

A do...while statement evaluates the condition at the *end* of the code block rather than the beginning like the while loop. The code is executed as long as the condition is true.

```
int $i = 0;
do {

   $i += 2; // The += assignment operator adds 2 to the current
     value of $i.
   print( $i + "\n" );
}
while ( $i <= 10 );
```

The printed result in the Script Editor is the same for the code above as for the while loop. Use caution with while and do…while statements; a simple error in the code can put your computer into an infinite loop, from which the only recovery is to force-quit Maya.

The Maya Help Library is an excellent source of information and examples about statements that control the logical flow of your MEL scripts:

MEL script flow control

Maya Help → General → MEL and Expressions → Controlling the flow of a script

Procedures

A procedure is a user-defined, self-contained set of MEL statements that carries out specific operations or actions. A procedure is executed (or *called*) with a single command: its name. Procedures are similar to pre-defined MEL functions like cos or rand; they take arguments, make calculations, and then return results. Procedures have three things in common with variables:

In some programming languages procedures are known as subroutines or functions.

1. **Procedures have data types (or *return types*) which are the same as the data types available for variables in Maya.**
2. **A procedure must be declared before it is used.**
3. **Procedures can be either local or global. A local procedure is available only within the script or expression in which it resides. A global procedure, on the other, is available at any time (after being declared), to any script or expression within the Maya environment.**

A procedure declaration takes the following basic form:

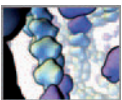
```
global proc returnType procedureName (arguments) {
    // MEL statement.
    // MEL statement.
    // MEL statement.
    // etc…
}
```

Omit the word global for local procedures. Arguments must be declared by type as you'll see in the following sample procedure which prints a message in the Script Editor.

1. **Open the Script Editor.**
2. **Type the following in the command input panel:**

```
global proc string headBump(int $monkeyNum, string $location) {
    // Declare and assign variables.
    string $little = " little monkeys jumping on the ";
    string $verse = $monkeyNum + $little + $location + "!";
    // Assign the return value.
    return $verse;
}
```

3. **Press Enter.** This declares the procedure—it is now loaded in memory and can be called at any time using the procedure name, headBump. headBump will be cleared



from memory when you quit Maya, so it must be re-declared when you restart Maya if you wish to use it again.

4. **In the Script Editor choose Edit → Clear All.**
5. **Call the procedure using its name and arguments: type the following into the Script Editor or Command Line and hit Enter.**

```
string $story = headBump(5, "bed");
```

Maya displays the procedure result (its return value) in the Script Editor and the Command Line:

```
5 little monkeys jumping on the bed!
```

Of course, you'll be using procedures for more than building nursery rhymes; procedures are a highly useful tool in the in silico biology workflow. They allow you to package sets of instructions that can be called only when needed. As your work with procedural techniques in Maya advances, and your MEL scripts get longer and more complex, procedures are good way to keep your code organized.

Sourcing procedures

Remember that If you add files to the Scripts directory when Maya is running, you will have to refresh the search path contents, using the **rehash** command, in order to have access to those files and their contents.

For the example above, entering the procedure into the Script Editor is a simple way to declare it and is a good way to declare short procedures in general. When you compose a longer procedure in an external text editor, you can copy and paste it in the Script Editor then press Enter to declare it. However, this can get tedious, particularly when you need to declare several procedures so that they'll all be available to Maya at once; locating and opening the text files, then copying and pasting the code each time you start Maya takes up valuable time. For this reason we recommend you take advantage of Maya's built-in script sourcing capabilities via the MEL script search path. When you call an undeclared procedure (e.g. `myProcedure()`)—for instance, by typing its name in the Script Editor—Maya scans its search path contents list for a MEL script file of the same root name (e.g. `myProcedure.mel`). If the file exists, Maya loads it into memory, thereby declaring any procedures the file contains.

One file—one procedure

Mike Taylor has been a member of the Maya developers team since 1995, prior to the launch of Maya version 1.

While it may be tempting to gang up multiple procedures within a single MEL script (text) file, it's not advisable according to veteran Maya developer Mike Taylor. He recommends saving each procedure out in its own file. "Maya knows how to automatically load the script that defines a global procedure *as long as the file name matches the procedure name*. If you include multiple global procedures in a single MEL file, then you loose that benefit and you add the overhead of making sure your global procedure is defined before you call it." Throughout the projects in *Part 03* of this book we recommend you to take Mike's advice and save each global procedure in a distinct text file that is named appropriately. For instance, a procedure called `rule1()` should be saved in a file called `rule1.mel` within your Maya Scripts directory.[2]

Animation expressions

Animation expressions are the engines of procedural animation in Maya. Generally speaking, an animation expression (**expression** for short) is a set of instructions used to animate one or more attribute. Typically, an expression evaluates every time the Maya frame number changes, meaning that the instructions are processed in regular

FIGURE 12.05

The expression Editor is where you'll typically interact with animation expressions in Maya.

time increments when Maya is in playback mode. In the following example you'll make a sphere and an expression to animate its `translateX` attribute.

1. **Start a new Maya scene: choose File → New Scene.**
2. **Open the Script Editor: type `ScriptEditor` in the Command Line and press Enter (remember that MEL is case sensitive).**
3. **Use a MEL command to set the playback options for your scene:**

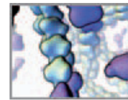
   ```
   playbackOptions -playbackSpeed 1 -loop continuous -min 1 -max 150;
   ```

4. **Create the sphere. Enter the following in the Script Editor:**

   ```
   polySphere -r 1 -n mySphere;
   ```

5. **With the sphere selected, highlight its Translate X attribute in the Channel Box.**

The playbackSpeed flag sets the scene view playback speed to a multiple of the frame rate. To alter the frame rate (say, from 24 fps (film) to 30 fps (NTSC)) you must open up Preferences and make the change under Settings → Time.



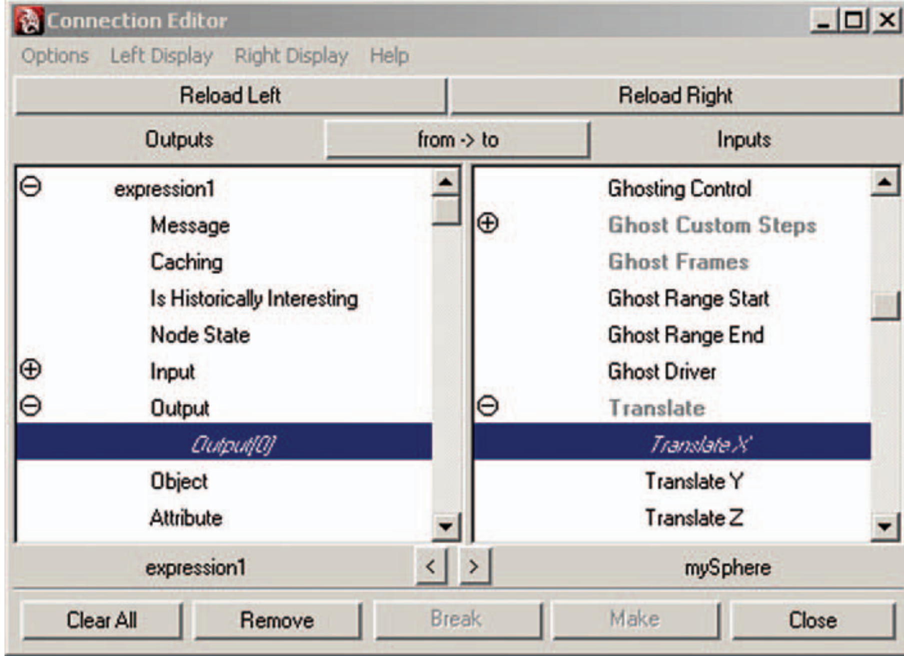


FIGURE 12.06

The Connection Editor showing the connection between an expression node and the attribute it controls. The settings under the Left Display and Right Display menus dictate what types of attributes will be shown (e.g. *keyable or non-keyable* attributes). In most cases, you'll interact with relatively few of the attributes displayed in the Connection Editor, such as the Translate attributes shown for mySphere in the right panel. Attributes that are connected (or *driven*) are displayed in italics (oblique text).

6. **RMB+click on Translate X and choose Expressions. This launches the Expression Editor with the attribute highlighted in the Selected Object & Attribute field.**

7. **Select** `mySphere.translateX` **in the Selected Object & Attribute field, and then LMB+drag the text into the Expression text field.**

8. **Create the expression shown in** Figure 12.05 **by adding "**`= frame - 1;`**" to the text.**

9. **Press the Create button to finish.**

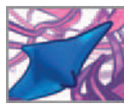
Maya will assign a default name to your expression which is displayed in the Expression Name Field of the Expression Editor.

10. **In the timeline controls of the main window, press the Play button.**

As the frame number increases, so does the X position of your sphere due to the expression. The result is a simple procedural animation similar to the one you made back in *Chapter 06*. From this, you can see how MEL can be used in place of the UI tools (Move, Rotate, and so on) to animate attributes. Next, let's take a look at what's going on behind the scenes when you use an animation expression.

The animation expression node

An animation expression is itself a type of **DG node**. It has attributes and is part of the **DG**, with direct access to all other nodes and their attributes. Figure 12.06 shows an expression represented in the Connection Editor: in the left panel are the expression



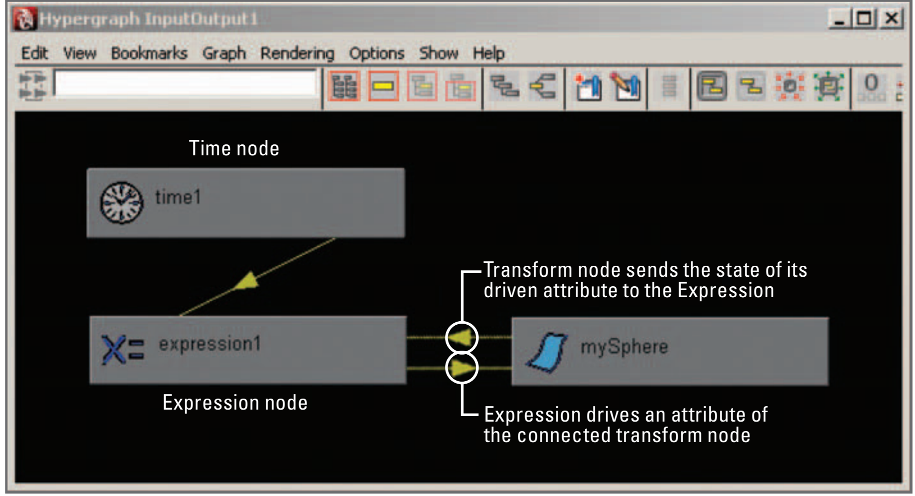


FIGURE 12.07

An expression node is driven by Maya's time node and in turn drives the attributes to which it's connected.

attributes; in the right panel are the attributes of mySphere. expression1.output drives mySphere.translateX. Another way to visualize this relationship is in the Hypergraph (Figure 12.07) where you can also see that Maya's time node drives the expression.

The Expression Editor

Although you can create and edit expressions using the MEL command expression, it is often more convenient to do so using the Expression Editor (Figure 12.05). We'll cover a few of this important editor's key features here. The **Select Filter** menu lets you choose how items are listed:

1. **By Expression Name**:

   for example, expression1

2. **By Object/Attribute Name**:

   displays attributes and expressions (if any exist) for selected objects

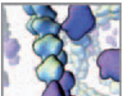
3. **By Script Node Name**

   (a Script Node is a DG node which stores a MEL script)

The Editor menu lets you choose to edit your expressions in the Expression Editor or in a default text editor on your system. For more information on using this feature, refer to Maya's Help Library:

Linking an external text editor to Maya's Expression Editor

Maya Help → General → MEL and Expressions → Animation expressions → Edit an animation expression with a text editor



Converting units

When Maya assigns and queries an attribute or internal variable whose value is a measurement unit—distance, playback speed, or an angle—the program does so using its default internal units of centimeters, 24 fps, and radians, respectively. If you alter any of these default working units in Preferences, Maya converts the attribute values that you specify to its default working units behind the scenes. For example, suppose you changed the Angular working units from radians to degrees—like you did back on page 266. When you assign a Rotation attribute value for an object, Maya will convert that value from degrees to radians for storage in the object's transform node. The attribute value will still be displayed in degrees in the Channel Box (and other UI editors, such as the Attribute Editor), but Maya will work with its equivalent value in radians for any calculations it makes in order to animate the rotation of the object.

The Convert Units settings in the Expression Editor tell Maya whether or not to convert All three types of unit, None of the three, or Angular (units) only. When you specify None, Maya will treat all distance units as centimeters and all angles as radians, regardless of the working units you specified in Preferences. When you specify All or Angular only, and set the default working units in Preferences, Maya will not convert units. In other words, unit conversion in animation expressions only happens when you're using non-default working units (set in Preferences) and specify unit conversion other than None in the Expression Editor.

Converting from non-default units adds extra computation steps and can therefore slow down the execution of an animation expression. For optimal speed, it's best to set working units in Preferences to their default values of centimeters, 24 fps, and radians. However, this is not always practical: you may find it more intuitive to work with angles in degrees rather than radians, and your project may call for distance units other than centimeters. Therefore, the choice of working units comes down to a trade-off between convenience and speed.

The Create and Edit buttons

Maya won't create or update an expression node until you press either the Create or Edit buttons at the bottom of the Expression Editor. Because of this requirement for manual intervention, via the UI it is easy to lose changes you've made while editing an expression. The Reload button reloads from memory the version of the expression that was stored last time you pressed the Edit button; it allows you to undo changes to an expression before pressing Edit.

You will get to know the Expression Editor quite well as you work through this book. For complete documentation, refer to the Help Library:

The Expression Editor

Maya Help → General → MEL and Expressions → MEL Windows and Editors → Expression Editor

Animation expression syntax

Expression syntax differs in two ways from the syntax of MEL statements that you enter through the Script Editor or use in procedures. First, you can query, assign, and

connect attribute values directly in an expression, whereas you must use the MEL commands, getAttr, setAttr, and connectAttr in a MEL script to do the same. In the following example, we set our sphere's translateY attribute to 5. The first line is something you might enter in the Script Editor or Command Line but will *also* work in the Expression Editor. The second line is proper *expression* syntax but will fail if entered as a *MEL statement* in the Script Editor.

```
setAttr mySphere.translateY 5;   // Standard MEL syntax.
mySphere.translateY = 5;         // Expression syntax.
```

The commands **getAttr** and **setAttr** can be used in animation expressions but are unnecessary in many cases since you can directly assign attribute values using the "equals" operator (=).

Secondly, expressions have two native variables, frame and time, that are set automatically through an input connection from the time node. These variables are not available to scripts outside of an expression. If you want to query the frame number outside of an expression, use the currentTime MEL command as in the following example:

```
// Incorrect:
int $myVar = frame;
// Error: int $myVar = frame; //
// Error: Line 1.19: Invalid use of Maya object "frame". //

// Correct:
int $myVar =`currentTime -query`;
// Result: 44 //
```

The expression command

When you press Create to make a new expression in the Expression Editor, Maya executes a MEL command—as it does with most other actions you perform in the UI. To demonstrate the expression command, let's recreate expression1 for mySphere, as follows:

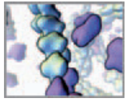
1. **Open the Script Editor and choose Edit → Clear All.**
2. **Open the Expression Editor: choose Window → Animation Editors → Expression Editor.**
3. **Choose Select Filter → By Expression Name.**
4. **Select** expression1 **in the Expressions field and press the Delete button.** This deletes the expression node from the DG and the expression code from the Expression Editor.
5. **Enter the following statement in Expression Editor text field, then press Create:**

   ```
   mySphere.translateX = frame - 1;
   ```

6. **Inspect the History field of the Script Editor. You should see the following MEL statement:**

   ```
   expression -s "mySphere.translateX = frame - 1;" -o "" -ae 1 -uc all ;
   // Result: expression1 //
   ```

The first flag, -s, indicates the expression "string" which holds the contents of the expression. The second flag, -o, is short for "object"—the default object for the expression. In this case, since you set the selection filter to Expression Name instead of Object/Attribute Name, the expression has no default object associated with it,



hence the empty quotation marks. The flag -ae is short for alwaysEvaluate. When this is set to 1, or true, the expression will evaluate each time the frame number changes. Finally, -uc is short for unitConversion. Its setting of all ensures that all times, distances, and angles will be converted to standard working units (frames, centimeters, and radians) for the purpose of making calculations, even if you specified different units in Preferences (see *Converting units* on page 296). Since -object, -alwaysEvaluate, and -unitConversion are set to their default values they could be omitted without any effect on the expression.

Next, let's use the expression statement that Maya produced in order to make a new expression. With the Expression Editor still open,

1. **Open the Script Editor and choose Edit → Clear All.**
2. **Open the Expression Editor and choose Select Filter → By Expression Name.**
3. **Select** expression1 **in the Expressions field and press the Delete button.**
4. **Enter the following statement in the Script Editor:**

```
expression -s "mySphere.translateX = frame - 1;" -name
  myFirstExpression;
```

5. **You should see the following result displayed in the History Panel:**

```
// Result: myFirstExpression //
```

You've just created your first expression using MEL! Its name will appear in the Selection → Expressions field of the Expression Editor. When you select it by name, it can be edited in the Expression text field.

Unlike a procedure, each time you run the **expression** command, Maya creates a new animation expression—it does not replace a previous version of it. It is good practice to delete redundant animation expressions in the Expression Editor since they may give unpredictable results.

Stand-alone animation expressions

Despite the emphasis we've put on the role of expressions in animating attributes, an expression can stand alone without any connections to objects and their attributes. For instance, an expression may be used only to make calculations and update global variables. In this role, the expression is a computer program that runs every time the frame number in your scene changes. By including a loop (do...while, for...in, and so on) in the "program" can run multiple times for each frame. For in silico modeling work, we use expressions both to make important calculations for the scene and to update attributes based on those calculations. Let's look at a simple example of this kind of use: an expression that moves a sphere around the scene randomly and displays its location in the Script Editor.

1. **Create a new scene: choose File → New Scene.**
2. **Open the Script Editor.**
3. **Use a MEL command to set the playback options for your scene:**

```
playbackOptions -playbackSpeed 1 -loop continuous -min 1 -max 150;
```

4. **Create a sphere. Enter the following in the Script Editor:**

```
polySphere -r 1 -n mySphere;
```

5. **Press your custom shelf button, EE, to launch the Expression Editor.**
6. **Choose Select Filter → By Expression Name.**

Convert Units: All None
Particle: Runtime before dynamics Runtime after dynamics
Evaluation: Always
Editor: Expression Editor
Expression:

```
if (frame <= 1) mySphere.translateX = mySphere.translateZ = 0;
else {
        // Move the sphere randomly in the XZ plane.
        mySphere.translateX += `rnad -1 1`;
        mySphere.translateZ += `rnad -1 1`;
        // Print its coordinates to the Script editor.
        print ("(x, z) = (" + mySphere.translateX + ", " + mySpher
};
```

Edit Delete Reload Clear Close

FIGURE 12.08

In the Expression Editor, attributes can be directly assigned values using the equal (=) operator.

7. **Enter the following in the Expression text field** (Figure 12.08) **to control the translateX (tx) and translateZ (tz) attributes:**

```
if (frame <= 1) mySphere.tx = mySphere.tz = 0;
else {
    // Move the sphere randomly in the XZ plane.
    mySphere.tx +=`rand -1 1`;
    mySphere.tz +=`rand -1 1`;
    // Print its coordinates to the Script editor.
    print ("(x, z) = (" + mySphere.translateX + ", " + mySphere.
      translateZ + ")\n");
}
```

Note the conditional `if` statement which is used to reset the sphere to the world origin each time the playhead returns to frame 0 or 1.

The **print** command can print multiple values at once, as long as they are enclosed in curved brackets. The contents of the brackets are added together as a string and then printed.

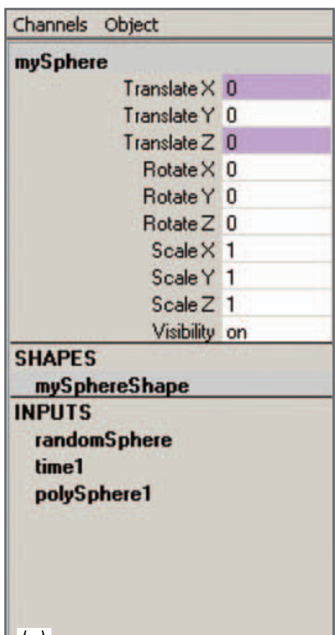
8. **Press the Create button.**

9. **In the Expression Name field, type: randomSphere.**

10. **Open the Script Editor, then re-size and position it such that you can clearly see the History field and a view of the sphere in the scene view.**

11. **In the timeline controls of the main window, press the Play button.**

Because your expression is, by default, set to evaluate “always” it will run once for each frame as your scene plays. The result is a random walk of the sphere in the XZ plane, with the accompanying coordinates displayed in the Script Editor. Notice in the Channel Box, that the Translate X field for the sphere is colored purple (Figure 12.09a); this indicates a connection to an expression node.



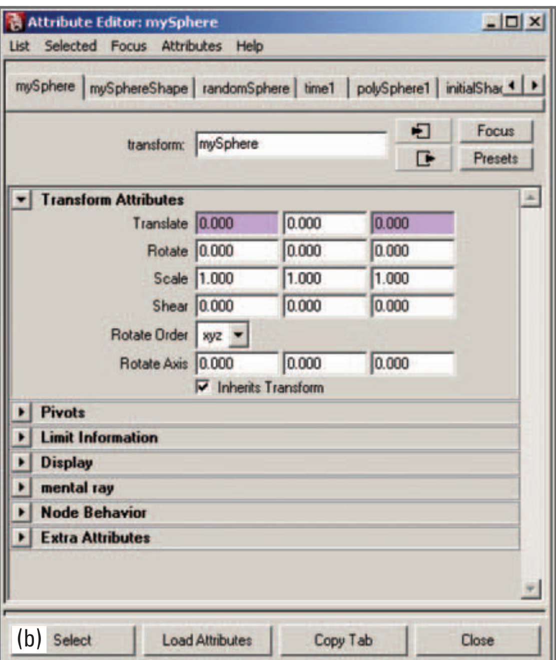


FIGURE 12.09

When connected to an animation expression node, attributes are colored purple in (a) the Channel Box and (b) the Attribute Editor. Note the Inputs listed in the Channel Box. **randomSphere** is the name of an expression to which Maya's time node is connected. **polySphere1** is the history node for the polygon geometry.

Line breaks in animation expressions

In the example above you entered the multi-line expression directly into the Expression Editor. However, if you wish to enter it via the `expression` MEL command, you must instruct Maya to ignore the internal quotation marks (i.e. those associated with the `print` command). Otherwise, Maya will take the second quotation mark it encounters to mean the end of the expression command string. Instead you want Maya to interpret only the first and last quotation marks as defining the expression string, while internal quotes are read in as part of the string. To accomplish this, you'll type a back slash character, \, before every quotation mark you want Maya to ignore. In this usage, we say the back slash *escapes* the quotation mark.

As well, because the `expression` command requires a continous string of characters, you must also escape line breaks (or *carriage returns*), using the character combination of \n\. Here is a simple example intended to make an expression that prints a message to the Script Editor.

```
expression -s "print("This is frame: " + frame + "\n")" -n myExpr;
```

```
// Error: expression -s "print("This is frame: " + frame + "\n")"
  -n myExpr; //
// Error: Line 1.36: Syntax error //
```

In the above statement, the quotation marks aren't escaped and Maya interprets the expression string as "`print(`", which is followed by indecipherable code. Properly escaped, the above statement should look like this:

```
expression -s "print(\"This is frame: \" + frame + \"\\n\")" -n
  myExpr;
```

Note that even the line break, \n, must be escaped by placing a back slash in front of it; otherwise Maya will read the lone \ and escape the n instead of \n together. The only quotation marks that aren't escaped are those enclosing the entire expression string. Entered as a MEL command the randomSphere expression you created on page 299 would be entered as follows:

```
expression -s "\n\
    if (frame <= 1) mySphere.tx = mySphere.tz = 0;\n\
    else {\n\
        // Move the sphere randomly in the XZ plane.\n\
        mySphere.tx +=`rand -1 1`;\n\
        mySphere.tz +=`rand -1 1`;\n\
        // Print its coordinates to the Script editor.\n\
        print (\"(x, z) = (\" + mySphere.tx + \", \" + mySphere.
          tz + \")\\n\");\
        n\
    }\n\
" -name randomSphere;
```

A simple attribute (like translateZ) cannot have more than one input. Therefore, an attribute controlled by an animation expression cannot have keyframes assigned to it as well.

You can imagine that once an expression script grows past a several lines, escaping quotes and line breaks can become a genuine nuisance and a potential source of errors. For this reason we generally avoid the expression command. Instead we enter expressions either directly in the Expression Editor or compose them in an external text editor, then copy and paste them into the Expression Editor. The latter approach will be used throughout the remainder of this book.

Putting it all together: The MEL script

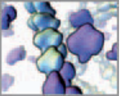
Now that you're familiar with the components of scripting in Maya, let's see how they can work together in a complete script. Because MEL is so completely integrated in Maya, we encourage you to use it whenever possible in order to work more efficiently—for example, by making shelf buttons to automate common tasks that might otherwise take longer to do through menu selections. However, where MEL truly shines is in the form of scripts to do things that would be difficult and tedious, if not impossible using the UI menus and tools: build complex models and run procedural animations.

A MEL script can take the form of a procedure, an animation expression, or a set of MEL instructions that executes when sourced from a text document with the file extension .mel. Some MEL scripts create custom UI windows that provide access to modeling tools. Other MEL scripts create animation expressions and load procedures into memory. There are even MEL scripts that build custom shading networks. Since Maya version 1 hit the market, thousands of scripts have been written for various purposes. Many of these are integral to custom workflows in animation studios, while others have been made freely available on 3D animation community websites. In the *Further reading* section we have listed a few of these community websites.

In the following tutorial you'll explore the structure of a typical script, and how the components we've described in this chapter—variables, attributes, commands, loops, conditional statements, etc.—fit together.

Text editors for writing computer code

For the rest of this book, the tutorials and projects involve scripts that are sufficiently long to warrant composing them in an external text editing program—preferably one that is designed for editing computer code—a **programmer's editor**—with automatic tabbing,



line numbering, and highlighting of special structures like strings. Below we've listed four text editors that are available for Windows either free of charge or for a nominal fee.

jedit
http://www.jedit.org/

Microsoft Visual Studio Express
http://msdn2.microsoft.com/en-us/express/

TextPad
http://www.textpad.com/

UltraEdit
http://www.ultraedit.com/

Presentation of the MEL scripts in this book

Much of what follows in this book is concerned with building complete MEL scripts to visualize and simulate phenomena in cell biology. In each tutorial and project, we present scripts in their entirety, interspersed with instructions and explanations. All MEL code is indented from the left page margin and set in the following typeface to make it easy to recognize.

```
// This is MEL code.
```

The beginning and end of a script will be clearly indicated in the explanatory text. Furthermore, when a script ends, you'll see a comment such as:

```
// End procedure.
```

Occasionally, the number of letters and symbols in a MEL statement will exceed the paragraph width of our book. To show the statement to you, we will have to break the statement at a convenient point and continue it on the next line of our book's text. When this happens, we will indent that next line (and, as needed, any subsequent lines) to mark the continuation. You, however, should type the code statement as one unbroken string! If, via your text editor, you were to put line feeds and carriage return commands in the middle of a MEL statement, the Maya will not be able to parse the statement correctly and you will get error messages. So far example in typing the MEL statement

```
setAttr ($name + ".inPosition") -type double3 ($pos.x) ($pos.y)
  ($pos.z);
```

you, in typing up that statement yourself, would not hit your 'Enter' key between typing "($pos.y)" and "($pos.z)".

As you follow along, you can build each script yourself in a text editor or simply test individual pieces of code by entering them in Maya's Script Editor or Expression Editor. The complete scripts are included on the accompanying CD-ROM within appropriate directories. For example:

18_Cell_Migration/MEL/cRule.mel

Tutorial 12.01: Building a MEL script

Other than using correct syntax and ensuring that information flows logically, there are no firm rules for building a script—just ideas and hard-won practical insights

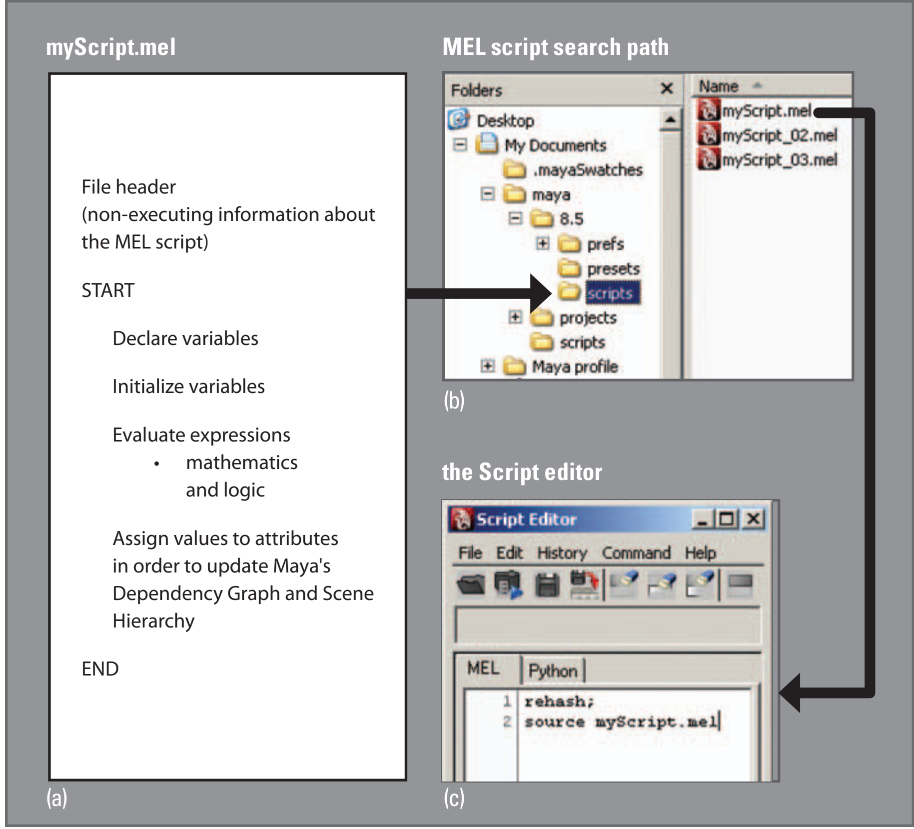


FIGURE 12.10

(a) The format used for MEL scripts in this book. (b) A typical MEL script search path for Maya. (c) You can load a script into Maya using the **source** command. First typing the **rehash** command refreshes the search path contents list.

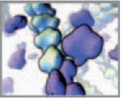
about style and form. As we mentioned earlier, MEL was designed with artists in mind and is therefore quite relaxed in its requirements for structure. What follows is a description of the form that scripts in this book take. Figure 12.10 outlines this form and shows how the script file relates to the search path vis-a-vis sourcing through the Script Editor.

In this tutorial you'll build a procedure that creates spheres and distributes them randomly within a cube of user-specified dimensions (Figure 12.11). The ability to quickly make and distribute objects in space is essential to 3D in silico biology simulations, which usually involve multiple interacting agents such as cells and molecules, and for structural models composed of many parts like the molecules and tissue scaffolds you'll make in *Chapters 14* and *17*, respectively.

Getting started

Open a text editor that is external to Maya and start a new file. As you work through this tutorial, enter the code in the text file in the order it's presented. Save the file under the name makeSpheres.mel within your Maya Scripts directory. If you don't know the path to this directory, open up Maya and run the following statement by entering it in the Command Line:

```
internalVar -userScriptDir;
```



Save your file periodically as you work. When you're done, you'll source your MEL script in Maya. If you wish to work with a ready-made file, we've included the complete MEL script on the CD-ROM:

12_MEL_Scripting/MEL/makeSpheres.mel

The file header

The file header is commented documentation. It usually includes: title; author name(s); creation and modification dates; a brief description of what the script does and how to run it in Maya. Here's the header for our sample procedure:

```
/***** FILE HEADER *****/
/*
makeSpheres.mel
Created 01 September 2007.
Modified 27 September 2007.
Modified 21 October 2007.
Authors: Jason Sharpe, Charles Lumsden, Nick Woolridge

Description:
This procedure makes polygon spheres and distributes them
throughout a cube. The procedure arguments are as follows:

$count       The number of spheres to make.
$radius      The sphere radius.
$cubeSize    The dimensions of the cube.

To use this script:

Save this entire script in a text file, using the .mel extension, in
your Maya Scripts directory, then source it through Maya's Script
Editor. Alternately, you can copy and paste the entire script into
Maya's Script Editor.
*/
```

Declare the procedure

This step is unique to procedures in Maya and wouldn't be taken for an expression. Following the bracketed procedure arguments, the procedure's contents are enclosed in curly brackets. Indenting the contents helps to distinguish them from the procedure command itself.

```
global proc makeSpheres (int $count, float $radius, float
  $cubeSize) {
```

Declare your variables

According to Maya's syntax rules, a variable needs only to be declared immediately before it's used. Nonetheless, grouping and declaring variables together makes them easier to keep track of, especially as your scripts begin to grow in length; it's much easier to refer back to one spot in your script to check the type and meaning of a certain variable than to hunt for it throughout your procedure. It is also helpful to others who use your scripts, and a good reminder for yourself, to provide some documentation describing what variables do.

```
/***** DECLARE THE VARIABLES *****/

/*
$half               Half the cubesize: used to position the cube.
$x, $y, and $z      Used to position each sphere.
$d                  The separation between evenly distributed
                    spheres.
*/
float $half, $x, $y, $z, $d;

/*
$cubeName[]         The return value of the polyCube command:
                    the transform node name.
$cubeShaderName     The return value of the shadingNode command:
                    the cube shader name.
$sphereName[]       The return value of the polySphere command:
                    the transform node name.
*/
string $cubeName[], $cubeShaderName, $sphereName[];

/*
$i              A counting index.
*/
int $i;
```

Content enclosed by the /* and */ characters is commented out and won't be executed by Maya. We use the character string, **** to highlight section headings in the code.

Initialize your variables

If variables require initial values before being used, this is the place to assign them.

```
/***** INITIALIZE THE VARIABLES *****/
$half = $cubeSize/2;
```

Main body of the script

The main body or your script is where you use MEL commands and expressions to create objects and determine their attribute settings.

```
/***** MAIN BODY *****/

// Make a cube to visualize the volume.
$cubeName =`polyCube -w $cubeSize -h $cubeSize -d $cubeSize`;

// Move the cube so that its corner lies at the world origin.
move $half $half $half $cubeName[0];

// Create a new Lambert shader.
$cubeShaderName = `shadingNode -asShader -shared lambert -name
  cubeShader`;

// Set the color to white and make the shader transparent.
setAttr ($cubeShaderName + ".color") 1 1 1;
setAttr ($cubeShaderName + ".transparency") 0.7 0.7 0.7;

// Assign the shader to the cube.
select $cubeName[0];
hyperShade -assign cubeShader;

// Make and position the spheres.
for ($i = 0; $i < $count; $i++) { // Loop once for every sphere.
```

```
        // Make the sphere.
        $sphereName = `polySphere -r $radius`;

        // Get a random value within the boundaries of the cube.
        $x =`rand $radius ($cubeSize - $radius)`;
        $y =`rand $radius ($cubeSize - $radius)`;
        $z =`rand $radius ($cubeSize - $radius)`;

        // Position the sphere.
        move $x $y $z $sphereName[0];
    }

} // End procedure.
```

Source your MEL script

Before you can use your procedure you must first declare it. As we mentioned previously, there are several ways to do this. Here, you'll take advantage of Maya's automatic search capabilities and simply call the procedure using its name and arguments. Since the procedure hasn't been declared and therefore doesn't exist in memory, Maya will scan its search path contents for a file of the same root name as the procedure: makeSpheres (the file extension .mel is optional).

1. **Start Maya. If Maya was already running when you started composing your script file, run the rehash command in the Command Line to refresh the search path contents:**

   ```
   rehash;
   ```

2. **Open the Script Editor and enter the following code:**

   ```
   makeSpheres(20, 1, 10);
   ```

Within a second or two you should see a cube and 20 spheres appear in your scene view (Figure 12.11). If this doesn't happen, chances are you either have errors in your code and need to debug it, or Maya was unable to locate your MEL script file on its search path. In the latter case, you will see the following message in the Command Line:

```
Error: line 1: Cannot find procedure "makeSpheres".
```

In this case, double-check that your file is named correctly (it's makeSpheres.mel) and that it is located in Maya's scripts directory, the path that you can query with the internalVar command as demonstrated on page 269. Alternately, you may need to refresh Maya's search path using the rehash command.

Once your script does execute, there will likely be some overlapping/intersecting of spheres. This is due to their random placement and the fact that the procedure has no contingency for intersections. On the CD-ROM we've included a version of the makeSpheres() procedure called makeSpheresAvoid() which includes a simple collision avoidance algorithm to space spheres apart from one another. We will explore similar approaches to collision avoidance in subsequent chapters.

12_MEL_Scripting/MEL/makeSpheresAvoid.mel

Debugging your scripts

Finding and correcting errors is an almost unavoidable part of writing computer code, and MEL scripting is no exception. Rarely will you create a script and enter it in Maya

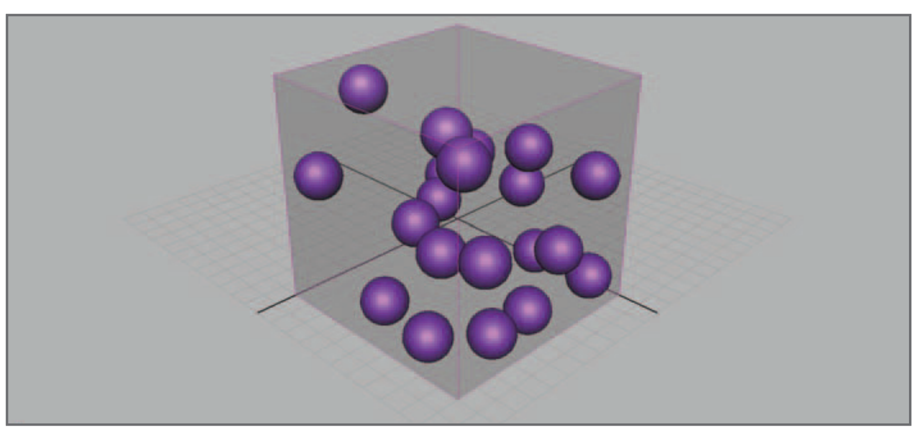


FIGURE 12.11

The MEL procedure in Tutorial 12.01 makes spheres and distributes them randomly in a cube.

without some complaint from Maya about incorrect syntax, such as an undeclared variable or an incorrect use of a MEL command. Syntax errors are relatively easy to debug when compared to logic errors. Logic errors are problems with the way your script executes that are not detected by Maya and can lead to incorrect results; they are often tricky to detect.

Syntax errors

Syntax error messages appear in the Command Line and Script Editor. Line and column numbers point you to the error location in the script. For example, the following error was caused by a mis-typed variable name, `$X` instead of `$x`.

```
// Error: move $X $y $z $sphereName[0];
 //
// Error: Line 62.12: "$X" is an undeclared variable. //
```

The mistake can be found and corrected in the original MEL script file using the line number **62** and column number **12** provided in the error message. To display line numbers in error messages:

In the Script Editor, choose History → Line numbers in errors.

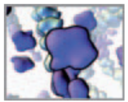
Below are some common syntax errors to watch out for:

- **typographer's quotation marks (" ") used instead of straight marks (" ")** (can occur when sourcing code from an external text editor)
- **within an expression statement, quotation marks have not been escaped using the back slash character (\").**
- **unterminated MEL statement (;)**
- **missing $ at the start of a variable name**
- **code comments not preceded with // or surround by /* and */**

Maya supports third-party **debugger** software to debug plug-ins that you write but has no built-in mechanism for debugging MEL scripts; they must be debugged manually by correcting syntax errors and analyzing the outcome of logical statements.

Logic errors

Logic errors can be dealt with in two ways. The first, and quickest, is to compare your script, line for line, with the corresponding file we've included on the CD-ROM, and



look for discrepancies. The second, and more informative, is to print to the Script Editor, variables from the part of the script that you suspect is causing the trouble. This will help you to see the difference between what you *think* Maya is doing and what is actually happening. You do this using the `print` command. For example:

```
for ($i = 0; $i < $count; $i++) { // Evaluate once for every sphere:
    $sphereName =`polySphere -r $radius`; // Make the sphere.
    print ("making a sphere called: " + $sphereName[0] + "\n");
    etc…
}
```

The example above reports the variable `$sphereName` in the Script Editor, for each sphere created. Tracing values in this way can help locate mistakes in variable assignment that may be causing logic errors. For more information on managing errors in Maya, refer to the Help Library.

Error handling in Maya

Maya Help → General → MEL and Expressions → Debugging, optimizing, and troubleshooting → MEL debugging features

In fact there is a whole field within the subject of computational methods devoted to the invention and study of pseudorandom number generation and quality testing. If you are interested in exploring the topic more fully as you work through this book (we will often invoke MEL's pseudorandom number facilities), a superb starting point is the chapter on random numbers in the famous text *Numerical Recipes* by William Press and his colleagues. As we go to press, this classic has just been released in its Third Edition (2007). Please see our *Further reading* section at the back of this book for further details. (Long at home in theoretical astrophysics at Harvard, Bill Press currently holds the Warren J. and Viola M. Raymer Chair in Computer Sciences and Integrative Biology at the University of Texas at Austin.)

Random number generation in Maya

By now you've seen Maya's rand() function used several times. Here we'll elaborate on random number generation and the relevant MEL functions. Essential to modeling events and processes that depend on probability are methods that simulate uncertainly within the otherwise deterministic working of the computer's digital circuitry. Although various ways of doing this have found favor over the years—such as storing tables of numbers drawn from unpredictable (random) natural events like flips of a coin or rolls of dice—modern techniques use a remarkable discovery: once suitably processed, certain successive permutations, multiplications, and rearrangements of the numbers stored in a computer's CPU register can mimic—often with high fidelity—successive draws from a random process. Because the number stream is not completely and exactly random, but rather a deterministic mimic of a random number sequence, it is conventional to call such computer-generated digits **pseudorandom numbers**. Eventually, a pseudorandom sequence will unmask its determined nature by repeating itself, again and again as the number of calls to it exceeds its period. The initial digit used by the computer, to start its process of giving you back a pseudorandom number each time you ask for one, is called quite naturally the "seed". An important feature of pseudorandom number generators is repeatability—the ability to generate the same stream of numbers over and over again for a given seed. Change the seed, and a different—but repeatable—number stream is generated.

Modern programming languages offer commands letting you mimic the act of picking random numbers from specific probability distributions. Very popular is the use of the uniform probability distribution on the unit interval. This command mimics the act of drawing a number between zero and one with equal likelihood you will fetch back a value anywhere between the bottom value of zero and the top value of

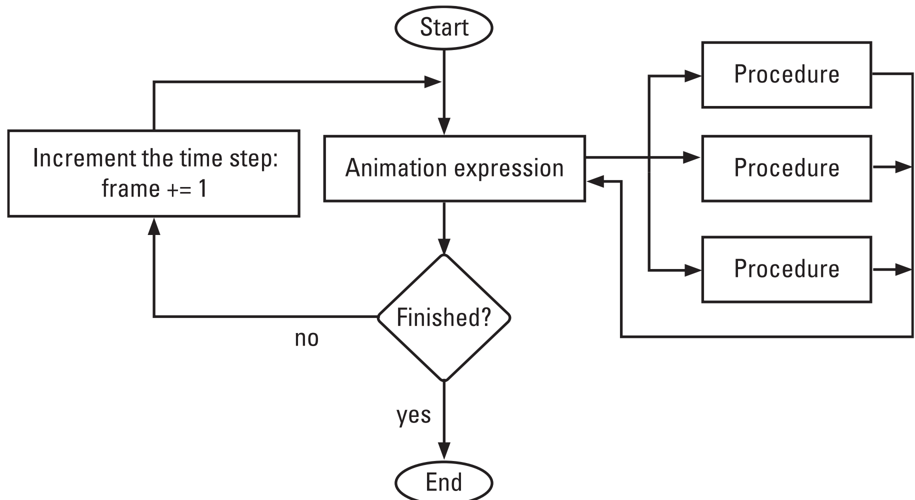


FIGURE 12.12

Flowchart of a typical in silico model combining an animation expression and procedures. You'll put this design into practice in *Chapters 15, 17, and 18*.

one. In Maya, this is accomplished with the rand() command. Other pseudorandom number functions in Maya include sphrand(), noise(), and gauss() which you'll use in *Chapter 15* to pick numbers from a Gaussian distribution in order to simulate molecular diffusion.

Summary

Just about anything you can do via the Maya UI can be done using coded MEL you write. A MEL script can take the form of single line or multiple lines of code that execute when entered or sourced through the Script Editor. Procedures are scripts that execute when called by name and are useful for organizing scripting tasks. They are typically stored in text files external to your Maya scene files. Animation expressions are scripts that are embedded in scene files and execute in relation to the timeline—often once every time the frame number changes. Figure 12.12 illustrates how an animation expression and procedures work together in a typical in silico biology model like the ones you'll explore later in this book, beginning with *Chapter 15*.

In this chapter we explored the basics of programming in Maya—syntax, variables, MEL commands, expressions and operators, conditional statements and loops—and how they fit together to make a MEL statement or a script. Tutorial 12.01 demonstrated a simple approach to quickly populating a scene with multiple objects using a procedure—a hint of what's to come in our explorations of 3D in silico molecules and cells in *Part 3* of this book.

An oft-cited strength of MEL is the ability it gives users to construct custom user interface elements. While we don't have space here to elaborate on this topic, the Maya Help Library is a good source of information on this topic of creating interfaces using MEL. As well, we've listed literary references under *Building Custom UIs* in the *Further reading* section at the end of this book.

Creating custom UIs with MEL

Maya Help → General → MEL and Expressions → Creating interfaces

In the next and final chapter of *Part 2*, you'll build on your MEL scripting skills and knowledge as you learn methods for writing and reading attribute values to and from external data files.

References

1. Janczyn J (Senior Product Manager, Autodesk, Inc.): *Personal interview with Jason Sharpe*, Toronto, ON, September 29, 2006.

2. Taylor M (Developer, Autodesk, Inc.): *Personal interview with Jason Sharpe*, Toronto, ON, September 27, 2006.

13 Data input/output

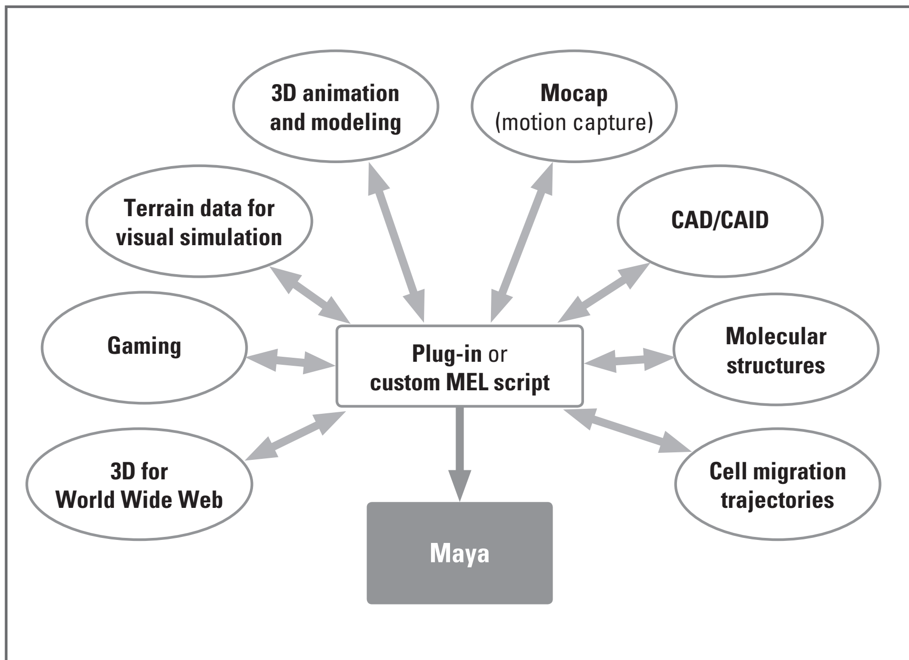


FIGURE 13.01

The many uses for Maya call for importing and exporting different data types, some of which are represented here.

Introduction

This chapter deals with moving alphanumerical data into and out of your Maya scenes. The data itself can take many forms (Figure 13.01). Three-Dimensional modelers and animators, for instance, routinely work with polygon vertex coordinates and with motion capture (or **Mocap**) data imported from various software packages. A research scientist, on the other hand, may deal with chemical concentrations, measurements of force, material properties, or, in the case of our own work, parameters of cellular and biomolecular behavior. If you're working with a widely used 3D data format, chances are that an import/export plug-in exists that will make your job easier. The next section introduces the plug-ins that come bundled with Maya. Still others are available through third-party suppliers. If, however, your data requirements are not served by a ready-made plug-in, you may have to code your own. For such a problem, advanced Maya programmers will typically code plug-ins through Maya's C++ API. However, for novice and intermediate users, MEL offers some handy commands that we'll show you how to use in order to read and write external data files.

In this chapter's tutorial you'll import and visualize trajectory data recorded for live mobile cells. In the second tutorial, you will generate a textual report summarizing key parameters of the cell's motion.

Set up your scene

To work through the examples and tutorial in this chapter, you'll need to have Maya running and the Project directory set up.

1. **Start Maya. If Maya is already running, start a new scene file.**
2. **From the main menu bar, choose File → Project → New.**
3. **Enter DataInOut_Project in the Name field.**
4. **For Location, browse to your projects directory or another location on your hard drive where you'd like to save this project.**
5. **Enter scenes, images, MEL, and FBX in the appropriate text fields.** FBX is a file type you'll work with in the next example.

FBX is an open-standard, platform-independent 3D file format. FBX files can be shared (via plug-ins) amongst users of Maya, Autodesk 3ds Max, Autodesk VIZ, and Autodesk MotionBuilder.

Note the headings *Project Data Locations* and *Data Transfer Locations*, under which are listed the standard file types used in Maya workflows. There is no field for "custom" data files, so you'll need to create a custom data directory outside of Maya.

6. **Press Accept to create the project and close the New Project window.**
7. **Leave Maya and navigate in Windows to the DataInOut_Project directory you created above then make a new directory (folder) called customData. The directory path should look something like the following:**

```
...\My Documents\maya\projects\DataInOut_Project\customData
```

Return to Maya and set the preferences for your scene.

8. **Choose Window → Settings/Preferences → Preferences.**
9. **Choose Categories → Settings and make the following settings:**

   Under Working Units → Linear: centimeter.

   → Time: NTSC.

Any settings that aren't specified above can be left at their default values for now.

10. **Press the Save button to set your preferences.**
11. **Select the Perspective view of your scene by pressing the button in the Layouts panel at the bottom-left of the main window.**

Translators

At the time this book went to press, Maya 2008 shipped with **translators** for importing and exporting many widely used 3D data formats. In order to allow Maya to start

The plug-in file extension in Maya for Windows is .mll (short for **m**aya **l**ink **l**ibrary).

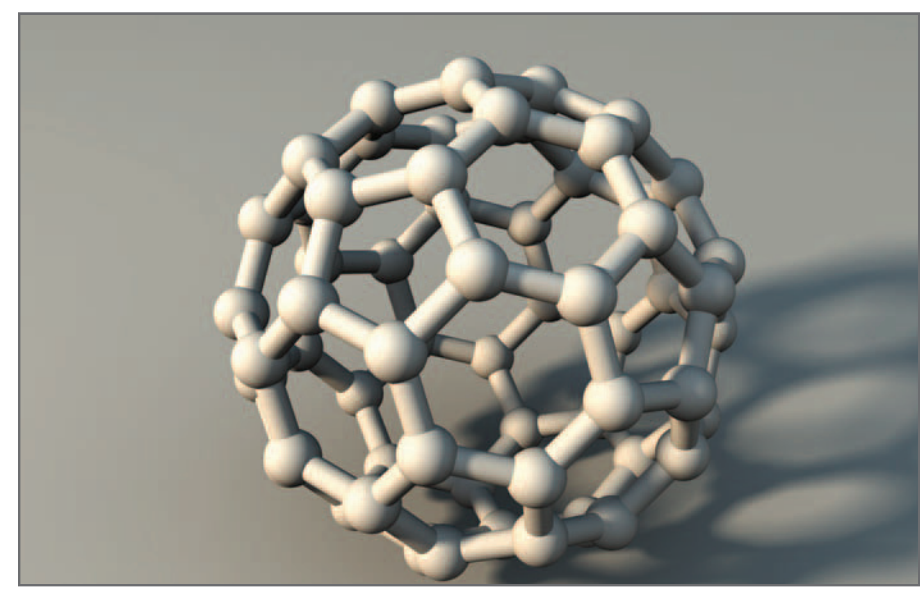


FIGURE 13.02

Plug-ins called translators and exporters let you import and export models in formats other than Maya's native one. This model of a carbon buckyBall was saved in FBX format then imported into Maya via the **fbxmaya.mll** plug-in.

quickly and to save memory, many of these plug-ins are not loaded by default when you launch Maya and must be loaded via the Plug-in Manager prior to use. We have included a sample FBX file on the CD-ROM which you'll import into Maya in the following example.

A buckminsterfullerene (sometimes referred to as a buckyBall) is a carbon molecule composed of 60 atoms (thus its chemical name C60), arranged in a spherical conformation. They were discovered in the mid-1980s by Harold Kroto, Robert Curl, and Richard Smalley, for which they won the Nobel Prize in Chemistry in 1996. They resemble the geodesic domes of famed inventor Richard Buckminster Fuller, for whom they were named. They have been investigated for their potential use in a number of medical applications, including the encapsulation and delivery of specialized antimicrobial agents.

File translators

Maya Help → Using Maya → Translators and Exporters

The model you'll import is of a buckyBall (Figure 13.02)—a carbon nanostructure composed of carbon and named for architect Buckminster Fuller.

1. **Copy the following file from the CD-ROM to the customData directory that you created inside your Maya Project directory.**

   13_DataInOut/FBX/buckyBall.fbx

2. **Start Maya. If Maya is already running, start a new scene file.**
3. **Choose File → Import. This launches the Import window.**
4. **Navigate to your customData directory, select buckyBall.fbx and press the Import button.**

If you get an error message stating: "`Unrecognized File Type`" or "`Error reading file`" it's because the FBX translator plug-in was not automatically loaded when you start Maya. You can load it using the Plug-in Manager, as follows:

1. **Choose → Window → Settings/Preferences → Plug-in Manager.**

At the top of the Plug-in Manager you'll see the plug-ins directory path—the location to place any plug-ins you wish to add to Maya. Below the path name is a list of about

MEL command	Sample use	Result
fopen	**$fileID = `fopen` $fileName "r"`;**	Opens the file, $fileName, for reading.
fclose	**fclose $fileID;**	Closes the file.
feof	**feof $fileID;**	Returns **1** if the end of the file has been reached, and **0** otherwise.
fgetline	**string $line = `fgetline $fileID`;**	Returns the next line of $fileName, then increments the line pointer.
fgetword	**string $word = `fgetline $fileID`;**	Returns the next word of $fileName, then increments the word pointer.
fread	**$char = `fread $fileID $str`;**	Returns the next set of bytes until either a null character or the end of the file is reached. Its type is specified by its second argument, a dummy variable.
frewind	**frewind $fileID;**	Returns the reading pointer to the start of the file.
fprint	**fprint $fileID "Hello World";**	Prints the argument to the file specified by $fileID.
fwrite	**fwrite $fileID "Hello World";**	Prints the argument to the file specified by $fileID. Null characters are added to the ends of new lines.
fflush	**fflush $fileID;**	The results of fprint and fwrite are not immediately written to the file, but are stored in a software buffer. fflush flushes the data to the file and clears the buffer.

TABLE 13.01

MEL commands used for reading and writing external data files.

50 plug-ins. Only those with “Loaded” or “Auto load” checked are current available to your scene. When Auto load is checked, the plug-in will be loaded automatically when you start Maya.

2. **Check the box next to fbxmaya.mll.**
3. **Press the Close button.**

Repeat steps 3 and 4 above to import buckyBall.fbx now that the plug-in is loaded.

Reading and writing files with MEL

Table 13.01 lists the MEL commands designated for reading and writing external data files. To help you try out these commands, we’ve included a sample text file on the CD-ROM which you’ll reference in this section’s examples. Start by copying it from the CD-ROM to your new customData directory.

1. **Start a new scene in Maya.**
2. **Copy the following file from the CD-ROM to your DataInOut_Project/customData directory.**

13_DataInOut/customData/helloWorld.txt

The file path

In order to read from and write to a file you must first be able to locate it on your hard drive from within Maya. You can do so using Maya's workspace command and the relative path name, customData. Maya uses the forward slash character to separate directories.

```
string $fileName = `workspace -q -fullName` + "/customData/
                    helloWorld.txt";
```

When working in Windows, you can use the back slash character (Windows NT notation) to separate directories. However, you must remember to escape each back slash so that it will remain part of the string. For example:

```
// INCORRECT (not escaped):
string $fileName = `workspace -q -fullName` + "\customData\
                    helloWorld.txt";
// Result: C:/ ... DataInOut_ProjectcustomDatahelloWorld.txt //

// CORRECT (escaped):
string $fileName = `workspace -q -fullName` + "\\customData\\
                    helloWorld.txt";
// Result: C:/ ... DataInOut_Project\customData\helloWorld.txt //
```

filetest command

Maya's filetest command is used to query information about a file. The -r flag tests if the file exists and is readable. filetest can be used together with the error command to halt a script and warn the user if a file cannot be located.

```
if (!`filetest -r $fileName`) error "No such file exists" ;
else print "File located successfully." ;
```

Opening and closing the file

Before reading or writing a file, you must open it for Maya with the fopen command which takes two arguments. The first is the file name and path. The second tells Maya whether the file is to be opened in read (r), write (w), or append (a) mode. Adding + to the mode letter (r+ or w+) opens the file for both reading and writing. fopen returns a file handle (identification number) which is used subsequently to refer to the file by number when using any of the file read/write commands. When you open a file in write or read/write mode, any data you write to the file from within Maya overwrites the file's previous data. Use the append mode if you wish to add to a file without deleting its existing contents.

```
// Open the file in read and write mode.
int $fileID = `fopen $fileName "r+"` ;
```

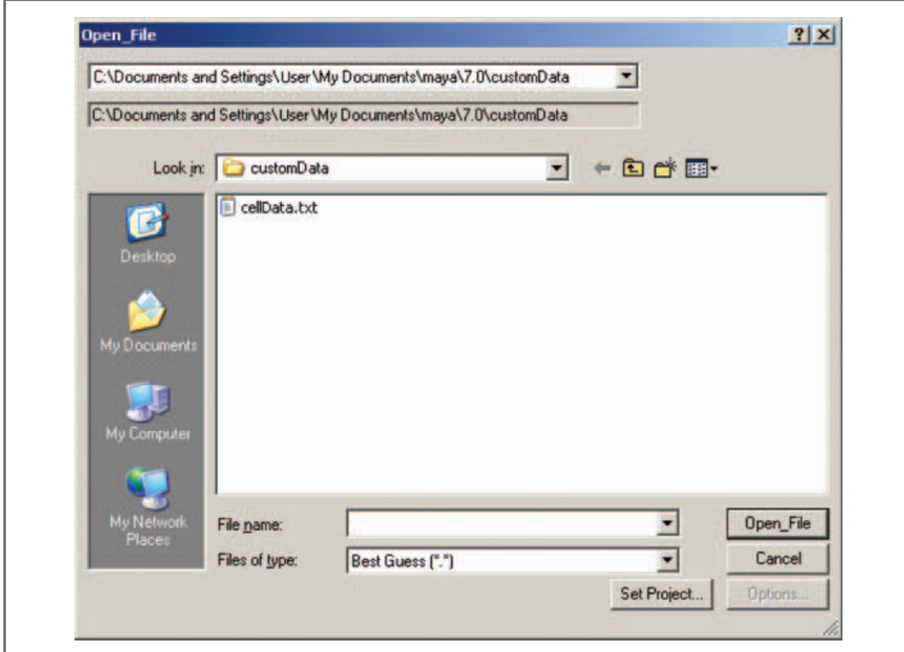


FIGURE 13.03

The file browser window launched by Maya's **filebrowserdialog** command.

If no file by the specified name exists, `fopen` creates a new one. When you're done reading or writing the file you must close it using the `fclose` command, as follows:

```
fclose $fileID;
```

The file browser

When you can't be sure of the directory in which a file resides, you can use the `fileBrowserDialog` command to launch a browser window (Figure 13.03). The user can then navigate to the desired file, select, and load it at the push of a button. `fileBrowserDialog` executes a command (or procedure) defined by the `-fileCommand` flag and sends that command into two arguments: the file name and type. Below, we've defined a custom procedure named `fopenFile` that will be called by `fileBrowserDialog`. `fopenFile` is declared with two arguments: `$fileName` and `$type`, corresponding to the arguments to be sent by `fileBrowserDialog`. The second argument, `$type`, is not used by `fopenFile` but must be declared nonetheless since `fileBrowserDialog` returns two arguments. The variable `$fileID` has been defined globally so that it will be accessible outside of `fopenFile`. To try out the following code, enter it in the Script Editor in the order it appears below.

```
global proc fopenFile(string $fileName, string $type) {
    global int $fileID;
    // Open the file for reading.
    $fileID = `fopen $fileName "r"`;
}

fileBrowserDialog -mode 0 -fileCommand "fopenFile" -fileType ""
  -actionName "Open_File" ;
```

For the -mode flag, a value of 0 opens the browser window in *read* mode. There is no "text" or "txt" file type defined for this command so we've left the -fileType argument empty (""). This will invoke the default "Best Guess" option in the "Files of type" menu of the browser window. The argument passed to -actionName is the text that appears on the browser "action" button (e.g. *Open*, *Save*, and so on). For a complete list of flags for the fileBrowserDialog command, refer to the MEL command reference in Maya's Help Library.

Reading data

You can read data one textual line, word, or character at a time using the MEL commands fgetline, fgetword, and fread, respectively. For the examples below to work properly, you must first complete the examples in the previous section which open and prepare for reading the file helloWorld.txt.

The fgetline and frewind commands

```
// Use fgetline to get each line of text in helloWorld.txt.
// When the file end is reached, fgetline will assign the integer 0
  to $line;

string $line; $line = `fgetline $fileID`;
while (`size $line` > 0) {
    $line = `fgetline $fileID`;
    print $line;
}
```

The printed result:

```
"The scientific study of the living cell brings countless
opportunities to apply computing and visualization to crucial
problems, to map the cell's molecular world and develop new
treatments for disease."
```

When referring to strings, a **null** value corresponds to the absence of content, which is sometimes denoted as an empty string: "". A null string value is not the same as the integer zero. However, if you were to assign an integer variable the value of a null string, Maya would convert the null string value to a zero integer value. In the example below, $str is automatically assigned a null value when declared. Upon assigning its value to $int, Maya converts *null* to zero.

```
string $str;
int $int = $str;
// Result: 0//
```

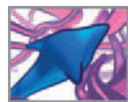
Each time fgetline executes, Maya advances to the next new line in the text file. Therefore, if you were to run the above 6 lines of code again nothing would print. Instead, $line would be assigned null values since there are no lines 4, 5, and 6 in the text file. To return to the start of the text file use the frewind command, as follows:

```
int $i;
for ($i = 0; $i < 3; $i++) {
   frewind $fileID;
   $line = `fgetline $fileID`; print $line;
}
```

The printed result:

```
"The scientific study of the living cell brings countless
  opportunities to
"The scientific study of the living cell brings countless
  opportunities to
"The scientific study of the living cell brings countless
  opportunities to
```

The above result is obviously not very useful but it illustrates the function of frewind well.



The fgetword command

`fgetword` reads a string of characters separated from other strings by blank spaces or tabs. Each time `fgetword` executes, Maya advances to the next word in the file.

```
// Use fgetword read the first six words in helloWorld.txt.
frewind $fileID;
string $word;
for ($i = 0; $i < 6; $i++) {
   $word = `fgetword $fileID`;
   print ($word + "\n");
}
```

The printed result is the first six words of helloWorld.txt:

```
"The
scientific
study
of
the
living
```

You can specify a custom separator by passing `fgetword` a second argument. In the following example, "words" (strings of characters) are not separated according to blank spaces but rather by the presence of a "new line" character `"\n"`.

```
frewind $fileID;
for ($i = 0; $i < 3; $i++) {
   $word = `fgetword $fileID "\n"`;
   print $word;
}
```

The result is that each line is treated as a single word:

```
"The scientific study of the living cell brings countless
opportunities to apply computing and visualization to crucial
problems, to map the cell's molecular world and develop new
treatments for disease."
```

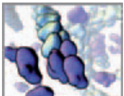
The fread command

`fread` reads an entire set of bytes (to a maximum of 1024 on strings) until either a null character or the end of the file is reached. `fread` casts the data to the type specified by a dummy variable (int, float, string, or vector). The dummy variable is `$tmpStr` in the following example.

```
frewind $fileID;
string $tmpStr, $textChars;
$textChars = `fread $fileID $tmpStr`;
```

Since your file helloworld.txt contains no null characters and is less than 1024 bytes long, `fread` reads the entire text string and returns the following result in the Script Editor:

```
"The scientific study of the living cell brings countless
opportunities to apply computing and visualization to crucial
problems, to map the cell's molecular world and develop new
treatments for disease."
```



The feof command

feof is short for "file end-of-file". It returns a value 1 if the end of the data file has been reached, and a value of 0 otherwise. In the following example, each word in the text file is read and stored in an array variable.

```
int $i = 0;
string $wordArray[];
// Clear the variable to a null value.
clear $wordArray;
frewind $fileID;

while (!`feof $fileID`) {
    $wordArray[$i] = `fgetword $fileID`;
    $i++;
}
print $wordArray;
```

The following gets printed to the Script Editor:

```
"The
scientific
study
of
the
living
cell
etc...
```

To make convenient examples, we've focused on textual instead of numerical data so far. However, you'll see shortly how numbers are handled by the same set of MEL commands.

Writing data

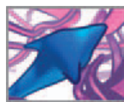
To write data from Maya into an external text file, you'll use either of the fprint or fwrite commands. fprint writes its argument as a string to the file, whereas fwrite does so as binary data, terminating strings with null characters.

fprint

In this example, you'll create a new file in and write text to it. You'll save the new file in your customData directory.

```
string $fileName = `workspace -q -fullName` + "/customData/
                    myDataOut.txt";
// Open the file in write mode.
int $fileID = `fopen $fileName "w"`;
string $content = "For all their mystery and beauty,";
fprint $fileID $content;
fclose $fileID;
```

The workspace command with the fullName flag returns your current projects directory (DataInOut_Project/) to which you added customData/ and the file name, myDataOut.txt. Leave Maya and browse to your customData directory. Open your new



file, myDataOut.txt, in a text editor. You should see the text: "For all their mystery and beauty,"

fwrite

Here you'll open myDataOut.txt in *append* mode and add a couple lines of text to it.

```
// Open the file in write mode.
int $fileID =`fopen $fileName "a"`;
string $content = "\nthe body's cells do not live or work in
                   isolation.";
fwrite $fileID $content;
string $content = "\nEach organ and tissue is an intricate society
                   of cellular specialists.";
fwrite $fileID $content;
// Close the file.
fclose $fileID;
```

Enter the above script in the Script Editor, then open the appended file, myDataOut.txt, in your text editor. Note the null characters that fwrite added to the end of the new lines. Different text editors represent null characters in different ways. You may see something like:

```
For all their mystery and beauty,
the body's cells do not live or work in isolation.□
Each organ and tissue is an intricate society of cellular
  specialists.□
```

When you read the file back into Maya using fgetline or fgetword, the null characters get dropped. In the case of fread, only the characters up to each null character are read each time fread is called. In the following example, fread reads up to the first null character:

```
// Open the file for reading.
int $fileID =`fopen $fileName "r"`;
string $tmpStr; string $textChars;
$textChars =`fread $fileID $tmpStr`; // $tmpStr is a dummy variable.
```

The fflush command

The commands fprint and fwrite store data in a software buffer. They don't actually write the data to your hard drive until you close the file with the fclose command. You can, however, force the data to be written while the file is still open by flushing the buffer with the fflush command, as in the following example:

```
string $fileName =`workspace -q -fullName` + "/customData/
                    myDataOut.txt";
// Open the file in write mode. The existing data will be
  overwritten.
int $fileID =`fopen $fileName "w"`;
string $content = "For all their mystery and beauty,";
fprint $fileID $content;
fflush $fileID;
```

Now browse to your Maya Scripts directory and open myDataOut.txt in your text editor application. It should contain the single line, "For all their mystery and beauty," which was written to the file when you flushed the software buffer using fflush. Close the file in your text editor. With the file still open in Maya, add two more lines of text to it:

```
string $content = "\nthe body's cells do not live or work in
                   isolation.";
$content += "\nEach organ and tissue is an intricate society of
             cellular specialists.";
fprint $fileID $content;
fclose $fileID; // Close the file.
```

You can view the updated contents of myDataOut.txt by reading the file back into Maya.

```
int $fileID =`fopen $fileName "r"`; // Open the file in read mode.
string $tmpStr;
print ("\n" +`fread $fileID $tmpStr`);
fclose $fileID; // Close the file.
```

That's it for the commands used to read and write external files from within Maya. In the following tutorial you'll bring several of the techniques and commands you've learned to bear on a case study involving data for living cells.

Reading and writing files

Maya Help → Using Maya → General → MEL and Expressions → I/O and interaction → Reading and writing files

Tutorial 13.01: Visualizing cell migration

In this tutorial you will build a short MEL script to read in position data for live, migrating cells and use it to visualize the their trajectories (Figure 13.03). The script will also analyze the data and prepares a summary report of key migration statistics.

The cell migration data file

The data—recorded for **CD4 lymphocytes** (white blood cells)—was generously provided by Prof. Peter Friedl of the Rudolf-Virchow Center, University of Würzburg, Germany. Dr. Friedl is a pioneer in the study of cancer cell migration in 3D environments (you'll explore 3D cell migration in *Chapter 18*). The data—for six cells undergoing planar (2D) motion—is saved in a tab-delimited file on the CD-ROM. Other common formats for numerical data include space- and comma-delimited (CSV), which can be interpreted by spreadsheet applications such as Microsoft Excel and Corel Quattro Pro, and by mathematics and statistics programs like MATLAB (The MathWorks). Below is an excerpt from the file you'll use in this tutorial, cellData.txt.

Time	Cell0		Cell1		Cell2		Cell3		Cell4	
	x	z	x	z	x	z	x	z	x	z
1	400	268	335	384	592	406	483	184	326	501
2	400.89	268.1	335.64	384.64	588	405	483	184	328	499
3	401.29	267.7	335.79	384.79	587	405	483	184	335	498
4	401.7	267.29	335.95	384.95	585.5	406.83	483	184	338.5	499
5	402.1	266.89	336.1	385.1	583.5	408.16	483	184	341	499
6	402	266	336.25	385.25	581	409	482	187	346	501
7 etc.	402	266	336.39	385.39	578	412	481	184	352	499.5

TABLE 13.02

An excerpt from the cellData .txt file. To produce their cell migration data, Peter Friedl and colleagues used a special videomicroscopy apparatus to make timelapse videos of cells. They then subjected the video to analysis by cell tracking software to produce the data featured in this tutorial.

Cell trajectory data provided by Peter Friedl, Rudolf-Virchow Center, University of Würzburg, Germany. Used with permission.

The first two rows contain labels for each of the data columns. The first column lists time increments of 40 seconds each. The remaining columns list X and Z coordinates, in video pixel units, for each of four cells. The file you'll use contains positions for six cells over a period of 230 time increments. Copy the file from the CD-ROM to your hard drive:

Copy the following file from the CD-ROM to your DataInOut_Project/customData directory. If you haven't created this directory, do so now.

13_DataInOut/customData/cellData.txt

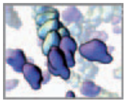
Spatial and temporal scales

In the XZ coordinate data, each pixel corresponds to 1.4 micrometers (µm for short). For simplicity, let's treat 1 Maya unit as being equal to 1 µm. Therefore each X and Z position value stored in `cellData.txt` will be multiplied by 1.4 to give an equivalent value in Maya units.

Each time step in `cellData.txt` represents 40 seconds of real cell migration time. Therefore, a 230 frame animation that shows the total displacement of the six cells in fact represents ($230 \times 40 \div 60 \approx$) 153 minutes of real time. Furthermore, if the animation plays back at 30 fps, one second of viewing time corresponds to ($153 \div 30$) $\approx$ 5 minutes of real time.

Visualizing the data

Visualizing the data in Maya using 3D objects in place of living cells gives you a time-lapse view of the migration dynamics. You'll represent each cell with a NURBS sphere of diameter = 10 Maya units (or 10 µm—the approximate size of a lymphocyte). You'll animate their positions, using an animation expression called `moveCells`, to match the migration data as your scene plays from frame 1 to 230. Furthermore, rather than read the position data in at each frame, you'll read it in only once and



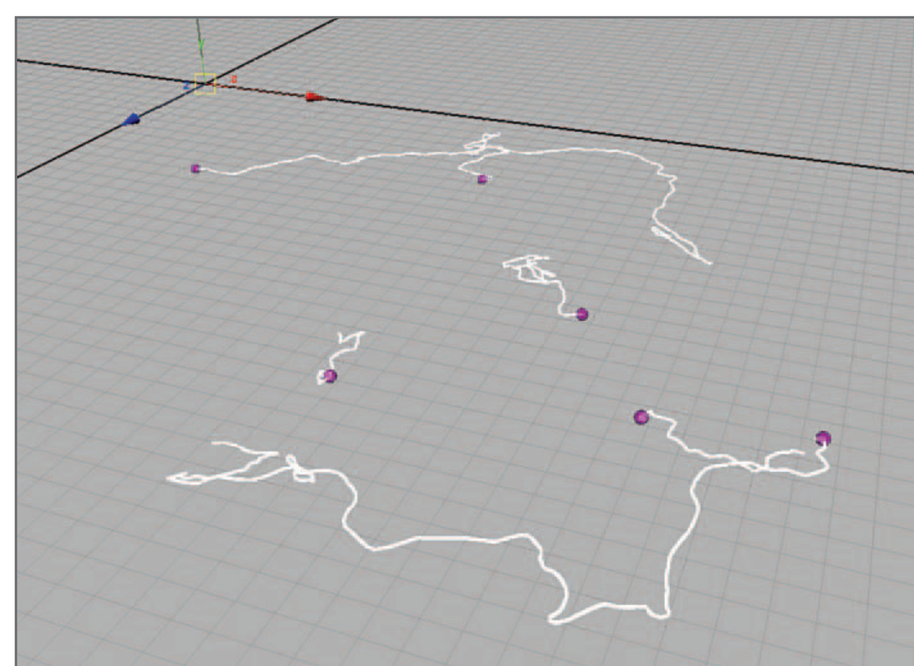


FIGURE 13.04

A rendering of cell (purple spheres) positions at frame 230 (the final time step). Spline curves are used to trace trajectory paths and are made visible in the rendering by Maya Paint Effects strokes (in white).

Cell trajectory data provided by Peter Friedl, Rudolf-Virchow Center, University of Würzburg, Germany. Used with permission.

store it in a global array variable. Updating the positions of your cell models then becomes a matter of querying this array and setting the translate attribute values.

In addition to moving your cell models, your script will trace their trajectories using NURBS curves and assign Maya Paint Effects strokes to the curves so that they can be rendered along with the cells. Figure 13.04 shows the scene view at the end of an animation run.

Plan your visualization algorithm

In order to get clear picture of what you require make this visualization happen, let's look at a flowchart—or logical diagram—of the steps to be taken in Maya (Figure 13.05). From the flowchart, you will build an algorithm in the form of a MEL script. Flowcharts are often used to plan and illustrate the logical flow of a computer program. If you're unfamiliar with flowchart conventions refer, to the legend in Figure 13.06.

Plan your summary report

During animation playback, you'll calculate the cell migration parameters, net displacement (Dnet), the total distance traveled (L), and the directedness coefficient (D_c):

1. **Directedness coefficient, D_c = Dnet/L**

When the animation has finished (i.e. at frame 230) you'll write D_C to a summary text file, using the layout shown in Figure 13.08. As we've written the code here, the summary data will be tab-delimited. However, tab characters can easily be replaced by commas or blank spaces if either format is better suited to an established spreadsheet workflow you're using.

Start on ***press Play***

frame = 1 ?

no

yes

- Read cellData.txt and store the data in a matrix
- Prepare the NURBS spheres (cells) and curves (cell trajectories)
- Clear the statistics variables for **L** and **Dnet**

- Move the cells
- Add a point to the trajectory curve
- Calculate and store **Dnet**

frame = 230 ?

no

(**frame** += 1)

yes

Stop playback

- Calculate **L** and **Dc** for each cell
- Calculate the *average* value for all cells of **Dnet**, **L**, and **Dc**
- Compose a data output string called $dataOut using **Dnet**,**L**, and **Dc**
- Open a new file and fprint **$dataOut** to it.

End

FIGURE 13.05

Flowchart for Tutorial 13.01.

We'll take a closer look at ways in which cell migration is quantified and qualified in *Chapters 16* and *18*. For now it's useful to know that the "directedness" of cells, their tendency to move in a directed rather than random fashion, is a key cell behavior parameter that is implicated in many normal and pathological processes within the body, including embryonic development, tissue regeneration, and cancer.

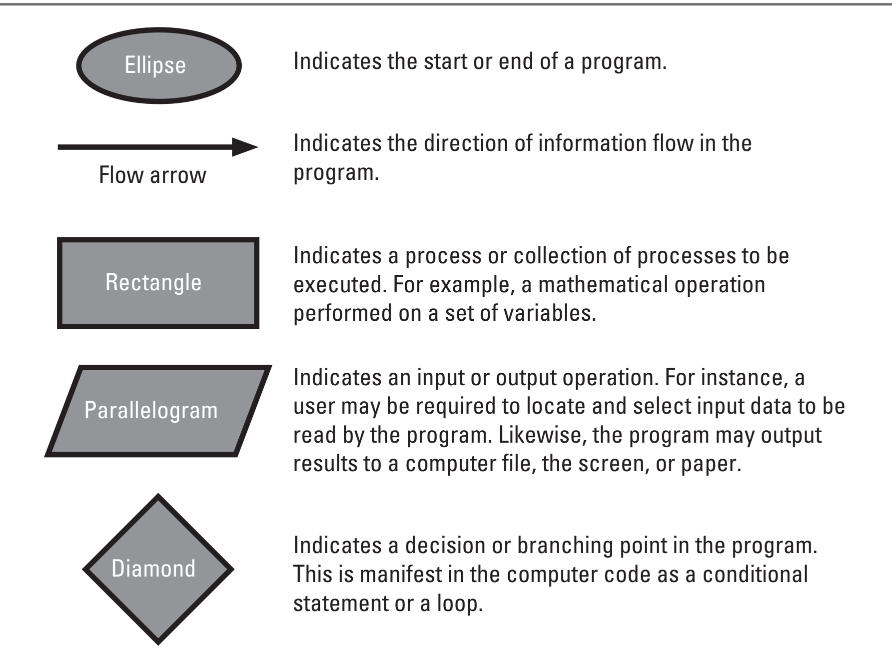


FIGURE 13.06
Conventional shapes used to build flowcharts.

Encoding the algorithm

Now it's time to take the algorithm plan outlined in Figure 13.05 and turn it into a MEL script to make it happen. Below we present the entire `moveCells` expression, interspersed with explanations where necessary. We've included the finished script on the CD-ROM. You'll find it useful for checking your work or for seeing `moveCells` in action right away!

13_DataInOut\MEL\moveCells.txt

Composing the script file

We recommend building your MEL script in a text editor like one of those we suggested on page 302. Ensure that the editor is set to use straight, not curly, quotes. You can type the script in as you follow along with the instructions below. That way you can save it as plain text (.txt) file in your Maya Scripts directory periodically. You can query the path to your Scripts directory in Maya using the `internalVar` command, as follows:

```
internalVar -userScriptDir;
```

In Windows, `internalVar` will return a path like:

```
Result: C:/Documents and Settings/User/My Documents/maya/8.5/
        scripts/
```

Give your file the name `moveCells.txt`. Since it's to be an expression and therefore uses slightly different syntax for assigning attributes, the script won't execute properly if you source it through the Script Editor. Giving your file the extension `.txt` instead of `.mel` will avoid confusion with scripts that are meant to be sourced through the Script Editor or loaded by procedure calls.

When you want to try out bits of code in Maya, simply copy and paste them from your text editor into Maya's Script Editor. When your MEL script is complete you will copy and paste it into Maya's Expression Editor to create the moveCells expression.

moveCells.txt

Let's begin with a short header to document the script:

```
/***** moveCells.txt *****/
/*
Date:
Authors: Donald Ly and Jason Sharpe.

Description:
This expression reads in cell migration data from an external text
file and uses it to move objects called "cell1", "cell2", etc. The
cells are created if they don't already exist in the scene file.
This expression also calculates L, Dnet, and Dc for the cells and
prints the results to a text file.

To use this script:
Start a new Maya scene, then copy and paste this entire script into
Maya's Expression Editor and press the Create button. Locate and
load the data file when prompted by the file browser. Press play to
animate the cells and generate the summary report.
*/
```

Now declare and assign your variables. This is necessary on frame 1 only—doing so on every frame would be redundant and eat up processing cycles—so you'll start with a conditional statement to test the current frame number. You'll be using the `fileBrowserDialog` command and a custom procedure called `fopenFile()` (as previously described on page 317) to load the data file. Therefore, you'll need to make the `$fileID` variable global so that it can be used outside of the procedure. Cell positions will be stored in a matrix called `$cellPos[][]`. Using a matrix allows you to store the data for all cells over the 230 time steps in one variable. However, unlike an array, a matrix cannot change size once it's been declared, nor can its dimensions be set using variables. Therefore you must give `$cellPos[][]` enough elements to store all of the position data: 230 rows and 16 columns (8 cells $\times$ 2 coordinates).

The following comments describe briefly what each of the variables is for.

```
/***** DECLARE THE VARIABLES *****/

/*
$fileName            The name of the open data file, cellData.txt.
                     At frame 230 it will be used to store the name
                     of your summary report file.
$coord               A temporary holder for the X and Z coordinate
                     values.
$line                The result of the fgetline command.
```

```
$name                   The current cell or curve name.
$Lstring                The $L values for writing to the summary file.
dNetString              The $dNet values for the summary file.
$DcString               The Dc values for the summary file.
*/
string $fileName, $coords[], $line, $name, $Lstring, $dNetString,
  $DcString;


/*
$dataOut                A single string containing all of the information
                        to be written to your summary file.
*/
global string $dataOut;


/*
$cellPos[][]            The XZ coordinates of the cells for all time
                        steps.
*/
global matrix $cellPos[230][12];


/*
$fileID                 The data file handle and is assigned by fopen.
$cellCount              The number of cells, 6.
*/
global int $fileID, $cellCount;


/*
$L[]                    The total distance by each cell.
$dNet[]                 The net displacement of each cell.
$Dc[]                   The directedness coefficient of each cell.
*/
global float $L[], $dNet[], $Dc[];


/*
$dNetSum                Used to calculate the average $dNet value.
$Lsum                   Used to calculate the average $Lsum value.
$scale                  The spatial scale of the simulation: 1.4 µm
                        per video pixel.
$x                      A Temporary holder for the cell X coordinate.
$z                      A Temporary holder for the cell Z coordinate.
*/
float $dNetSum, $Lsum, $scale, $x, $z;


/*
$newPos                 The position of the current cell in the current
                        frame.
$oldPos                 The position of the current cell in the
                        previous frame.
                        The above values are used to calculate the
                        distance traveled by the cell in one time
                        step.
*/
vector $newPos, $newPos;
```

```
/*
$i                      A counter for matrix rows.
$j                      A counter for matrix columns.
*/
int $i, $j;
```

Only a few variables need to be assigned up front. The others will be assigned at appropriate points throughout the script. The playbackOptions command is used to set a playback range corresponding to the number of time steps in the data set: 230. Maya must play every frame in order for the expression to execute properly. After that you'll declare the `fopenFile` procedure we demonstrated back on page 317.

```
if (frame == 1) {

/***** INITIALIZE THE VARIABLES *****/
$cellCount = 6;
$scale = 1.4;
$dNetSum = $Lsum = 0;

/***** MAIN BODY *****/

// Set the playback speed (0 = play every frame) and range.
playbackOptions -playbackSpeed 0 -loop once -min 1 -max 230;

// Define a procedure to fopen the data file.
global proc fopenFile(string $mode, string $fileName, string
  $type) {
    global int $fileID;
    $fileID = `fopen $fileName $mode`; // Open the file for reading.

}
```

Read and store the data

Next, you'll read `cellData.txt` line by line using `fgetline` and store the X and Z positions in `$cellPos[][]`. The file will be accessed and read using the combination of `fileBrowserDialog` and the `fopenFile` procedure. In order to break each line into separate strings, you'll use the `tokenizeList` command. `tokenizeList` collects elements (or tokens) of a string that are separated by white spaces or commas, and assigns them to a string array.

```
    /***** LOAD THE DATA *****/

    /* Check if the data has already been loaded. If it has, the
      file ID will not be 0. */
    if (!$fileID) {

        // Open a file browser and send the "r" mode argument to
          fopenFile.
        fileBrowserDialog -m 0 -fc ("fopenFile" + " \"r\" ") -ft ""
          -an "Open_File";
```

It's possible that you or another user will press Escape or close the browser window without selecting a file. In this case, you'll want to halt the expression and print a message in the Script Editor using the `error` command.

```
        // Stop the expression if the user doesn't select a file.
        if ($fileID == 0) {
            error "No file chosen";
        }

        /* Run fgetline twice to increment to the start of the
          numerical data (past two lines of header text). */
        fgetline $fileID;
        fgetline $fileID;

        $i = 0; // The row counter.
        $line =`fgetline $fileID`;

        while (!`feof $fileID`) {// feof returns 1 at the file end.

            // Break $line into tokens and store them in $cellNames.
            tokenizeList($line, $coord);
```

When assigning values to $cellPos[][] using variables to specify the elements, remember that X and Z reside in alternating columns. The column element expressions $j *2 and $j *2+1 for X and Z, respectively, ensure that X values occupy odd numbered columns (1, 3, 5, and so on) and Z values occupy even numbered columns (2, 4, 6, etc.) for every value of $j . Furthermore, because the XZ values in the first row of $coord[][] are all integers (refer to page 323), you'll type cast $coord[] by prefacing it with (float). Without (float), Maya will interpret the data in $coord[][] as integers, based on the numbers in the first row. When those integers are in turn assigned to $cellPos[][] (whose type is not set by the matrix declaration—remember, a matrix is declared on as type "matrix") Maya will set $cellPos[][] to type: integer. After that any values assigned to $cellPos[][] will be automatically converted to integers, which you don't want. The (float) preface avoids this problem by ensuring Maya interprets the numbers as float values.

You'll recall from Chapter 12 that type casting coverts a value to the data type specified in brackets.

```
            for ($j = 0; $j < $cellCount; $j++) {// The column
              counter.
                $cellPos[$i][$j*2] = (float) $coord[$j*2+1] * $scale;
                $cellPos[$i][$j*2+1] = (float) $coord[$j*2+2] * $scale;
            }

            $line = `fgetline $fileID`; // Get the next line of data.
            $i++; // Increment the row counter.

        } // End while loop.

    } // End if (!$fileID) statement.
```

Prepare the geometric models

There are three scenarios to consider when this expression executes at frame 1. In the first, you're running this project for the very first time: no spheres and no curves (which trace the cell trajectories) exist yet. In the second, you've reopened the scene after saving it with the spheres and, possibly, the curves in place. In the third, you've just rewound the timeline and need to reposition the spheres to their starting positions and delete the curves so that new ones can be created the next time you press Play. The following code covers all three scenarios by first deleting existing curves, then checking for existing spheres. If none exist, new ones are created. Next, the

spheres are moved into position according to the coordinates stored in `$cellPos[][]` and new curves are started at the center of each cell. You need only delete the curves once, whereas creating and/or positioning a sphere and starting a new curve must occur six times—once for each of your cells.

```
/***** PREPARE THE SPHERES AND CURVES *****/
// Sphere represent the cells, and the curves, the cell paths.
// Delete existing curves.
string $tmpStr [ ];
$tmpStr = `ls -tr "curve*"`; // Make a list of curves.
```

If the size of the `$tmpStr` array is greater than 0, then curves exist. The following `if` statement checks this condition. If it's true, the curves are deleted.

```
if (`size $tmpStr` > 0) delete $tmpStr; // Delete the curves.
```

Next, the expression deletes Paint Effects strokes and brushes that may exist in the scene from a previous animation run. You'll see how these nodes are created and what they do shortly. To check if strokes or brushes exist in your scene you can simply test the $tmpStr array for a zero or non-zero value with a Boolean *if* statement—rather than explicitly comparing the size of $tmpStr to zero, as you did for the curves above. If $tmpStr is of size zero, the *if* statement returns zero or *false*, and doesn't execute the next command. Conversely, if $tmpStr has a non-zero size, the *if* statement returns *true*, and executes the next command.

```
// Delete existing Paint Effects strokes.
$tmpStr = `ls -tr "stroke*"`; // Make a list of strokes.
if (`size $tmpStr`) delete $tmpStr; // Delete the strokes.

// Delete existing Paint Effects brushes.
$tmpStr = `ls -tr "brush*"`; // Make a list of brushes.
if (`size $tmpStr`) delete $tmpStr; // Delete the brushes.

// Make and/or position the spheres, and make the curves.
for ($i = 0; $i < $cellCount; $i++) {
```

Since the animation starts at frame 1 and the index of the first row of `$cellPos[][]` is "0", you'll need to subtract 1 from Maya's internal `frame` variable in order to it as an index in `$cellPos[][]`. Furthermore, `frame` is of type, float, not int. Therefore, to use `frame` as an index variable for an array or matrix, you must preface it with `(int)` in order to explicitly type its value to an integer.

```
// Store the cell coordinates in the variables $x and $z.
$x = $cellPos[(int)(frame-1)][$i*2];
$z = $cellPos[(int)(frame-1)][$i*2+1];
```

The variable `$name` is used to store the name of the current cell. Below you'll assign this variable and use it to check if the cell exists in the scene already. If it doesn't exist, you'll create a sphere called `$name`.

```
$name = "cell" + $i; // $name is the cell name.
// Make a sphere if one doesn't already exist.
if (!`objExists $name`) sphere -r 5 -n $name;

// Move the sphere into position.
move -absolute $x 0 $z $name;
```

The **objExists** command checks the existence of an object in your scene. If the object does exist, objExists returns **1**, if not, it returns **0**.

```
        // Make the curve.
        // The curve command doesn't require a -name flag in
           create mode.
        curve -point $x 0 $z -name ("curve" + $i);
    }
```

Reset variables

Frame 1 is also the place to clear variables that should be reset each time you play the animation. If these variables are not cleared, future assignments will simply add to their current values rather than replace them.

```
    // Clear the migration statistics variables.
    clear $L; clear $dNet; clear $Dc;
    $dataOut = $Lstring = $dNetString = $DcString = " ";
```

Now close the outer `if` statement that checked if Maya is at frame 1. The rest of the expression will execute for every frame greater than 1.

```
} // End if (frame == 1) statement.
```

The main loop

This next section executes for every frame after frame 1, up to the end of the run at frame 230. It positions the cells and calculates incremental values for L (`$L`).

```
else { // frame > 1
    // Position the cells.
    for ($i = 0; $i < $cellCount; $i++) {

        // Store the cell coordinates in the variables $x and $z.
        $x = $cellPos[(int)(frame-1)][$i*2];
        $z = $cellPos[(int)(frame-1)][$i*2+1];

        $name = "cell" + $i;
        move -absolute $x 0 $z $name;
```

You won't be using the cell name again in this loop, so you can reuse the variable `$name` to store the name of the curve that will trace out the trajectory of the current cell (cell `$i`).

```
    // Add to each cell's trajectory curve.
    $name = "curve" + $i;
    // The -append flag tells the curve command to add to an
      existing curve.
    curve -append -point $x 0 $z $name;

    $newPos = <<$x, 0, $z>>;

    $x = $cellPos[(int)(frame-2)][$i*2];
    $z = $cellPos[(int)(frame-2)][$i*2+1];
    $oldPos = <<$x, 0, $z>>;

    // Calculate the displacement, L, for cell $i.
    $L[$i] += `mag ($newPos - $oldPos)`;
    // mag returns the scalar length of the vector.
    }
}
```

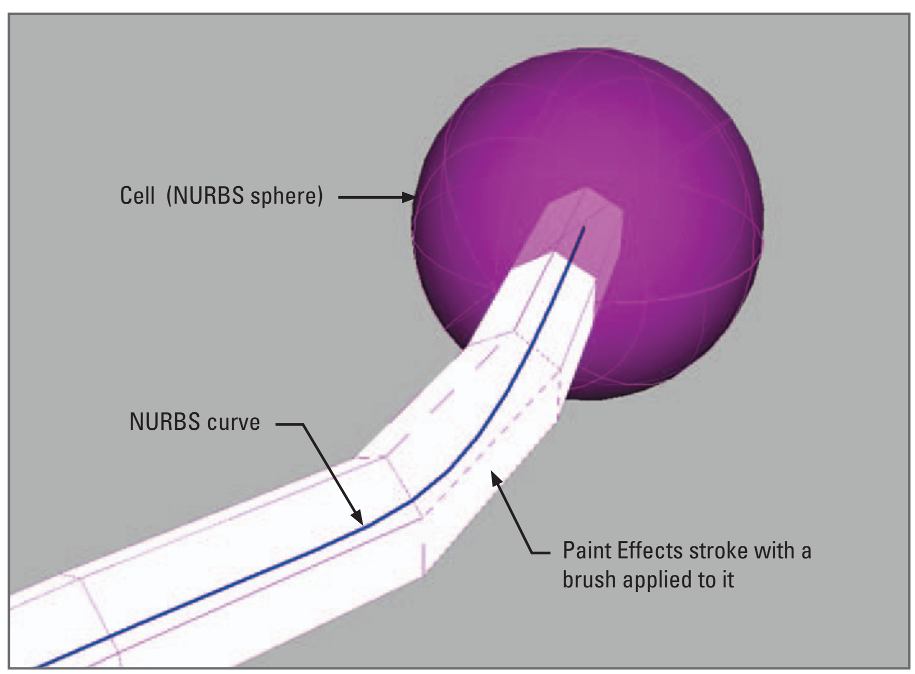


FIGURE 13.07

A Maya Paint Effects stroke is attached to a NURBS curve that traces out the trajectory of the cell (purple sphere).

Create the Paint Effect strokes

Here you make the cell trajectory paths renderable by converting each NURBS curve to a Paint Effects stroke (Figure 13.07). By frame 4, each cell path curve has 4 CVs—the requisite number for a stroke to properly adhere to its curve. A brush is a Paint Effects node that controls the appearance of a stroke. You can simplify your task of setting attributes for all six strokes by forcing them to use the same brush node via the shareOneBrush command.

```
if (frame == 4) {// Add a Paint Effects stroke to each curve.

   // Make the Paint Effects strokes.
   select `ls -tr "curve*"`;
   AttachBrushToCurves;
   convertCurvesToStrokes;
   string $strokes[] = `ls -tr "stroke*"`;
   select $strokes;
   ShareOneBrush;
```

Next, you'll set the `sampleDensity` and `useNormal` attributes of the strokes' shape nodes. Sample density determines the smoothness of the stroke—the higher the sampling, the smoother the curve. Use normal alignes the stroke with the curve normals so that it doesn't twist and cause constrictions. To see the effect of useNormal turn it on and off manual in the Channel Box and see what happens to your strokes.

```
   string $name;
   for ($name in $strokes) {
```

```
        string $tmp[] = `listRelatives -children $name`;
        setAttr ($tmp[0] + ".sampleDensity") 50;
        setAttr ($tmp[0] + ".useNormal") 1;
      }
```

Query the name of the brush node and use it to set the color and brushWidth attributes. You can set these manually in the Channel Box or Attribute editor to tune the stroke appearance once the nodes have been created.

```
      string $list[] = `listConnections strokeShape1`;
      // Make the stroke white.
      setAttr ($list[0] + ".color1") -type double3 1 1 1;
      setAttr ($list[0] + ".brushWidth") 0.2; // Set the brush width.
    } // End if (frame == 4).
```

You've reached the end!

Upon reaching the final frame, 230, you'll need to make some summary calculations and then write the statistics out to a file. An easy way to test if playback has stopped is to run `playbackOptions` in query mode. If `playbackOptions` returns a value of 1 then playback has stopped.

```
    if (frame == `playbackOptions -query -maxTime`) {

      /*
      Calculate the net displacement, Dnet, and the average values for
        $N, $L, and $D.
      */

      for ($i = 0; $i < 6; $i++) {

        // Net displacement.

        $x = $cellPos[(int)(frame-1)][$i*2];
        $z = $cellPos[(int)(frame-1)][$i*2+1];
        $newPos = <<$x, 0, $z>>;

        $x = $cellPos[(int)0][$i*2];
        $z = $cellPos[(int)0][$i*2+1];
        $oldPos = <<$x, 0, $z>>;

        $dNet[$i] = `mag ($newPos-$oldPos)`;
        $dNetSum += $dNet[$i]; // Sum the net displacement for all
          eight cells.
        $Lsum += $L[$i]; // Sum the total distance traveled for all
          cells.
        $Dc[$i] = $dNet[$i]/$L[$i]; // Calculate the directedness
          ratio.
      }

      $dNet[6] = $dNetSum/6; // Average net displacement for all cells.
      $L[6] = $Lsum/6; // Average distance traveled for all cells.
      $Dc[6] = $dNet[6]/$L[6]; // Average directedness ratio for all
        cells.
```

Title: Cell Migration Summary Data
Author: Your Name
****

***** Directedness Coefficient *****

Cell	0	1	2	3	4	5	Average
L	424.370	260.083	243.172	1141.795	1040.529	490.636	600.098
Dnet	100.810	62.922	110.635	513.991	412.584	71.386	212.055
D_c	0.237	0.242	0.455	0.450	0.396	0.145	0.353

FIGURE 13.08

The summary report file as it should appear in a text editor or spreadsheet application. It contains the migration statistics net displacement (Dnet), total distance traveled (L), and the directedness coefficient (D_c).

```
/*
Put $N, $D, and $L arrays into single-value strings to print to
the text file. The first character in each string identifies
the variable: L, Dnet, or Dc. Tab characters are used to
separate each array element.
*/
for ($i = 0; $i < 7; $i++){

     $Lstring = $Lstring + "\t" + $L[$i];
     $dNetString = $dNetString + "\t" + $dNet[$i];
     $DcString = $DcString + "\t" + $Dc[$i];
}
```

One way to write the data out to a file is in the form of a single, long string with tab (\t) and new line (\n) characters used where appropriate. This allows you precompose your output data and send it to the file with a single `fprint` command. To make the following line of code more legible we broke the variable assignment into several statements, each time adding a new chunk to the existing string. Figure 13.08 shows what the output file will look like when viewed in a text editor or spreadsheet application.

```
// Compose the summary report for printing.
$dataOut = "Title: Cell Migration Summary Data\n";
$dataOut =  $dataOut + "Author: Your Name\n";
$dataOut = $dataOut + "***********************************\n\n";
$dataOut = $dataOut + "***** Directedness Coefficient *****\n";
$dataOut = $dataOut + "Cell:\t0\t1\t2\t3\t4\t5\tAvg.\n";
$dataOut = $dataOut + "L" + $Lstring + "\n";
$dataOut = $dataOut + "N" + $dNetString + "\n";
$dataOut = $dataOut + "D" + $DcString;
```

The following procedure is very similar to the one you used to open cellData.txt at the beginning of this expression. The difference with this procedure is that it opens (creates) a new file and then uses `fprint` to print the long string value you have stored in `$dataOut` to it. You will be prompted to provide a file name by the file browser which opens in “write” (1) mode.

```
// Define a procedure to fopen and fprint the data file.
global proc fprintFile(string $mode, string $fileName, string
  $type) {
```

```
    // Open the file for reading.
    $fileID = `fopen $fileName $mode`;
    // Print the data string and close the file.
    fprint $fileID $dataOut;
    fclose $fileID;
  }
  // Open a file browser and send the "w" mode argument to
    fprintFile.
  fileBrowserDialog -m 1 -fc ("fprintFile" + " \"w\" ") -ft ""
    -an "Save_File";
}
```

That's it for your animation expression. Be sure to save your file. Now it's time to watch the "cells" move and check out your summary report.

Running the script

Before you create the expression, make sure that there are no external files currently open in Maya—you may have inadvertently left one or more open while testing out bits of code throughout this chapter. Enter the following code in the Script Editor. It will close up to 10 open files. If you like, you can save this to your custom shelf for easy access.

```
int $i;
global int $fileID;
for ($i = 0; $i < 10; $i++) fclose $i;
```

Before anything can happen in your scene, you must create an expression from your script.

1. **Open moveCells.txt (either the file you just created or the one on the CD-ROM) in your text editor.**
2. **Select and copy the entire script.**
3. **In Maya, open the Expression Editor by entering the following command in the Command Line or Script Editor.**

   ```
   ExpressionEditor;
   ```
4. **Press the New Expression button.**
5. **LMB + click in the Expression text field.**
6. **Press Ctrl + V to paste your copied script into the text field.**
7. **Press the Create button at the bottom of the Expression Editor.**
8. **In the Expression Name Field, replace the default name with moveCells and press Enter.**

If Maya accepts your script without displaying error messages, the file browser should open and prompt you to locate and select a file.

9. **Browse cellData.txt, select it, and press Open_File.**

The default perspective camera settings won't show enough area in your scene to view all of the cells and their paths at once. To remedy this, increase the far clipping plane:

10. **Enter the following in the Script Editor or Command Line:**

```
setAttr "perspShape.farClipPlane" 30000;
```

11. **Manipulate your scene view so that all cells are visible.**
12. **Press the Play button to start the animation.**

Debug if necessary

If Maya generates one or more errors when you created the animation expression, you will need to debug your script: open the Script Editor to view the specific error messages and to read the line number(s) that generated them. If your text editor can display line numbers, use this feature to cross-reference the error messages to the offending lines in your script. If you are unable to resolve the errors, you can compare your script to the version of **moveCells.txt** included on the book's CD-ROM.

Play your animation

After you've successfully created the moveCells expression and loaded `cellData.txt`, press the Play button to see the results. When the animation reaches frame 230, the file browser will prompt you to enter a name and choose a location on your hard drive for the summary file. After saving this file, open and inspect it in your text editor. If the columns do not line up properly, you can adjust the placement of tabs in the application so that there is enough width between each to facilitate the data. You may wish to open or import your file into a spreadsheet program to test how the tab-delimited strings are parsed into spreadsheet cells.

The way you've crafted moveCells, you can rewind and play the animation as many times as you like. You may also want to make a quick movie of your animation using Maya's playblast feature. For a reminder on how to do this, refer back to page 184 in *Chapter 7*.

Summary

Maya's ability to interact with your computer's file system makes it easy to import and export custom data—a feature that makes the program extensible to a wide range of data visualization and in silico biology simulation applications. In the tutorial above you read the cell migration coordinates into a variable in single step. For large data sets it may be more desirable to import values only as they are needed rather than all together which can use up valuable memory. Conversely, when running a simulation that calculates a large number of attribute values—say, for a population of migrating cells or a large number of interacting molecules—you can write the values out to a file periodically rather than storing them in variables or keyframing them. Until a scene is saved, keyframe values are stored in RAM. When RAM is used up, virtually memory takes over, which can slow a simulation considerably because it involves writing the keyframe information to your hard drive and then fetching it back again, every time the CPU needs it. Likewise, if values are stored in variables and those variables meet or exceed you computer's memory limitations, your simulation can fail in

midstream. A safer alternative to ever-growing sets of variables and keyframes is to write attribute values to a file, frame by frame, using the fflush command. Doing so gives the added protection of keeping a permanent record of your results as they're produced. In the unlikely event of a Maya abort or a computer crash, your simulation results will be safely stored in a file up to the time of the crash.

In the next chapter, the first of the case study projects in *Part 3* of the book, you'll use Maya's file reading commands (`fopen`, `fgetline`, and so on) to work with a different kind of data: the atomic coordinates stored in a protein structure file. You'll learn a practical workflow for building biomolecules—and the first step in modeling living systems in Maya.

Part 3

Biology in silico

Maya in action

This page intentionally left blank

14 Building a protein

Introduction

In this chapter you will create a 3D model of a protein using data stored in a molecular structure file. This is a natural spot to begin both because of the place of single molecules in the organization in biological systems and because it involves simple geometry and texturing in Maya. It also takes us into a very practical application of MEL scripting: to automate a modeling task that would be tedious if done 100% manually. By the end of this chapter you know how to import data from a file in **Protein Data Bank** (**PDB**) format into Maya and how to use it to create surface models of a biomolecules similar to the one in Figure 14.02d. Models like this are the basic elements of the universal visual language scientists use to describe the structure of living matter.

The PDB was founded in 1971 by Brookhaven National Laboratory. Currently administered by the Research Collaboratory for Structural Bioinformatics (RCSB), it is a key worldwide resource in structural biology. As of January 2008, it housed over 48,000 structures.

The PDB data file format is a global standard for macromolecular structure data derived from X-ray diffraction and NMR crystallography studies.

Research Collaboratory for Structural Bioinformatics (RCSB) Protein Data Bank (PDB) website:

http://www.pdb.org/

Visualizing macromolecules: A very brief history

Since the early models of John Dalton—who in 1808 proposed that all matter is made of atoms—scientists have created 3D depictions of molecules in order to understand their structure. Structure in turn elucidates function; how a molecule works as it performs its duties inside the cell. A knowledge of the function of biomolecules, notably nucleic acids (DNA, RNA, etc.) and proteins, ultimately leads the development of therapies, cures, and strategies for prevention of disease. 3D models were essential to James Watson and Francis Crick as they solved the geometric structure of DNA.[1] In turn, the arrangement they uncovered of nucleotide bases in opposite pairs revealed the mechanism for copying genetic material. This is just one of many examples of the vital role played by visualization in the discovery process in biology.

Moreover, while Watson and Crick, and their contemporaries, labored over large, handmade models, advances in computing have made creating 3D molecular models from structural data a relatively simple task. Maps showing the positions of atoms in a molecule were once transcribed by hand into 3D models by innovators such as Linus Pauling and Robert Corey (Figure 14.01). Software now exists to convert these maps into letters and numbers describing the element (carbon, oxygen, etc.) and location in space of each individual atom in a given dataset, and then turn them into 3D computer visualizations.

Further reading → **iVis in Action: Molecules, cells, and tissues →Interpretive visualization (iVis)**

Wires, ribbons, and surfaces

Computer algorithms enable us to represent biomolecules in different visual styles, each of which has its historic roots and utility. Several widely used styles are shown for a lysozyme molecule in Figure 14.02. Surface models, similar to those shown in Figure 14.02d, e, and f, are used in many applications including the visualization of

FIGURE 14.01

Linus Pauling and Robert Corey with some of their signature space-filling molecular models (c. 1951).

Courtesy of the Archives, California Institute of Technology.

shape complementarity between molecules, which plays an important role in the drug discovery process. Simply put, candidates for a new drug are molecules which bind to and enhance or inhibit the activity of a certain biomolecule implicated in a disease. Just as a key fits a lock, these molecules of complementary shape can recognize on another and so activate their functions in the cell. A well-designed drug can target this recognition event to enhance its effects or, if it's dangerous, block it.

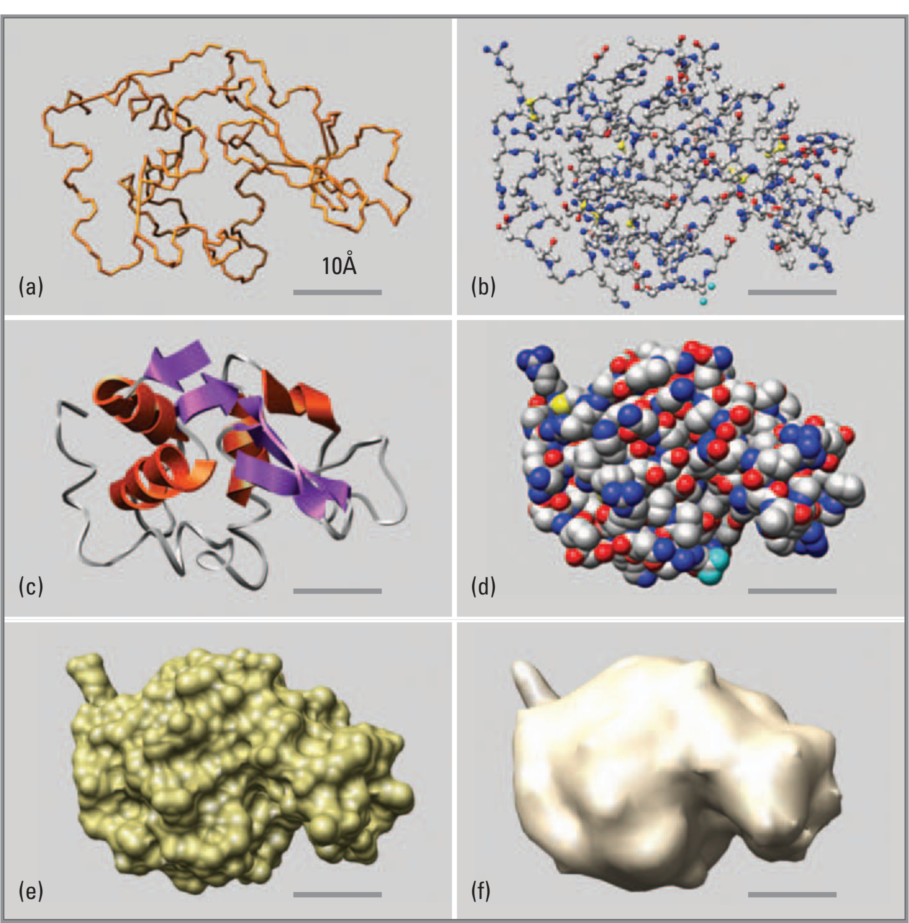


FIGURE 14.02

The structure of the digestive enzyme lysozyme (14.3 kDa; 1001 atoms) was solved in 1965 by David Chilton Phillips. With it, he went on to explain how enzyme function relates to structure. Scale bar = 10 Å

(a) wire backbone

(b) ball and stick

(c) ribbon

(d) CPK surface

(e) contact surface

(f) low-resolution surface.

We modeled these structures using the UCSF Chimera package from the Resource for Biocomputing, Visualization, and Informatics at the University of California, San Francisco (supported by NIH P41 RR-01081). (Petterson EF et al.: UCSF Chimera – A visualization system for exploratory research and analysis. *J. Computational Chemistry* 25: 1605–1612, 2004). For molecular structure, we used PDB entry 2LYZ (Diamond R, Phillips DC, Blake CCF, North ACT: Real-space refinement of the structure of hen egg-white lysozyme. *J. Molecular Biology* 82: 371–391, 1974).

The Dalton (Da) is a unit of atom mass. One Dalton is 1/12 the mass of one Carbon-12 atom. A kilodalton (or kDa) is equal to 1000 Da.

The Angstrom (Å) is a unit of length used for atomic dimensions and light wavelengths. It is equal to 1×10^{-10} meters, or 0.1 nm.

Putting molecular scale in perspective:

10^{0} meter (m) → a dog

10^{-2} meter (cm) → a tooth

10^{-3} meter (mm) → a pin head

10^{-6} meter (μm) → a cell

10^{-9} meter (nm) → a molecule

10^{-10} meter (Å) → an atom

Understanding how molecules interface with one another—their shape complementarity—accelerates the search for potential matches.

In this chapter you will create a surface model of an **actin** molecule in a style reminiscent of the **CPK** models produced by Harvard Apparatus, USA. The CPK style, named for chemists Robert **C**orey, Linus **P**auling, and Walter **K**oltun uses a colored sphere to represent the **van der Waals** (vdW) **radius** of each atom in a molecule (see Figure 14.02d). With Maya and MEL tools in hand for a basic surface model, you'll be ready to tackle more advanced structures such as ball and stick, wireframe, and ribbon models of molecules using the data embedded in PDB files.

Level of detail

Figure 14.02d, e, and f, are all depictions of the surface of a lysozyme molecule but they differ in the level of detail (**LOD** from here on) shown. Another term that is sometimes used interchangeably with LOD is **resolution**; high LOD corresponds to high resolution and vice versa. There are two factors to take into account when considering LOD for a given model. The first is the need for effective visual communication.

For instance, how much surface detail is important to what you're trying to show? The individual colored atoms in a CPK model become superfluous when the point is to depict the general globular shape of a protein, and will likely even interfere with the perception of that shape. The second factor has to do computer processing. The more detail that must be calculated and drawn on a computer display, the greater the demand on processing speed and power. For example, a molecular dynamics simulation shown as highly detail CPK models like the one in Figure 14.02d would likely be much slower to run than the same simulation containing the low-resolution surfaces shown in Figure 14.02f.

The van der Waals atomic radius (vdW for short) is named for Johannes Diderik van der Waals, b. 1837, d. 1923. It corresponds to the distance between two atoms when they are "just touching", that is when they are packed side by side but have negligible chemical bonding interaction. The interior of a crystal, where atoms can be eased together cheek by jowl, has been a favorite hunting ground for chemists' pursuit of van der Waals' radii, which in turn are used by structural chemists to represent an idealized "contact surface" for each atom when they push together but do not embrace through the added force of a chemical bond.

In this project, you are going to create a detailed space-filling model for purpose of rendering the atomic structure of an actin protein. In the next chapter we will introduce a low LOD actin model.

Visualization freeware

The 1990s saw an explosion of molecular visualization (a visualization process sometimes described as **MolVis**) and modeling software applications, many of which are available to use free of charge. A group of these applications derive from computer scientist Roger Sayle's RasMol[2] program for which he generously made the C source code freely available. While we don't describe this software here, we do encourage you to become familiar with one or more of the key **freeware** applications listed below. They provide a quick, interactive way to view molecular structures in different styles. One can therefore preview different structures before choosing which one to import into Maya. This is often helpful since there may be multiple PDB entries for a given molecule, each with features peculiar to the circumstances under which the data was collected.

Molecular visualization freeware

UCSF Chimera **http://www.cgl.ucsf.edu/chimera/**

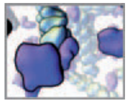
JMol **http://jmol.sourceforge.net/**

VMD **http://www.ks.uiuc.edu/Research/vmd/**

USCF Chimera is a MolVis freeware application with particular utility in our current in silico modeling workflow (see Figure 14.03). With it, you can generate and export a single polygonal object which is representative of a given molecule's surface and can be subsequently imported into Maya. Furthermore, Chimera lets us preprocess the raw molecule data into varying levels of detail for the surface. The low-resolution actin model used in the next chapter was created using a Chimera-to-Maya workflow and is provided in a Maya ASCII file in the CD-ROM supplement for *Case study 2*. To date, we find Chimera the best suited of the MolVis freeware programs to a MolVis/Maya surface modeling workflow. Chimera is relatively intuitive to use and, most importantly, produces well-constructed 3D surface models that are easily edited in Maya.

Freeware is a term given to software applications made available to the public free of charge, often over the World Wide Web.

This discussion of MolVis software perhaps begs the question, with all of this molecular modeling and visualization software freely available, why use Maya? Maya's capabilities for editing, shading, animating, and rendering molecules are far more extensive and flexible than those of any MolVis freeware application currently available. Maya has unmatched built-in scripting and dynamic simulation features which



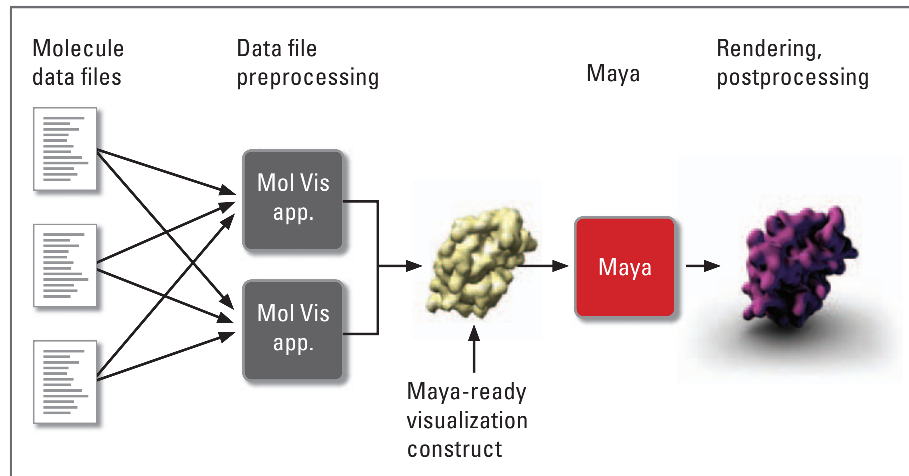


FIGURE 14.03

A workflow that leverages molecule surface modeling capabilities in MolVis freeware applications for advanced visualization in Maya. We have found the 3D models created in UCSF Chimera (versus other MolVis freeware programs) to be the most compatible with Maya.

are not rivaled in any current MolVis program. Maya was developed to create stunning visual effects: the lighting, shading, and rendering capabilities of MolVis applications don't even come close to Maya's. To create a basic still image of a molecule for journal publication or classroom teaching, a program like Chimera may be all you need. But to go further, Maya's capabilities open a world of possibilities for molecular simulation and visualization. In this case study, you won't be using a MolVis program. Instead, we want you to work with all the steps, from raw protein coordinates to final polished model, so you will import protein structure data directly into Maya to build an actin model. You will automate the process using a custom MEL script and then light and render the model to create an image that no MolVis program can rival.

Problem overview

Your objective is to prepare a MEL script that creates a CPK surface model of a molecule using structure data contained in a PDB file. This model will be static, intended for structure visualization, so there is no need to rig it for dynamic behaviors. You will use the MEL script first to create a small adenosine triphosphate (**ATP**) molecule (Figure 14.04). This makes a great learning exercise. You'll follow that up with the big, complex actin protein. Finally you will make a production–quality rendering of the actin.

ATP: The energy currency of life

The **nucleotide**, ATP, is a source of energy for many metabolic processes in living organisms. It is a small molecule, weighing 0.5 kDa, with only 31 atoms (excluding hydrogen) and therefore makes good data with which to try out your script since the PDB file can be loaded and modeled quickly; if there is an error in your script, you won't have to wait long to find out!

Actin

Actin is 43 kDa **structural protein** that is plentiful in **eukaryotic** organisms and critical to many biological functions including cell movement and muscle contraction.

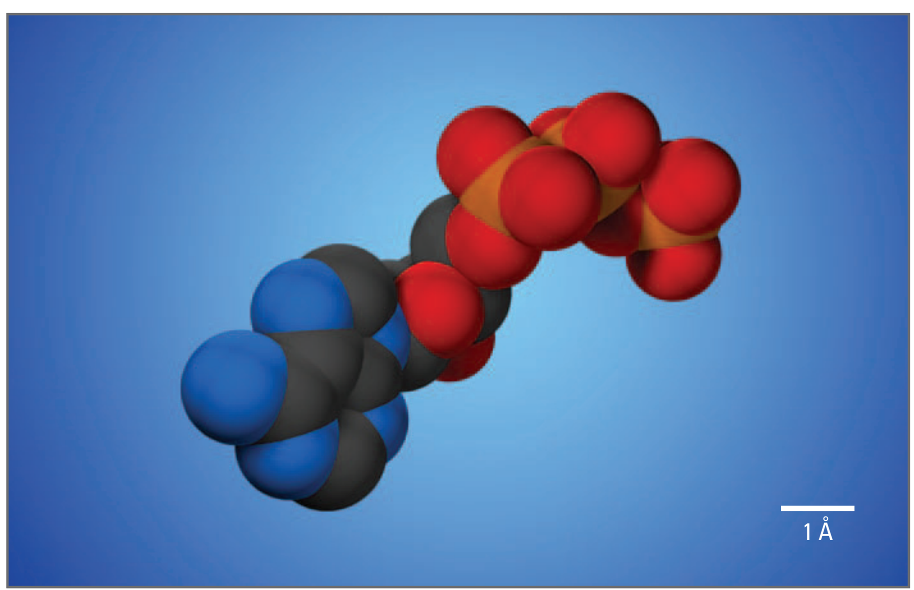


FIGURE 14.04

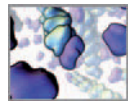
The nucleotide, **ATP**, is a vital source of energy for a large number of metabolic processes within cells. It is a small molecule and has a distinct shape, making it an ideal test subject for building your first molecule in Maya.

Relative to ATP, actin is fairly large, comprised of some 2,880 atoms. It is a typical polypeptide—a chain of amino acids, folded in upon itself to produce a higher-order functional unit.

Under the right conditions, actin polypeptides form regular chain-like polymers called actin filaments. An individual actin chain is called a **subunit** within a filament and a **monomer** when in the unpolymerized state. Actin filaments are a major component of the protein scaffolding, or **cytoskeleton**, that give a cell its shape. Figure 14.05a shows actin filaments (stained red) comprising the cytoskeleton in a fibroblast, a kind of connective tissue cell. It is the assembly and disassembly of actin filaments that enable cells to change shape and move in their surroundings. In muscle cells, contraction is caused by motor proteins, called myosin, ratcheting along actin filaments. Figure 14.05b shows actin filaments arranged in contractile fibers, called myofibrils, within cardiomyocytes.

The distinctive shape of the actin monomer (Figure 14.06) determines its orientation relative to other subunits in a filament and, ultimately, the shape and physical and chemical properties of the filament itself. The actin monomer has two recognition regions on its surface; their properties give actin a fundamental polarity in the polymerized state. These are commonly referred to as “barbed” and “pointed” or “plus” and “minus” ends of the actin filament. An obvious cleft between the two peaks of the pointed end reveals the binding site for a nucleotide—one of ATP or its de-energized form, ADP (adenosine diphosphate), which is ATP less one phosphate group.

Its many roles make actin one of the most widely studied of all biomolecules. There is plentiful literature of science journal articles devoted to its structure and function. As well, actin filament assembly has been the subject of numerous computational modeling efforts. Therefore, you have much material to draw upon when using actin as a test subject around which to build 3D animation-based strategies for modeling biology.



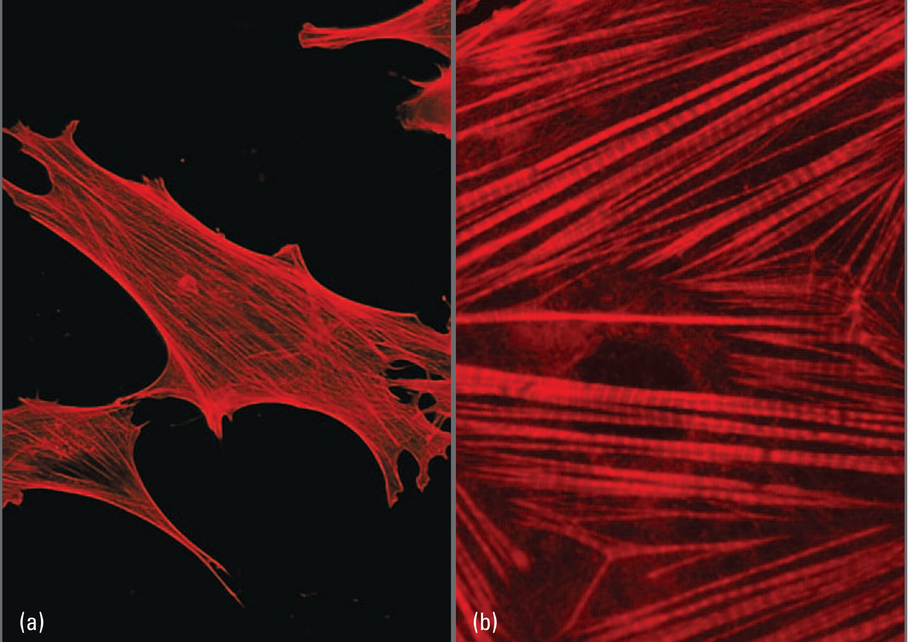


FIGURE 14.05

Actin filaments labeled with phalloidin conjugated to a red fluorescent dye.

(a) The actin cytoskeleton of a cultured fibroblast cell.

(b) The actin component of myofibrils within a cultured cardiomyocyte.

Images courtesy and copyright © 2006 Sylvia Papp and Michal Opas, Institute of Medical Science, University of Toronto. From research supported by the Canadian Institutes of Health Research (CIHR).

Pointed (minus) end
10 Å
monomer
= subunit
filament
(polymer)
10 Å
Barbed (plus) end

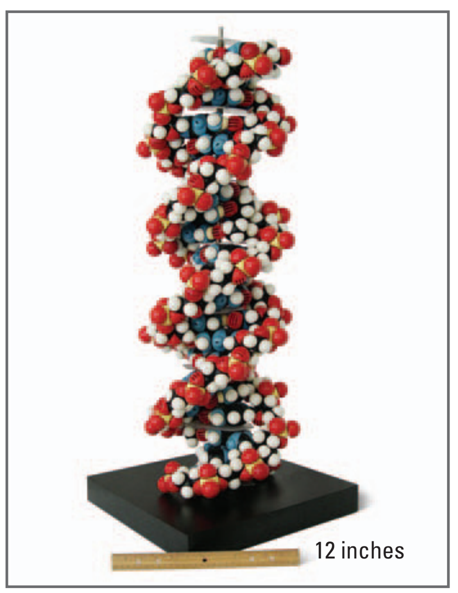
FIGURE 14.06

Actin momomer and filament. The monomer is shown as a contact surface model with its bound nucleotide (ADP) represented as a CPK model. The actin filament is depicted using low-resolution surfaces of the monomers. The surfaces were generated by USCF Chimera then modified and rendered in Maya.

The CPK look

What's in a look?

Atoms and molecules are much smaller than the wavelength of visible light (5,000–7,000 Å). This makes them completely invisible to the naked eye. Even stranger is the fact that atoms and their building blocks, the atomic nuclei which center atoms



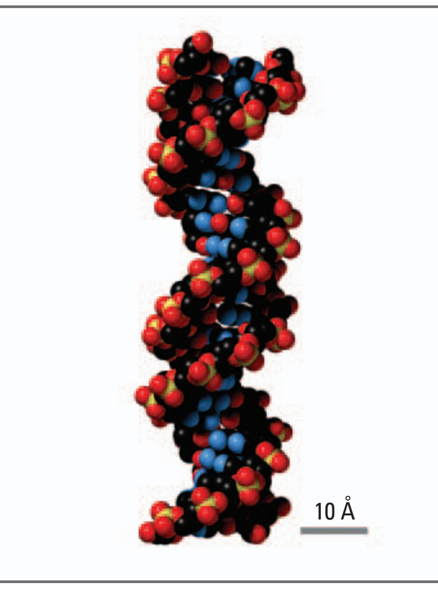


FIGURE 14.07

(a) A plastic CPK model of DNA, built circa 1972 and photographed by us in 2006; scale: The ruler is 12 inches long.

(b) A computer-generated CPK model of DNA; scale bar = 10Å. Note the absence of hydrogen atoms (white spheres) in the computer model.

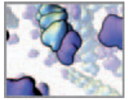
Plastic DNA model courtesy of Dr. Laurence A. Moran, Department of Biochemistry, University of Toronto, Canada. Computer model created in Maya using data from PDB entry 1A36 (Stewart L, Redinbo MR, Qiu X, Hol WG, Champoux JJ, A model for the mechanism of human topoisomerase I. *Science* 279: 1534–1541, 1998).

and give them most of their weight as well as the electrons that make atoms stick together by whizzing round them in smeared-out orbits, don't move the same way baseballs or satellites do. Physicists have determined that electrons play by different rules—the rules of atomic physics, which makes them very stealthy: unlike a baseball or the space shuttle, there is at no one moment any place an electron is "at", even though its action glues atoms together in molecules. Like fickle friends, electrons and nuclei have only places they are *likely* to be found (Figure 14.07).

The plastic CPK models manufactured by Harvard Apparatus (originally by Ealing Scientific Ltd.) use the color black to represent carbon. The convention in computer graphics is to use grey instead—presumably because black tends to flatten rather than accentuate form in print or display graphics, and because black objects are difficult to see against dark backgrounds.

Molecules, including the big molecules comprising living cells, are effectively flocks of electrons in motion through the space rendered by the centers of the atoms (the atomic nuclei) of the molecules. Since the electrons are not only too miniscule for the eye to see if they could be pinned down—but in fact *can't* be pinned down—atoms and molecules do not have a *visual appearance* in the sense we usually apply that term to the look of a friend's face or a gorgeous sunset. The pictures of atoms and molecules we see throughout today's media are therefore not a true-life rendering of a visual appearance we would all see if only the molecule could be expanded to the size of a baseball. These pictures are visual interpretations that artists, working with scientists, have made about certain vital properties of these molecules, which have a natural spatial interpretation. Two of these are very important for you to keep in mind in working with PDB and other molecular structure data. First, atomic nuclei, being a lot heavier than their flightier cousins the electrons, are more settled. They are more likely by far to frequent rather small regions (compared to the typical "size" of the atom, which we'll get to next) of space. The PDB coordinate you see for an atom's location is a measurement that says where the center of this preferred hang-out region is located.

The space-filling molecular models don't simply show where the atomic nuclei are, however. Compared to the size of an atom, the atomic nuclei are just specks. What is the space-filling substance portrayed in these evocative, often beautiful visual constructions? It all comes back to the electrons and their duties as the "glue" that bonds atoms together into molecular frameworks. Backing up the electrons that



Element	C	H	N	O	P	S
vdW radius (Å)	1.70	1.20	1.55	1.40	1.90	1.85
CPK color	gray	white	blue	red	orange	yellow
RGB values (normalized from 0 to 1)	R: 0.7 G: 0.7 B: 0.7	R: 1.0 G: 1.0 B: 1.0	R: 0.0 G: 0.5 B: 1.0	R: 1.0 G: 0.0 B: 0.0	R: 1.0 G: 0.5 B: 0.0	R: 1.0 G: 1.0 B: 0.0

TABLE 14.01

The vdW radii and CPK colors for the elements that occur in ATP and actin.

vdW radius data from: Pauling L. *The nature of the chemical bond and the structure of molecules and crystals*, Cornell University Press, Ithaca, NY, 1960.

While many publications and MolVis applications agree on colors for the most common elements (gray for carbon, white for hydrogen, red for oxygen and so on), they lack consistency in their RGB values. For instance, the blue used for nitrogen in one application may look quite different from the blue used in the next.

join the glue is an electron pack which prefers the space closer to the atomic nucleus, between the boundary electrons and the nucleus. This intermediate region of space is their turf and so on and they resist having it invaded: if you try to push two atoms together, you quickly find the inner electrons pushing back. The atoms bounce apart, or at least back to the distance they settle into under the joint influence of the gluing action of the outer electrons and the turf defense mounted by the inner electrons. Using the equations of atomic physics, chemists can map out how far from the atomic center or nucleus it is to the edge of this turf zone. So equipped, they can then measure this distance for specific atoms. So although atoms and molecules are jaw-droppingly strange and complicated when thought about scientifically—in terms of atomic physics and its esoteric math—atoms can behave toward on another in impressively simple ways: if they get too close, they act (thanks to the turf defense mounted by the inner electrons) rather like hard balls. The diameter of the hard balls is an estimate of the spatial turf over which the inner electrons resist intrusion, reduced a bit by any gluing action of outer electrons extending *their* turf into the space of neighbor atoms. The van der Waal's (vdW) radius is the name given to this turf size, in honor of an early investigator of atomic collisions. So, the inspiration of the space-filling molecular model is to ask not how the molecule would look to us if we made it bigger (though to atomic math it has no such "look"), but how as artists we might instead visualize how the atom looks to another atom or molecule. As you work with the CPK and other visual languages then, keep in mind you are modeling visual analogues of rather abstract things—the edges of quantum turfs that let molecules sense one another!

Atoms as spheres

CPK models represent each atom as a solid sphere of radius equal to the atom's vdW radius. Table 14.01 lists the vdW radii for the atoms that make up ATP and actin. These spheres are large compared to those used in a ball-and-stick model, and obscure the bonds between atoms. For convenience, we will deem one Maya unit equal to 1 Å. You will model each CPK sphere with a NURBS sphere, using the MEL command, `sphere`, as follows:

```
sphere -r $radius -n $atomName;
```

Color and material

Table 14.01 shows the CPK colors for the elements that comprise ATP and actin. By assigning a shader to each group of elements (all of the carbon atoms, for example), you will be able to easily adjust the colors after the model is built.

The semi-gloss quality of the material assigned to the spheres in Table 14.01 is typical of computer-generated CPK models and recalls the original plastic CPK models manufactured by Harvard Apparatus, Massachusetts, USA (www.harvardapparatus.com). This look can be achieved in Maya using Blinn shaders, which you can make and assign to atoms either during or after the creation of the model. You will do the former, automating the creation and assignment of the shaders within the main MEL script, using the following command:

```
shadingNode -asShader -shared blinn -name $shaderName
```

The **-shared** flag for the **createNode** command is important. It ensures that the node will only be created if one of that name doesn't already exist. Use it when you want to create only one of each type of shader. For instance, once the first carbon atom is created, a carbon shader is made. When a second carbon atom is created, you'll want to assign it the existing shader rather than create a new one.

What creates the appearance of "semi-gloss" is a diffuse specular highlight from the main light source. The Blinn shader attributes `Eccentricity` and `Specular Roll Off` control the size of the highlight and the distance over which its intensity fades to zero. These attributes can be adjusted after the model and shaders have been created by the MEL script.

Data

You will use two PDB files in this project. The first, `atp.pdb`, is located on the book's CD-ROM and contains atomic coordinates (xyz positions) for the atoms in an ATP molecule. You'll download the second file, `1j6z.pdb`, from the RCSB PDB website. `1j6z.pdb` contains the atomic coordinates for an actin protein monomer.[3]

Start by creating new project directory called CPK_Project on you harddrive, and within it, a directory called PDB. The resulting file path should look something like:

```
\My Documents\maya\projects\CPK_Project\PDB
```

Copy the file called `atp.pdb` from the CD-ROM to the PDB directory you created.

14_Protein/PDB/atp.pdb

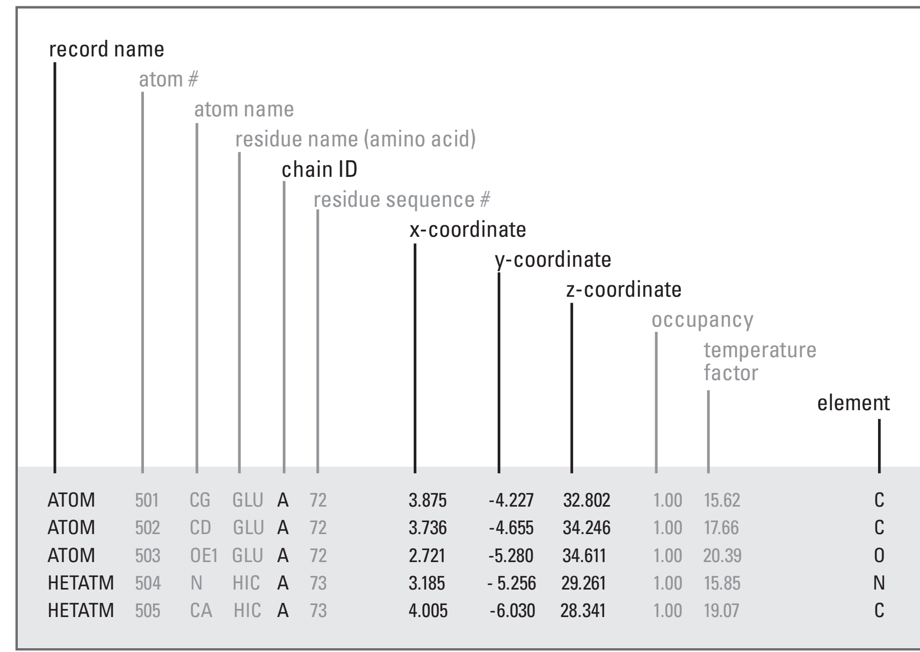
In a Web browser, navigate to the following website:

http://www.pdb.org/pdb/explore.do?structureId=1J6Z

Under the heading *Download Files*, click on *PDB text*. When prompted, browse to your PDB directory and press Save. This download/save process may vary slightly from one Web browser to another. What matters is saving the file `1j6z.pdb` to the new PDB directory on your computer. Next, open and inspect `1j6z.pdb` in a text editor.

PDB file format

Figure 14.08 shows a several lines taken from the middle of the actin file `1j6z.pdb`. It describes how data is to be organized in rows and columns within a text file. Each record occupies a row, while columns are reserved for specific data types. For this



record name	atom #	atom name	residue name	chain ID	residue sequence #	x-coordinate	y-coordinate	z-coordinate	occupancy	temperature factor	element
ATOM	501	CG	GLU	A	72	3.875	-4.227	32.802	1.00	15.62	C
ATOM	502	CD	GLU	A	72	3.736	-4.655	34.246	1.00	17.66	C
ATOM	503	OE1	GLU	A	72	2.721	-5.280	34.611	1.00	20.39	O
HETATM	504	N	HIC	A	73	3.185	- 5.256	29.261	1.00	15.85	N
HETATM	505	CA	HIC	A	73	4.005	-6.030	28.341	1.00	19.07	C

FIGURE 14.08

Excerpt from the PDB file, 1j6z.pdb[3] for the crystal structure of uncomplexed rabbit actin. Note that each row corresponds to a unique atom and that the columns contain data specific to that atom. These are the column entries commonly found in PDB files. Those that you'll use in your MEL script are printed in black.

In most PDB files, hydrogen atoms are absent because they are too small for the resolution of X-ray diffraction crystallography. In contrast, hydrogen atoms are present in data derived by NMR crystallography.

project, you're interested in the Cartesian coordinates of individual atoms, which are stored in record entries of types ATOM and HETATM. HETATM records are for water molecules and for **heterogens**—entries that are not part of amino or nucleic acids, such as inhibitors, solvent molecules, and ions. Your actin model will not include heterogens, but they are listed in the PDB file nonetheless.

Descriptions of the PDB format for ATOM entries

http://www.wwpdb.org/documentation/format23/sect9.html

Important note on PDB file column numbers: Since Maya reads files word-by-word and not character by character, one can truly be sure which columns certain data appear in only by opening and inspecting a given PDB file in a text editing application such as Wordpad in Windows or TextEdit in Mac OS.

A column in the PDB format is one character wide. Therefore one data type will often occupy several columns. For example, columns 1 through 6 are reserved for the record name (ATOM or HETATM, for instance). Columns 7 through 11 are reserved for data called “Atom serial number”, and so on. There are 16 data types in all reserved for ATOM and HETAM entries. However, real PDB files rarely contain all types (Figure 14.08 is a fairly typical example). This results in blank or empty columns, which is not a problem as long as your program that reads and interprets the file counts characters (including blank spaces) correctly and therefore knows exactly when it arrives at a particular data type. If, on the other hand, the program reads in data word-by-word instead of character by character, it may mistakenly attribute data to the wrong column.

Maya falls into this latter category; it reads *word-by-word* using the `fgetword` command. We therefore urge you always to preview the PDB file in a text browser to determine which columns Maya needs to locate the appropriate data. From here on, we will use the Maya interpretation of a column, as shown in Figure 14.08.

To create a vdW surface model, the columns we are interested in are:

record name

This identifies the type of data to follow in the row. For the present example, we're interested only in ATOM records. HET or HETATM records include hydrogen and oxygen atoms belonging to water molecules and heterogens. The record name enables us to filter out these and other unwanted entries.

chain ID

It is worth noting that atp.pdb and 1j6z.pdb have only one chain each, but you'll want your solution to accommodate other PDB files, many of which contain two or more chains. That way you can use the same script to handle any PDB entry, regardless of the number of chain it contains.

x-coordinate y-coordinate z-coordinate

Orthogonal coordinates in angstrom (Å) units, relative to the reference origin chosen by the file authors.

element

The one- or two-digit shorthand for the chemical element.

The remaining columns don't concern us for this project and won't be dealt with here. For more information on these and other PDB file format specifications, visit the following link:

PDB file format specifications

http://www.rcsb.org/pdb/file_formats/pdb/

Naming your models

Assigning unique names to objects is essential to the way Maya operates. It follows that your script must create uniquely named objects as it builds a molecule. Furthermore, it is often helpful to have hierarchical groupings of objects for easy selection. For instance, if all carbon atoms are in one group, you can quickly assign a new shader to them or by assigning it to the group. Likewise, if you wish to transform a molecule, you can do so by transforming the molecule group node. This strategy is consistent with Maya's scene hierarchy which maps transformations and deformations through parent/child relationships. We will use the following naming convention for a molecule, its chains, and their atoms. To remain consistent with Maya's default naming strategy, numbering will begin at zero (0).

molecule0 (the first molecule created in the scene)

→ **chain_A0 (chain A belonging to the first molecule)**

→ **elementGroupA0 (a group of like atoms within chain A)**

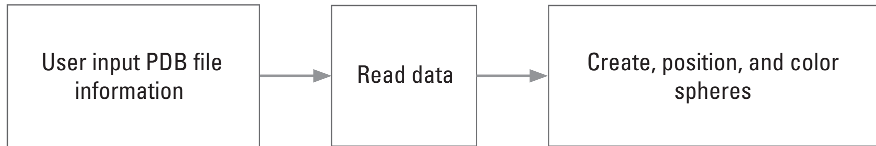
→ **element (an individual atom; e.g. "carbon")**

Unique *molecule*, *chain*, and *group* names will allow you to build hierarchical relationships without encountering naming conflicts. Since individual atoms have no children, you won't need to select them by name in order to parent objects underneath them.

Therefore atoms don't require unique names. You'll let Maya assign default names to atoms in order to make them unique only within their parent atomGroup nodes.

Methods: Algorithm design

Simply stated, your solution in Maya must do the following:



Due to the repetitive nature of creating, positioning, and coloring almost 3,000 atoms, this problem is well suited to procedural modeling using a MEL script. After Maya creates and shades the models, you can then light and render your scene using the techniques described in *Chapters 10 and 11*.

Flowchart

Now let's make a plan by identifying the necessary steps and flow control for your molecule-building procedure. In the flowchart show in Figure 14.09, the steps in rectangles will become MEL commands, while those in diamonds will become conditional statements and loops in your script. You will build the script in two parts. One procedure will read and store data. A second procedure will use the data to create, position, and shade the atoms.

Methods: Encoding the algorithm

You'll now turn each element in our flowchart into a form that Maya understands: the MEL script. Your script will take the form of a procedure (a user-defined function in MEL). Essentially, the procedure function, proc, wraps many separate instructions into a single global command, specified by the procedure name, that you can call any number of times from anywhere in the Maya environment. Let's build the code in pieces, parceled by command or logical task, with a brief explanation for each. To work as a procedure in Maya, all lines of code must be entered together, in the same sequence that they appear in the text. The complete MEL script can be found on the book's CD-ROM. You'll find it useful for checking your work or, if you want a very fast results, seeing cpk.mel in action right away!

14_protein/MEL/cpk.mel

Composing the MEL script

We recommend building your MEL script in a text editor that is well suited to scripting (see page 302 in *Chapter 12*). You can type it in as you follow along with the instructions below. That way you can save a script file periodically. Save it in your **Maya Scripts directory** which, in Windows, will be something like:

C:\Documents and Settings\User\My Documents\maya\8.5\scripts

Use the following file name:

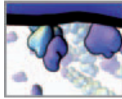
cpk.mel

START
procedure #1
cpk()
open PDB file
read record type
is record of type "ATOM"?
no
go to next line
yes
read & store record:
• chain ID
• x, y, z coordinates
• element
end of file?
no
yes
procedure #2
vanDerSphere()
make, position, and shade spheres
END

FIGURE 14.09

Flowchart for the cpk.mel script to read data from a PDB file and then create a CPK model in Maya.

When you want to try out bits of code in Maya, simply copy and paste them from your text editor into Maya's Script Editor. When your MEL script is complete you can load it into Maya in one step by sourcing the text file. We don't recommend composing anything longer than a few lines of code in the Script Editor. Its editing capabilities are elementary and if Maya crashes, you will lose anything you typed in that you didn't explicitly save out.



cpk() Procedure

The script starts with the procedure command proc. Your procedure, cpk(), is going to take three variables for arguments. As procedure "arguments", these variables are declared within the brackets next to the proc command (below).

```
/***** cpk.mel *****/
/*
Date: created February 01 2006; modified August 15 2007.
Authors: Jason Sharpe, Charles Lumsden, Nick Woolridge.

Description:
This procedure reads ATOM entries from a PDB file. It calls a
second procedure to make and shade a sphere to represent each atom
according to its element (oxygen, carbon, etc). The end result is
a CPK-style model built from sphere representing the van der Waals
radii of the constituent atoms.

The procedure arguments are as follows:
$chaincol       The pdb file column in which the chain letters reside.
$xcol           The column in which the x-coordinates reside
                (starting with 1).
$elemcol        The column in which the element (atom) names reside.

To use this script:
Save the entire script in a text file, using the .mel extension, in
your Maya Scripts directory, then source it through Maya's Script
Editor. Alternately, you can copy and paste the entire script into
Maya's Script Editor.
*/

global proc cpk(int $chainCol, int $xCol, int $elemCol) {

    // Start of cpk()...
```

You need not specify the Y- and Z-coordinate columns in the procedure call since they always follow the X-coordinate. For instance, if $xcol = 7, we can count on the Y and Z columns being 8 and 9, respectively.

We have not closed the curly brackets "{" here because **cpk()** continues below.

Next you'll declare all variables and clear those whose values should be reset each time the script is run. Definitions of key variables are commented in the code below. The others will be explained as they are used in the script.

```
    /***** DECLARE THE VARIABLES *****/
    /*
    $filename       A string that stores the name of the PDB file
                    being read.
    $molNames[]     A list of objects named "molecule* ".
    $chains[]       The chain name for each line of the PDB file.
    $chain          A single array element within $chains[].
    $atom           The name of the current atom.
    $elements[]     The element name for each PDB file line.
    $element        A single array element within $elements[].
    $word           The return value of the fgetword command.
    $letters[]      A list of letters: A, B, C, etc.
    $letter         Each element in the array $letters[].
    $group          The node name for a group of like elements
                    belonging to a given chain if that node exists.
                    For example: $group = "oxygenGroupA".
```

Reminder: small and capital versions of the same letter are different characters as far as MEL is concerned. Keep this in mind when you name, assign, and query variables.

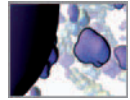
```
$groupName          The name of a group node yet to be created.
$newNodeName        An empty transform node used to group all chains.
*/
string $filename,  $molNames[], $chains[], $chain, $atom,
                    $elements[];
string $element,   $word, $letters[], $letter, $group[],
                    $groupName, $newNodeName;

/*
$xyz[]              The XYZ coordinates for each PDB file line.
*/
vector $xyz[] ;

/*
$x, $y, and $z      Coordinate values read from the PDB file.
$rad                The van der Waal's radius a given element.
$cpkColor[]         RGB color values used to set the element
                    shader colors.
$xyzArray[]         Stores the vector $xyz as an array.
*/
float $x, $y, $z, $rad, $cpkColor[], $xyzArray[];

/*
$molNum             The size of $molNames--the number of objects
                    named "molecule*" in your scene.
$fileId             The index number of the file opened by the
                    fopen command.
$i and $j           Counters.
$lineNumIndex for the variable arrays including $xyz[].
*/
int $molNum, $fileId, $i, $j, $lineNum;

/*
$molNumStr          The number of existing nodes called molecule
                    converted to a string for the purpose of
                    naming objects.

*/
global string $molNumStr;
```

The `clear` command (below) empties an array, setting its size to zero. This is good practice if you plan to run the script in succession for different molecules. Setting the counters `$i` and `$j` equal to zero is good form, although not entirely necessary since they will be initialized when they are used subsequently.

```
/***** INITIALIZE THE VARIABLES *****/
$i = $j = $lineNum = 0 ;
clear $xyz;
clear $elements;
clear $chains;
```

Check for other molecules in the scene

Before querying the PDB file, the script will take quick stock of what items, if any, are in the current Maya scene. The requirement for unique node names can lead to trouble if you run `cpk.mel` more than once in one scene file. As the script creates and



First execution	Second execution	Third execution
molecule0	molecule1	molecule2
chain_A0	chain_A1	chain_A2
carbonGroupA0	carbonGroupA1	carbonGroupA2
carbon	carbon	carbon
carbon1	carbon2	carbon3
hydrogenGroupA0	hydrogenGroupA1	hydrogenGroupA2
hydrogen	hydrogen	hydrogen
hydrogen1	hydrogen6	hydrogen11
hydrogen2	hydrogen7	hydrogen12
hydrogen3	hydrogen8	hydrogen13
hydrogen4	hydrogen9	hydrogen14
hydrogen5	hydrogen10	hydrogen15
oxygenGroupA0	oxygenGroupA1	oxygenGroupA2
oxygen	oxygen	oxygen

FIGURE 14.10

Objects must have unique names to allow for multiple molecules to be created in one scene file. In this example, the finished MEL script was run three consecutive times in a single Maya scene. It called the same data file each time: ethanol (C_2H_6O).

groups objects, it could encounter more than one Maya node with the same name, then falter. To prevent this, the script must account for existing objects and take appropriate action. Counting the number of existing *molecules* (the top node in the hierarchy created by `cpk.mel`) at the start of the script allows you to specify the *number* of the *molecule* being created in the current execution. This number will in fact be equal the number of existing molecules because we begin counting at 0. For example, if no molecules currently exist in the scene, the script will create a node called `molecule0`. Figure 14.10 shows your hierarchical naming structure applied to three molecules that were created with subsequent executions of `cpk.mel` within one scene file. To count the number of *molecules* in the scene, you will use the `ls` command to list objects named `molecule*` and then count the number of objects in the list using the `size` command. That count is then converted to a string and stored in `$molNumStr` for the purpose of naming the `molecule`, `chain`, and `elementGroup` nodes later on.

Reminder: When used with the **ls** command, an asterisk, *****, indicates "beginning or ending with this". For example, a list of comprised of "atom*" will return all Maya nodes whose name begins with "atom". Alternately a list of "*atom" will return all nodes ending in "atom".

```
// Get a list of objects called molecule*.
$molNames = `ls -tr "molecule*"`;
$molNum = size($molNames);
// convert $molNum to a string.
$molNumStr = $molNum;
```

Open the PDB file

In *Chapter 13* we introduced a group of MEL commands used to read and write files. Here you'll use the `fileDialog` command to open a window and allow the user to browse for a file.

```
// Open the PDB file.
$filename = `fileDialog -directoryMask "*.pdb"`;
```

The `directoryMask` flag allows you to specify the directory and must contain a file-type specifier such as "`pdb`". The asterisk (*) on its own will return all file types residing

in the directory. If no directory is specified the current directory will be used. The fopen command returns an integer we'll call $fileId, which you will use subsequently to query data from the PDB file. The next line of code is:

```
/*
fopen opens the file for Maya to read. The command returns an
index number to $fileId which you'll use subsequently to refer
to the file within Maya.
*/
$fileId = `fopen $filename "r"`;
```

Error checking

There is always a chance that a user will choose *Cancel* in the file browser window after executing cpk(). Without a contingency for this event, Maya will attempt to execute the rest of the script and wind up in a infinite loop, from which the only recovery is to force quit the application—not a good option because it means crashing Maya and losing unsaved work. Therefore, in the event that the user hits *Cancel*, you can use the error command to stop the execution of a script—in this case, the balance of cpk.mel. When called, error displays a message in the Command Line and the Script Editor, and returns control of the Maya scene to the user. It can be embedded in a conditional statement so that it's called upon when a problematic situation arises.

```
// Stop the script if no file is selected.
if ($filename == "") error "No file selected. Please run cpk()
  again";
```

After hitting *Cancel* in the file browser, the user would simply run cpk() again if they wished to continue building a molecule.

Main loop

Now let's create the loop that reads and stores the essential data. The condition for remaining in the loop is not having reached the end of the PDB file. Therefore, you'll use the feof command to check each time through the loop whether the file end has been reached. If feof returns **1**, the file end has been reached. If it returns **0**, the file end has not been reached.

```
/***** MAIN LOOP *****/
while (`feof $fileId` == 0) {
    // A non-zero `feof $fileId` value means the file end has
      been reached.
```

Record type

As long as you haven't reached the end of the file, it's okay to read the first word of the current line in the file. The fgetword command reads words, that is, strings of characters separated by space or tab characters. You will store the return value of fgetword in a string variable called $word. Each time the command is used, Maya advances to the next word in the file. The first time fgetword is used on each new line in the PDB file, it returns the first word—the record type. If the record is "ATOM" then the script should proceed to read and store the data. The following lines begin the while loop you opened.

The variable **$word** is used frequently during this part of the script, which is responsible for reading the PDB file data. $word is a container used to temporarily hold each string returned by the **fgetword** command.

```
    $word = `fgetword $fileId`;
    if ($word == "ATOM") {
        // Ready to read and store the PDB record...
```

Case	Column 1	Column 2	Column 3
1	A	X	E
2	A	E	X
3	X	A	E
4	X	E	A
5	E	A	X
6	E	X	A
A = chain; X = x-coordinate; E = element.			

TABLE 14.02

Cases for the possible order of **chain**, **x-coordinate**, and **element** columns in a PDB data file. These cases must be considered when reading data into variables in your MEL script.

Read the record

Here you read in the chain, x-, y-, and z-coordinate, plus the element for the ATOM entry. The column in which each data appears is stored in the variables, `$chainCol`, `$xCol`, and `$elemCol`, which were supplied by the user as arguments in the procedure call. In preparing the code, don't lose track of the rule that on each line of a PDB-formatted file, each column is represented by a word. You will use `fgetword` to increment from word to word (i.e. from column to column) until you reach the column corresponding to the number stored in `$chainCol`, `$xCol`, or `$elemCol`. At that point, you will store the value of `fgetword` in the temporary holder `$word`, which is in turn used to assign `$chain`, `$x`, or `$element`.

Note that, although you know the column numbers for the entries we're interested in, you don't explicitly know their order from left to right. These can vary from one PDB file to another. Three columns give you six cases for possible orders (show in Table 14.02). Therefore you must test for each case, and read in the data accordingly.

The flowchart in Figure 14.11 shows the steps you'll take in determining the order of data columns and then reading the data into your three array variables, `$chains[]`, `$xyz[]`, and `$elements[]` (via the variables `$word`, `$chain`, `$x`, and `$element`). Let's look at the MEL code for Case 1, which immediately follows the `if ($word=="ATOM")` statement above.

For concision, the code for Cases 2 through 6 are not included here in the text. You'll find the entire code listing in the MEL script, cpk.mel on the CD-ROM.

```
// Determine order of columns and read in data.

// CASE 1.
if ($chainCol < $xCol && $xCol < $elemCol) { // CASE 1.
    // Chain column.
    for ($i = 1; $i < $chainCol; $i++) {

        // This increments fgetword until the chain column is
          reached.
        $word =`fgetword $fileId`;
     }
     $chain = $word;
```

record is of type "ATOM"

repeat these steps for every "ATOM" record in the PDB file ($lineNum += 1)

Case 1? → no → Case 2? → no → etc...

yes → etc...

yes

Use fgetword to increment to the **chain** column.

$chains[$lineNum] = fgetword

Use fgetword to increment to the **x-coord** column.

$x = fgetword
– *advance one column* –
$y = fgetword
– *advance one column* –
$z = fgetword

$xyz[$lineNum] = $x, $y, $z

Store coordinates together in one vector array.

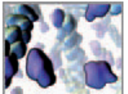
Use fgetword to increment to the **element** column.

$elements[$lineNum] = fgetword

make, position, and shade spheres

FIGURE 14.11

The steps involved in reading a line from a PDB file using Maya's file-reading commands. There are six possible cases for the order in which the data columns for **chain ID**, **x-coordinate**, and **element** appear.




```
            $j = $i; // Continue column counter where it left off.

            // X column.
            for ($i = $j; $i < $xCol; $i++) {
                $word = `fgetword $fileId`;
            }
            $x = $word;
            $y = `fgetword $fileId`; $i += 1;
            $z = `fgetword $fileId`; $i += 1;

            $j = $i;

            // Element column.
            for ($i = $j; $i < $elemCol; $i++) {
                $word = `fgetword $fileId`;
            }
            $element = $word;

        } // End CASE 1 if statement.
```

The same approach applies to the remaining five cases, the only difference being the order in which the variables, `$chain`, (`$x`, `$y` and `$z`), and `$element` are assigned.

Now let's assign values to the array variables `$chains[]`, `$elements[]`, and `$xyz[]` for the current line in the PDB file. When you assign string data to a numerical type variable, Maya converts the data into numerical values. This feature allows us to turn the string data stored in `$x`, `$y`, and `$z` into floating point numbers to use for positioning the atoms. Furthermore, since `$xyz[]` is a vector array, we must pass it the `$x`, `$y`, and `$z` values in vector form, using the `<< >>` characters.

```
        // Chain ID.
        $chains[$lineNum] = $chain; // String.

        // X, Y, Z coordinates.
        $xyz[$lineNum] = <<$x, $y, $z>>; // Vector.

        // Element name.
        $elements[$lineNum] = $element ; // String.
```

Next, we increment the `$lineNum` counter, close the `if` "ATOM" statement, update the `$fileEnd` variable, and close the main `while` loop.

```
        // Only count lines starting with "ATOM".
        $lineNum += 1;
    } // End if ($word == "ATOM").

    // Advance to the next line in the PDB file.
    fgetline $fileId;

} // End while loop.
```

Create the atoms

Here is where you will make, position, and shade the spheres which represent atoms in your CPK model. The flowchart in Figure 14.12 illustrates the steps involved.

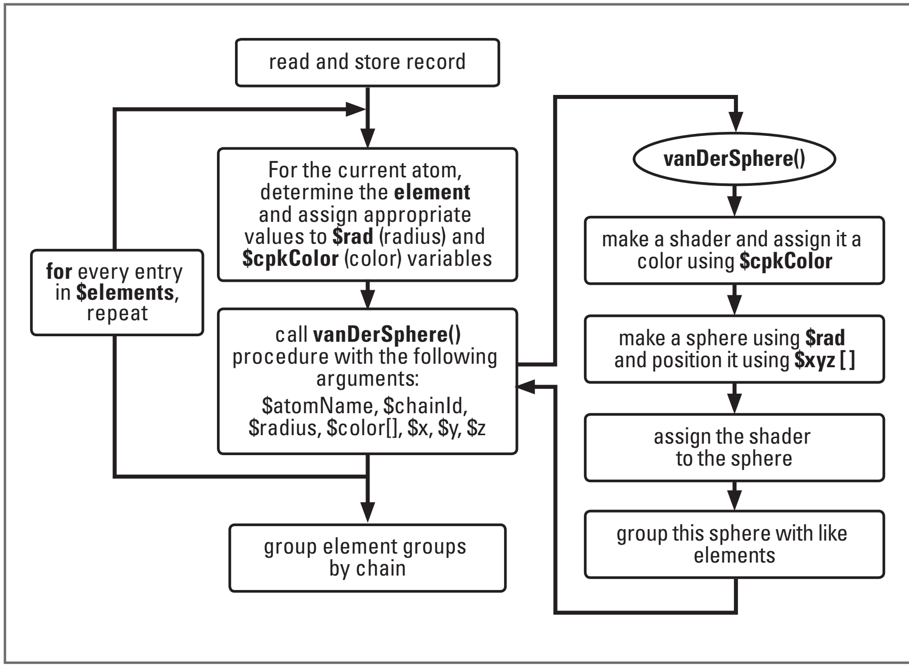


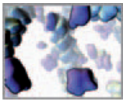
FIGURE 14.12

The steps involved in making the atoms from the PDB file data.

The script increments through the entries in the `$elements[]` array using a `for...in` loop. Within this loop, a series of nested `if...else` statements query the element type. The type of element in turn determines the atom's radius and color. The radius, color, element, chain ID, and XYZ coordinates are used as arguments to call the second procedure in your script, `vanDerSphere()`. `vanDerSphere()` makes a sphere and a shader, positions the sphere, assigns the shader to the sphere, and groups the sphere with like atoms. Parceling these tasks in a second procedure, or subroutine, means they need only be included once in the main body of the script rather than in every place they're needed. The code below continues the main script, following the previous code line which closed the `while` loop.

```
// initialize $i, the counter in the following loop.
$i = 0;

for ($atom in $elements) {
    // Put the xyz vector into an array for use below.
    $xyzArray = $xyz[$i];
    $chain = $chains[$i];

    // CARBON.
    if ($atom == "C") {
      $rad = 1.70; // van der Waals radius for carbon.
      $cpkColor = {0.35, 0.35, 0.35}; // Gray, the CPK color for
        carbon.
      // Call procedure.
      vanDerSphere("carbon", $chain, $rad, $cpkColor, $xyzArray[0],
        $xyzArray[1], $xyzArray[2]);
```

You may recall from *Chapter 12*, that for...in loops cycle through elements in an array. The "for" part is a holder for the element value, while the "in" part is the array itself.




```
            }
            // HYDROGEN.
            else if ($atom == "H") {
                $rad = 1.20;
                $cpkColor = {1.0, 1.0, 1.0}; // White.
                vanDerSphere("hydrogen", $chain, $rad, $cpkColor,
                  $xyzArray[0], $xyzArray[1], $xyzArray[2]);
            }
            // NITROGEN.
            else if ($atom == "N") {
                $rad = 1.55;
                $cpkColor = {0.0, 0.5, 1.0}; // Blue.
                vanDerSphere("nitrogen", $chain, $rad, $cpkColor,
                  $xyzArray[0], $xyzArray[1], $xyzArray[2]);
            }
            // OXYGEN.
            else if ($atom == "O") {
                $rad = 1.52;
                $cpkColor = {1.0, 0.0, 0.0}; // Red.
                vanDerSphere("oxygen", $chain, $rad, $cpkColor,
                  $xyzArray[0], $xyzArray[1], $xyzArray[2]);
            }
            // PHOSPHORUS.
            else if ($atom == "P") {
                $rad = 1.80;
                $color = {1.0, 0.5, 0.0}; // Orange.
                vanDerSphere("phosphorus", $chain, $rad, $color,
                  $xyzArray[0], $xyzArray[1], $xyzArray[2]);
            }
            // SULFUR.
            else if ($atom == "S") {
                $rad = 1.80;
                $cpkColor = {1.0, 1.0, 0.0}; // Yellow.
                vanDerSphere("sulfur", $chain, $rad, $cpkColor,
                  $xyzArray[0], $xyzArray[1], $xyzArray[2]);
            }
            // Add more elements if you like.
            // increment the for...in loop counter.
            $i += 1;
        } // End for loop.
```

In the script excerpt above, we have accounted for the five elements which occur in chain A of the actin data. The file on CD-ROM, `cpk.mel`, includes `else...if` statements for additional elements that occur in many proteins: chlorine, fluorine, iron, phosphorus. More elements can easily be added. If your script is missing an element that is present in a PDB file you're reading, those atoms simply won't be created when you run `cpk()`.

Because `vanDerSphere()` is a separate procedure, you'll want to wrap up your main `cpk()` procedure before moving onto this atom-building subroutine.

Organize the scene hierarchy

cpk() ends with a for...in loop that organizes the models into the hierarchy described back on page 358. It collects like elements together under group nodes called chain_A*, chain_B*, etc., and then collects these chains under a group node called molecule*. In the Maya scene hierarchy, group nodes are parents and group members, their children. In the code that follows, you will use the string array variable, $group[] to hold a lists of node names for groups of like elements within a given chain. These groups will have been created in vanDerSphere() which you will see in a minute. As an example, the data file 1j6z.pdb, produces the groups carbonGroupA, hydrogenGroupA, nitrogenGroupA, and oxygenGroupA.

```
// Group atoms by chain ID (add more letters for more chains).
$letters[] = {"A", "B", "C", "D", "E", "F", "G", "H"};

for ($letter in $letters) {

   // Ensure $group is empty at the start of each loop.
   clear ($group);

   // Create a list of element groups within the current molecule.
   string $tmpStr = "*Group" + $letter + $molNumStr;
   // $molNumStr is the number of the current molecule.
   $group = `ls -transforms $tmpStr`; // e.g. carbonGroupA, etc...

   if ($group[0] != "") {
        // $group is not empty, therefore this chain exists.
        $groupName = "chain_" + $letter + $molNumStr; // E.g.
          chain_A1.
        createNode transform -shared -name $groupName;
```

The script checks to see if there exists at least one **chain** in your scene, using the statement, `if ($group[0] != "")`. It does this before using the chain object in parenting (grouping) operations. It's necessary to check because some PDB files do not include chain IDs.

Starting with "|" ensures that the element group gets parented to the newly created group node and not to an existing one that is parented under a molecule ($groupName) that already exists in your scene.

```
        parent $group ("|" + $groupName);
    } // End if.

} // End for.

// Parent all chains under a group node called molecule*.
$tmpstr = "|chain*" + $molNumStr;
$group = `ls -transforms $tmpstr`; // a list of nodes called
  chain*.

if ($group[0] != "") // There exists at least one chain*.
{
     $newNodeName = `createNode transform -name ("molecule" +
       $molNumStr)`;
     // parent chain* nodes to molecule*;
     parent $group $groupName;
}

} // End cpk() procedure.
```

That wraps up the main procedure. Now let's have a look at the code that creates and positions the atoms, which are represented as NURBS spheres:

vanDerSphere() procedure

This procedure takes seven arguments and has no return value. It follows the steps outlined in Figure 14.12. Up to this point, you have created shaders and assigned them to models using the Hypergraph, as described back in *Chapter 8*. In this procedure you will use the MEL command `shadingNode` to make a shader and then the `hyperShade` command with its `-assign` flag to connect the shader to the appropriate geometry. You'll use Blinn shaders to achieve the semi-gloss appearance of the traditional hard plastic CPK models.

You'll build this procedure in a new file, separate from `cpk.mel`, and save it under the name vanDerSphere.mel (capital letters are optional for file names).

```
/***** vanDerSphere.mel *****/
/*
Created: February 2006, modified August 2007 .
Authors: Jason Sharpe, Charles Lumsden, Nick Woolridge.

Description:
This procedure creates and shades a sphere to represent the van der
Waals contanct surface of a particular element (oxygen, carbon,
etc.), based on arguments send to it from the cpk() procedure.

The procedure arguments are as follows:
$atomName       The name of the current atom. For example: carbon1.
$chainId        The PDB file chain letter (A, B, C, etc.) and is
                used in naming the group node to which the atom will
                belong.
$radius         The vdW radius of the atom.
$color[]        The name of the shader node to be assigned to the
                atom.
$x, $y, and $z  The world space coordinates for the atom.

To use this script:
Save the script in a text file, using the .mel extension, in your
Maya Scripts directory, then source it through Maya's Script Editor.
*/

global proc vanDerSphere (string $atomName, string $chainId,
  float $radius, float $color[], float $x, float $y, float $z) {

    /***** DECLARE THE VARIABLES *****/

    /*
    $molNumStr      Was assigned in the cpk() procedure and is used
                    for naming when more than one whole molecule
                    exists in your scene.
    */
    global string $molNumStr;
```

```
/*
$shaderName        The name of the shader to be assigned to the
                   current atom.
$groupName         The name of the element group to which the
                   current atom will be parented.
$newNodeName       The transform node name of the shader,
                   $shaderName.
$atomNodeName[]    The transform and history node names
                   returned by the sphere command.
*/
string $shaderName, $groupName, $newNodeName, $atomNodeName[];

/***** INITIALIZE THE VARIABLES *****/

$shaderName = $atomName + "Shader";
$groupName = $atomName + "Group" + $chainId + $molNumStr;
```

Below, the shadingNode command is used with the -shared flag to ensure that only one shader is created for each element. We set the shader diffuse value to 0.9 to brighten it up slightly from the default setting of 0.8. After your model is built you can try different RGB and diffuse settings for each of the shaders.

```
// Create a shader and store its name in $newNodeName.
$newNodeName = `shadingNode -asShader -shared blinn -name
  $shaderName`;

// Set the shader's color and diffuse attributes.
setAttr ($newNodeName + ".color") $color[0] $color[1] $color[2];
setAttr ($newNodeName + ".diffuse") 0.9;
```

Next you'll make and position the NURBS sphere. The sphere command returns a string array of size 2, the first element (index 0) of which is the transform node name. After the sphere is made, it remains selected. The hyperShade command then assigns it the appropriate shader.

```
$atomNodeName = `sphere -r $radius -n $atomName`;
// Position the sphere.
move -worldSpace $x $y $z $atomNodeName[0];

// Assign the shader.
hyperShade -assign $shaderName ;
```

Again, you'll use the -shared flag, but this time with the createNode command to ensure that only one group node is created corresponding to this type of element, chain ID, and molecule.

```
    // Create a group node to hold this type of atom.
    createNode transform -shared -name $groupName;;
    parent $atomNodeName[0] $groupName;

} // End procedure.
```

Save the entire script—including both procedures—under the name cpk.mel in the Maya Scripts directory on your hard drive.

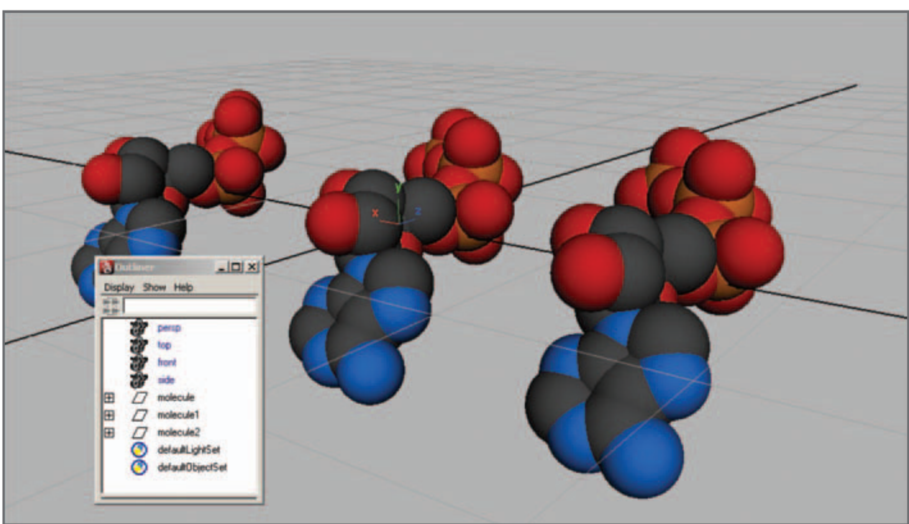


FIGURE 14.13

The cpk() procedure was run three times to create these CPK models of an ATP molecule.

Results: Running the script

The ATP model

Now let's try out the script on ATP (Figure 14.13).

Source the script

1. **Open the Script Editor**
2. **Choose File → Source Script**
3. **In the file browser, navigate to and select your script file,** cpk.mel.
4. **Press Open.**

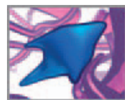
This loads cpk.mel into memory so that the procedures `cpk()` and `vanDerSphere()` can be called from within Maya.

The character in column 3 is the Atom Name, a data type that is distinct from the Element, displayed in column 12. For ATP they happed to be the same character. This is not usually the case; Atom Names are often 2 to 3 character long, whereas Elements are always 1 character in length. Atom Names are used by MolVis applications in determining connectivity between atoms.

Examine the PDB file

Next, open the PDB file `atp.pdb` in your text editor and examine an ATOM entry (see below). The column numbers corresponding to *chain*, *X-coordinate*, and *element* are 5, 7, and 12, respectively. Therefore, these will be the arguments sent to the main procedure, `cpk()`.

					chain		X-coordinate					element
Column:	1	2	3	4	5	6	7	8	9	10	11	12
Data:	ATOM	10	C	ATP	A	1	0.291	-2.472	-5.311	1.00	0.00	C



Run the script

1. **Enter** cpk(5, 7, 12); **in the Command Line or Script Editor.**
2. **When the file browser window appears, locate** atp.pdb **file in your project PDB directory (CPK_Project/PDB/atp.pdb).**

Given the small size of atp.pdb, the script should execute quickly. When it's done, have a look at the various elements in your scene, including the shaders you created. Note that the organization of elements into groups makes it ease to assign a different shader to carbon, for example, by assigning the shader to the entire group. Alternately, you could change the color attribute in the carbon shader, which is already connected to all the carbon atoms in the scene.

Make a Shelf button

Placing the procedure call, cpk(5, 7, 12), on your Custom Shelf enables you to execute the script at the push of a button. The button will work as long Maya can *locate* a procedure called cpk, which will happen under one or both of the following conditions:

A. **Since starting Maya you have loaded the procedure by sourcing its parent script cpk.mel, or by copy>paste>Entering cpk() in the Script Editor.**

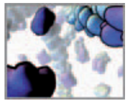
B. **The script cpk.mel has been placed in a directory that is listed in Maya's Scripts path; notably, the default Scripts directory, which by default in Windows is:**

C:\Documents and Settings\User\My Documents\maya\8.5\scripts

When you *call* a procedure, if it isn't currently in memory, Maya will search the .mel files within its Scripts path until it locates a procedure named cpk. Furthermore, when the procedure vanDerSphere() is called from within cpk(), Maya will again automatically search for and load it. To create the cpk() button:

1. **Make sure Shelves are displayed and that you have created a custom shelf already (see page 86 in *Chapter 04: Maya basics*).**
2. **Type** cpk(5, 7, 12) **in the Script Editor.**
3. **Select the text** cpk(5, 7, 12) **then MMB+Drag it to your Custom shelf (**Figure 14.14a**). A new MEL icon will appear.**
4. **Choose Window → Settings/Preferences → Shelves, and click the Shelves tab. This opens the Shelves Editor.**
5. **Under the Shelf Contents tab, select** cpk(5, 7, 12)**. In the Icon field, type** cpk (Figure 14.14b)**.**
6. **Hit Save All Shelves.**

You can now execute cpk(5, 7, 12) simply by pressing the cpk Shelf button. Keep in mind that this button will only work with PDB files that have the "5, 7, 12" column arrangement. For files with different arrangements, you can source cpk.mel, and then enter appropriate arguments in the cpk() procedure call.



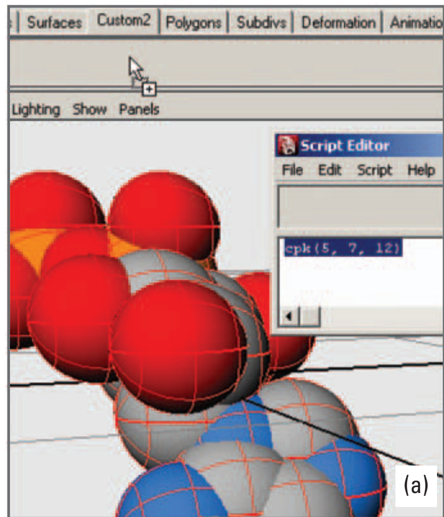


FIGURE 14.14

(a) Drag the procedure call, cpk(5, 7, 12) to your custom shelf to make it a button.

(b) Locate cpk() in the Shelves Editor and give it an Icon Name so that you can easily recognize it on the Shelf.

Create an actin protein model

Now at last you're ready. Use the cpk Shelf button created above to make your actin model. If you wish, save your ATP and create a new scene, although this is not necessary since the script facilitates multiple molecules within one scene. To create the actin model:

1. **Press the** cpk **button that you created in your Custom Shelf.**
2. **When the browser window appears locate the file** 1j6z.pdb **and hit Enter.**

That's it! With nearly 100× more atoms, actin will naturally take longer to build than the ATP model. While the script is running, Maya will be unresponsive to any other commands, menu selections, or view changes. It will seem as if the application has crashed—but it hasn't, it's just busy. Unfortunately there is by default no ticking clock or scroll bar to indicate progress, you just have to be patient. Figure 14.15 shows the completed CPK actin model in the scene view.

Applying the script to other molecules

You may now want to try out cpk.mel on other molecules. PDB files for tens of thousands of biomolecules can be downloaded from the RSCG PDB website free of charge:

http://www.pdb.org/pdb/

Use of the PDB archive is subject to conditions listed at:

http://www.rcsb.org/pdb/static.do?p=general_information/about_pdb/pdb_advisory.html

Please keep in mind, however, not all PDB lay their data out exactly alike. The number of columns and their order can be inconsistent from file to file. Before running cpk.mel on a PDB file, remember to open the file in a text editor and determine the column numbers for the chain, the x-coordinate, and the element; then enter those column numbers as arguments in the cpk() procedure. Strange results, or none at all, usually indicate a misinterpretation of the column numbers.

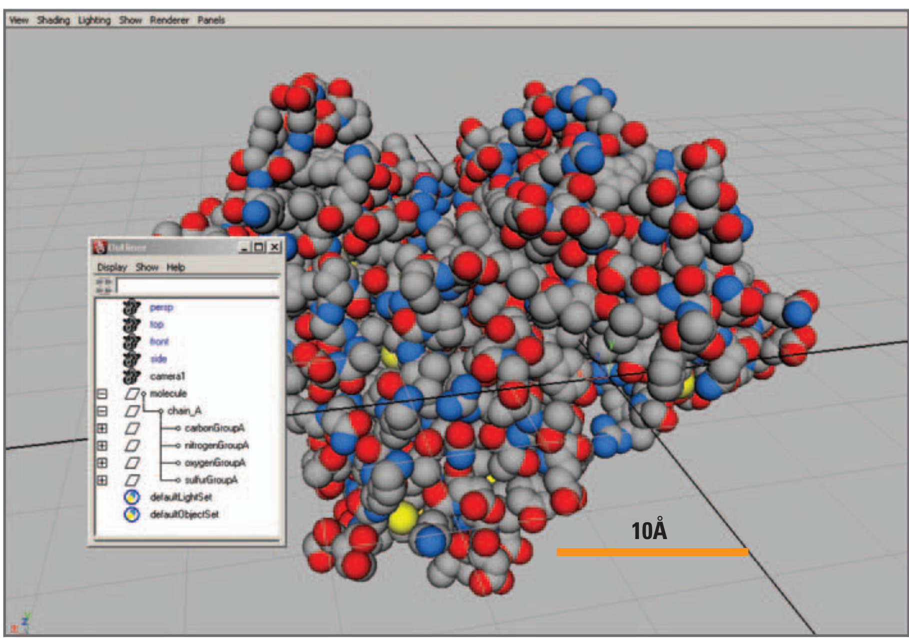


FIGURE 14.15

Finished CPK model of actin. Note the group hierarchy in the Outliner. Scale bar = 10 Å.

Debugging the script

In the unlikely event that you encountered no errors when you sourced and ran the script, skip this section. If, on the other had, you transcribed the code piece by piece from this chapter or rewrote it yourself by following the written instructions, chances are you encountered syntax errors at some point.

Syntax errors

Syntax errors will appear as usual in the Command Line and Script Editor. Line and column numbers will point you to the error location in the script file `cpk.mel`. For example, the following error was caused by a mistyped variable name, $xYz instead of $xyz. The `clear` command empties an array variable, setting its size to zero.

```
// Error: clear $xYz; //
// Error: Line 70.11: "$xYz" is an undeclared variable. //
```

The mistake can be found and corrected in the original script file using the line number, `70`, and column number, `11`, provided in the error message.

Logic errors

Logic errors can be dealt with in two ways. The first, and quickest, is to compare your script, line for line, with the file, `cpk.mel`, included on the CD-ROM, and look for discrepancies. The second, and more informative, is to print to the Script Editor, variables from the part(s) of the script that you suspect is causing the trouble. This will

help you to see the difference between what you think Maya is doing and what is actually happening. You do this using the `print` command.

```
int $i;
for ($i = 0; $i<3; $i++) {
     string $name = "molecule"+$i;
     print ("$i = " + $i + ", $name = " + $name + "\n");
}
```

The example above reports the variables, $i and $name, in the Script Editor, as follows.

```
$i = 0, $name = molecule0
$i = 1, $name = molecule1
$i = 2, $name = molecule2
```

Tracing values in this way can help locate mistakes in variable assignment that may be causing logic errors.

Results: Rendering your molecule

In visualization, it's the look that counts. To conclude this project, let's create a rendering of the actin molecule as it appears on the title page of this chapter. You'll start by repositioning the model, then add a camera and lights, and finally render it using the Maya Software Renderer.

Reposition the model

If you select the parent node, `molecule0`, in the Outliner and show its move, rotate, or scale handles, you'll see that its origin is at the world origin (0, 0, 0) and not at the model's center. A centered origin will make positioning the model for rendering easier and more intuitive. Also, for other work you may need to position multiple molecules relative one another in order to make them interact dynamically, as in *Case Study 3* just ahead. In this case it helps to have the default rotation values (0, 0, 0) correspond to a preferred orientation for the model. In other words it may be helpful to consider the model in terms of a convenient "right side up" and "front and back". Any required deviation from this could be handled as a change from rotation values of 0, 0, 0.

Actin has a distinct shape, which is closely linked to its function (Figure 14.05). For didactic purposes, actin is often illustrated with its pointed end up and one of the wide faces toward the viewer.

To center the origin and reorient the model:

1. **Select** `molecule0` **in the Outliner.**
2. **Choose Modify → Center Pivot.**
3. **Hit "E" to show the rotation handles.**
4. **Rotate the molecule in the three orthographic views to look somewhat like the model in** Figure 14.16**. That is, pointed end up and wide face perpendicular to the positive z-axis.**

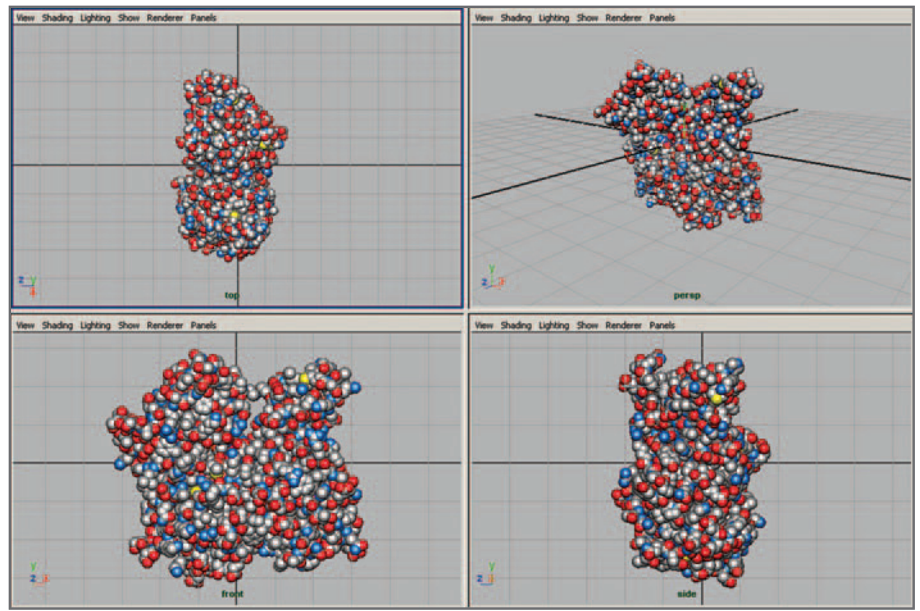


FIGURE 14.16

We used the default orthographic scene view to position and orient the actin model for rendering.

5. **Turn on "Snap to grids" and use the Move Tool to drag the model to the world origin.**

6. **Turn off "Snap to grids".**

7. **Choose Modify → Freeze Transformations. This will set all transform values to 0.**

The model is now "zeroed" in its default position.

Set up the camera

1. **Choose Create → Cameras → Camera**

2. **Select the new camera and open the Attribute Editor.**

3. **Set the camera's focal length to 50 (this is reasonably close to human vision).**

4. **Click on the transform node tab and rename your camera "renderCam".**

5. **From the Panel menu set, choose Panels → Look Through Selected.**

6. **Manipulate the camera using alt + mouse button (rotate, track, and dolly) to get a view you're satisfied with.**

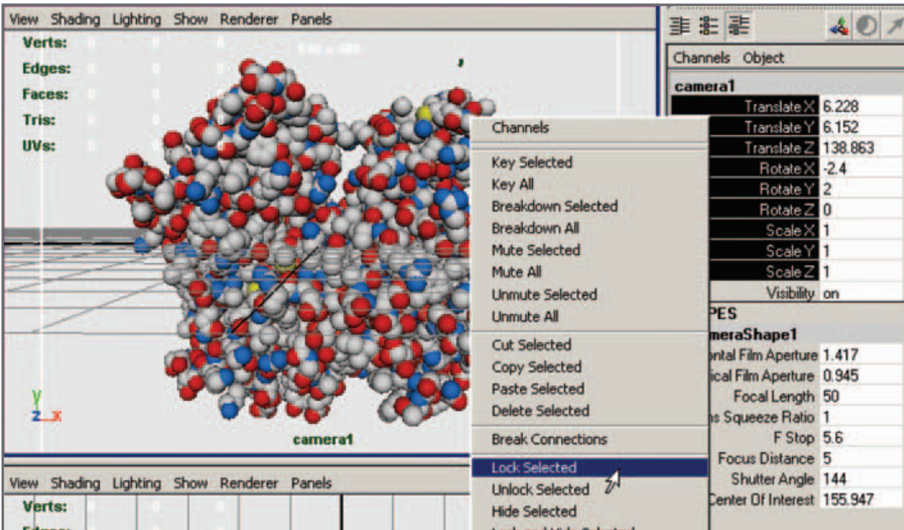


FIGURE 14.17

Once you are happy with the camera, lock its transforms so that you don't accidentally move it.

7. **Fix the renderCam's position by locking its transform attributes, so that you don't accidentally move it (**Figure 14.17**):**

   (a) **Select renderCam if it's not already selected.**

   (b) **In the Channel Editor: highlight the transform attributes; RMB to bring up the context-sensitive menu.**

   (c) **Choose Lock Selected.**

   Note: If you want to move the camera again, repeat steps (a) and (b), then choose Unlock Selected.

Background

Set the background color in the Camera settings:

1. **Select renderCam.**
2. **Open the Attribute Editor and select the shape node tab, e.g. renderCamShape.**
3. **Choose Environment, then click on the Background Color palette to launch the Color Chooser.**
4. **Adjust the color then hit Accept.**

Set up the workspace

It is helpful to have more than one view of the workspace when setting up a scene like this one. Figure 14.18 shows the two-panel view introduced on page 223 in *Chapter 09*. The left panel shows the persp view and the right, `renderCam`'s view.

Adjusting view panels

Maya Help → Using Maya → General → Basics → Basic menus → Panel menus → Panels

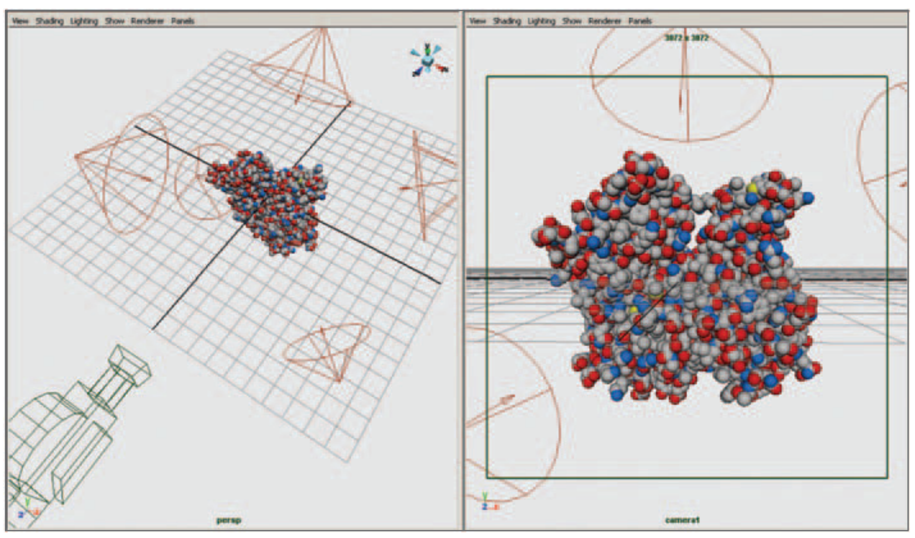


FIGURE 14.18

Multiple workspace panels are helpful when setting up a scene for rendering.

Set up the lights

Here we'd like you to explore the 3-point lighting setup we described in *Chapter 10*. Back then you used two Point lights and an Area light. Here you'll use Spotlights instead in order to get some practice setting up lights that have a directional bias (not to be confused with Directional lights).

Tip: We increased the size of our lights and camera by increasing their scaleX, scaleY, and scaleZ attribute values. This makes their wireframe icons easier to see in the scene view, without changing their functionality.

1. **Choose Create → Lights → Spotlight. This will be your key light.**
2. **Select the light and open the Attribute Editor (ctrl + A).**
3. **Rename the light as "keyLight".**
4. **The Cone and Penumbra Angles are used together control the edges of a spotlight. In this project you don't want the edge of the light to be visible. Instead you want to light the model all over for a smooth, even effect:**

   Set the Cone Angle to 90º or greater.
5. **In the persp workspace panel, choose Panels → Look Through Selected. This creates a non-rendering camera that is connected to the light, enabling you to interactively position the light.**
6. **Manipulate the light view, using the navigation controls (rotate, track, and dolly), to light the model from the upper-front-left direction.**
7. **Create two more lights and name them, fillLight, and backLight, respectively.**
8. **Set the fillLight Intensity Attribute to 0.5 to decrease the amount of light it emits.**
9. **Position fillLight and backLight as described in steps 4 and 5 above.**

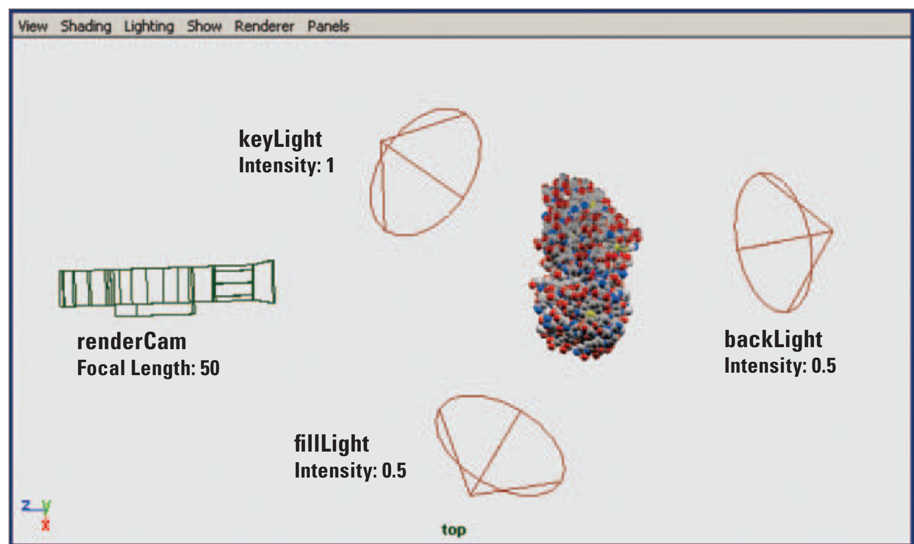


FIGURE 14.19

The lighting setup we started with.

In the next section, you will tune the positions and intensities of the three lights to get a pleasing result. Figure 14.19 shows the lighting rig we started with.

IPR preview

In addition to lighting changes, changes to shader colors are updated in the IPR preview as well.

Now use IPR (for Interactive Photorealistic Rendering) to tune the light positions and intensities.

1. **Open the Render View: choose Windows → Rendering Editors → Render View.**

2. **In the Render View, choose → IPR → IPR Render. Select the camera you want to render from. After a pause a message will appear at the bottom of the Render View, prompting you to "Select a region to begin tuning".**

3. **LMB + drag to select the entire picture (Figure 14.20). Maya will take a few seconds to load the pixels.**

4. **One by one adjust the positions, orientation, and intensities of your lights until the IPR image is satisfactory.**

Er ... make that a 6-point lighting rig

For some rendering situations three lights just won't cut it and you have to add more. After IPR previewing the lights, we decided to add two more spotlights, each with an Intensity of 0.5 in order to round out the back lighting. As well, because the preview was too dark overall, we added an ambient light, with an Intensity of 1.0 to brighten the scene up evenly. Our final lighting setup is shown in Figure 14.21.

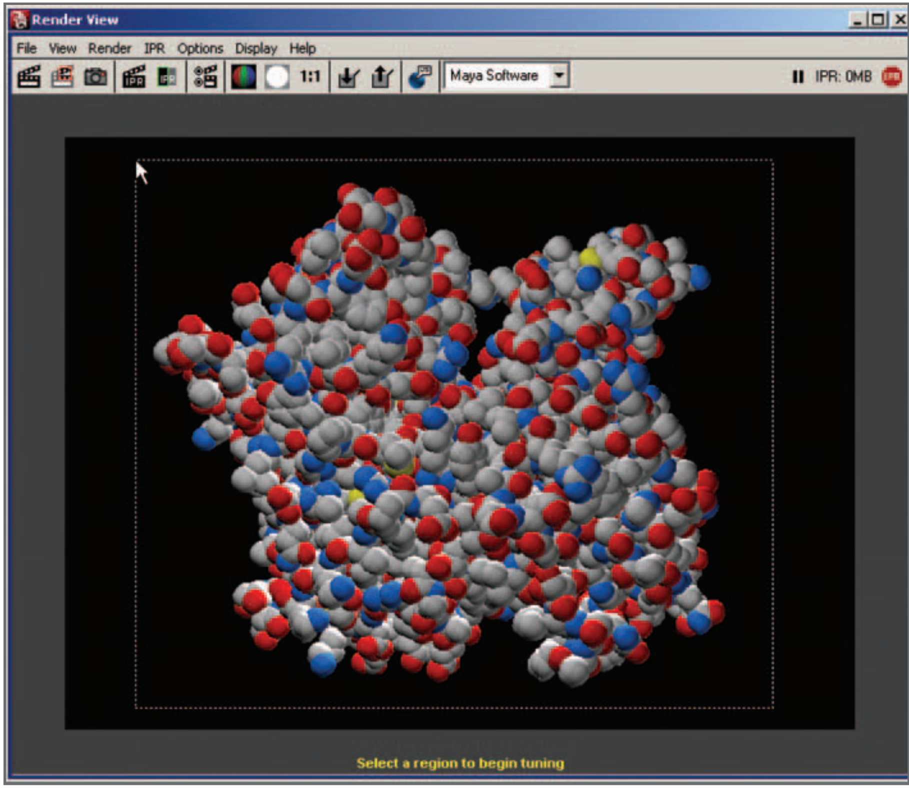


FIGURE 14.20

After choosing IPR, use your cursor to select the region that you want to update in the Render View as you tune lights and shaders.

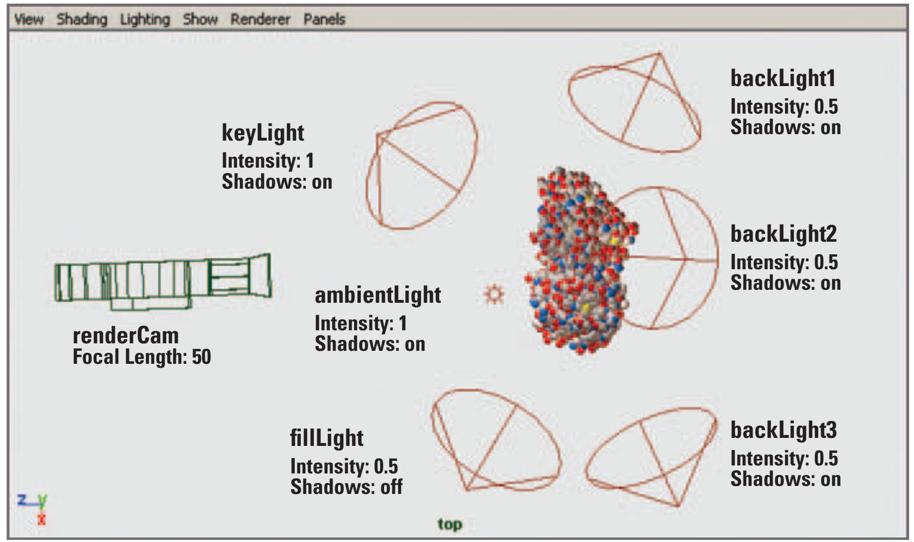
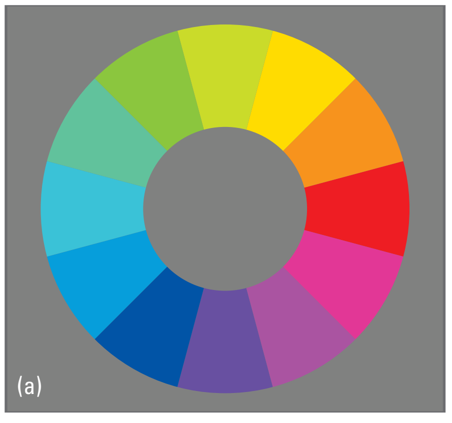


FIGURE 14.21

The final lighting setup that we used to render the title page image for this chapter.



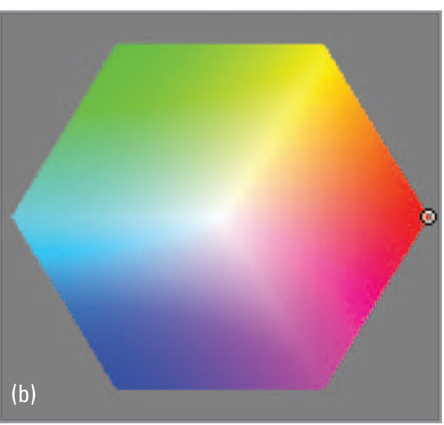


FIGURE 14.22

A color wheel is a circular plot of the spectrum of visible light. Plots like this are often employed in computer graphics programs for the purpose of choosing colors to apply to objects. Artists often discuss colors in terms of *warm* versus *cool*, which correlate to locations on the color wheel proximal to the *red* and *blue* ends of the spectrum, respectively (think *fire* and *ice*).

Color your lights

Colored lights can enhance the atmosphere of a rendering and aid the perception of depth. In very general terms, warm colors—toward the top-right of the color wheel shown in Figure 14.22—in a picture appear to advance and cool colors to recede. In combination, warm and cool lighting can set up a visual tension between warm and cool, foreground and background, in order accentuate the illusion of 3D space and add a dramatic edge to a rendering. The effect can be bold or subtle. We opted for the latter, just to give a hint of color to edges of the atoms that make up our actin molecule. The following are the HSV values we used to color our back and fill lights.

backLight Color	fillLight Color
H: 22.0	H: 180.0
S: 0.35	S: 0.2
V: 1.0	V: 1.0

Add Depth Map Shadows

We deal only with Depth Map Shadows here. They are generally less accurate than ray traced shadows but quicker to render. Maya Help covers ray traced shadows.

Although we now have a nicely lit picture of a lot of individual atoms, the image in Figure 14.20 portrays little of the molecule's striking variations in depth. This is because every atom is receiving the same amount of light. If, on the other hand, some atoms block light from hitting others, you'll get a better picture of how all the atoms relate to one another in space. This is a basic consequence of the interplay between light and shadow in nature.

By default, the lights you created are set not to cast shadows. By turning on Use Depth Map Shadows and adjusting a few attributes you get the image shown in Figure 14.23b. This provides a much better sense of the 3D form of the actin molecule than Figure 14.23a. Here are the steps we followed to activate the shadows:

1. **Select keyLight and open the Attribute Editor.**
2. **Under Shadows, check Use Depth Map Shadows.**
3. **Set Filter Size to 3, but leave all other settings at their default values.**

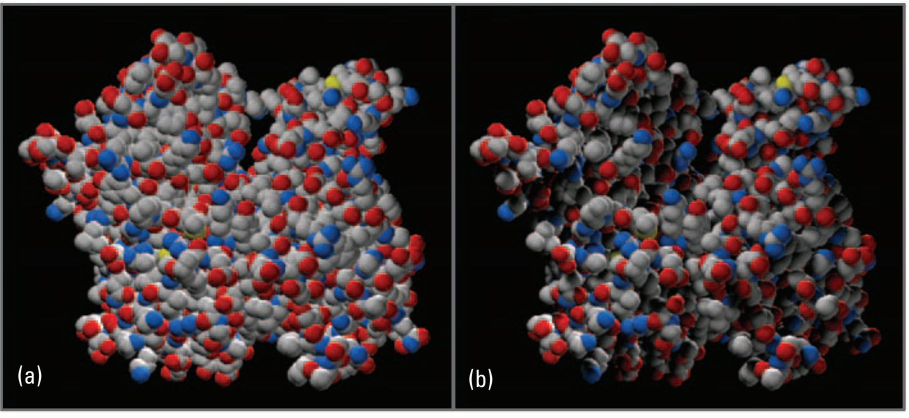


FIGURE 14.23

IPR preview renderings with:
(a) No shadows.
(b) Depth Map Shadows turned on.

Depth Map Filter Size affects shadow softness: The greater the filter size, the softer the shadow. We turned on Depth Map Shadows for the back lights, but left them off for the fill light. We wanted the back light to help separate the edges of the molecule from the dark background. The fill light, on the other hand, helped define the forms of the atoms that lie in the shadow of the key light. In our experience, good use of lights and shadows is often a matter of trial and error; start with a standard 3-point setup, then experiment to get the best results for a given purpose.

Shadows

Maya Help → Using Maya → Rendering and Render Setup → Lighting → Basics of Lighting → Shadow → Shadow in Maya

Set up the rendering

You will make the image using the Maya Software Renderer, which we introduced back in *Chapter 11*. IPR will help you tune the lights. First, adjust the Render Settings.

Render Settings

1. **Choose Rendering Editors → Render Settings.**
2. **Make sure "Render Using" is set to Maya Software.**
3. **Under the Common tab, set the file name, images format, and so on. Below are the settings we used:**

Common tab

Image File Output

File Name Prefix	cpk_model
Frame/Animation Ext	name.ext (Single Frame)
Image Format	Tiff (tif)
Camera	renderCam

Remember: the higher the Antialiasing quality, the longer the render time.

Image Size

Preset	640x480

Maya Software tab

Anti-aliasing Quality

Quality	Production Quality
Edge Anti-aliasing	Highest Quality

4. **Hit Close.**

Hit Render!

When you're happy with the lighting and colors, render your scene and save out an image file:

1. **In the Render View, choose → Render → renderCam.**
2. **When Maya has finished rendering, choose → File → Save Image.**
3. **Enter a file name, choose a file type, and select or create a directory in which to save the image.**
4. **Save your scene file.**

We've included a render-ready scene file on the CD-ROM:

14_Protein/scenes/actin_render.ma

Summary

In this chapter you took a logical first step in the modeling of living systems. Proteins are a core component in what molecular biologist David Goodsell calls the *machinery of life*.[4] You now have a MEL script you can use to model virtually any protein you lay your hands on in PDB format. Moreover, you're not limited to proteins; PDB files are available for the other types of molecules manufactured by living cells: nucleic acids, polysaccharides, and lipids.

You saw that molecular representation can take several forms. CPK models help us to visualize the overall 3D form of a molecule and are useful in shape complementary studies. Scientists also routinely use ball and stick and ribbon models to study molecular structure. MolVis applications create these models from information contained within PDB files, such as the HELIX, SHEET, and TURN records. You may wish to build on your efforts in this chapter to explore the PDB format further and develop additional modeling tools using MEL.

Next we'll move from the atomic-level detail of a single protein to the next level of biological organization: multi-protein self-assembly.

References

1. Watson JD: *The Double Helix: A Personal Account of the Discovery of the Structure of DNA*, Atheneum, New York, 1968 (Touchstone, New York, 2001).
2. Bernstein + Sons: Home Page for RasMol and OpenRasMol Molecular Graphics Visualization Tool (website): http://www.openrasmol.org, accessed August 2, 2007.
3. Otterbein LR, Graceffa P, Dominguez R: The crystal structure of uncomplexed actin in the ADP state. *Science* 293: 708–711, 2001.
4. Goodsell DS: *The Machinery of Life*. Springer-Verlag, New York, 1993.

This page intentionally left blank

15 Self-assembly

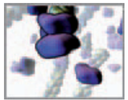


FIGURE 15.01

Computer models of dense, complex biomaterials can be explored interactively, using simulations to "fly" into the 3D meshwork. Pictured here is a joystick-operated simulation in the authors' lab. Inset: Detail from an actin assembly simulation.

Introduction

Within the living cell, proteins do not exist in point-like isolation, floating alone in the aqueous cytoplasm of the cell interior. Quite the opposite: proteins are highly gregarious molecules, though choosy about the chemical company they keep. Through the action of inter-atomic forces, which allow proteins to sense and react to one another in specific ways, proteins like to join together and assemble with non-protein molecules such as sugars and nucleic acids. This makes the cell interior a crowded place, with proteins jostled into 1D arrays (filaments and polymers, the structural "bones" of the cell), into 2D groupings (such as ion pumps and receptor channels on the cell membrane), and 3D, multi-protein machines (like the ribosome, the gigantically intricate multi-protein "jig" on which nucleic acid tapes of genetic information are decoded and the matching protein built from amino acids). Small regulatory factors like calcium and phosphorous bind to many proteins and modify their affinity for interaction, so these 1D, 2D, and 3D protein arrays essentially build themselves from their component protein subunits. This is called self-assembly and is at times assisted by assembly supervisors, chaperone proteins that help the subunits find the folding pattern best suited to interaction with other building blocks of the macromolecular array.

In this chapter you will use Maya to step into this remarkable world of macromolecular self-assembly. It will be exciting (and challenging!) enough to tackle the case of the 1D protein array. As we noted above, such polymers or filaments are enormously important in cell biology because they form the basic structural "bones" of the living cell—the cytoskeleton defining the cell's characteristic shape and capacities for movement. Diverse kinds of cytoskeleton polymers and polymer building blocks function within the cell, and their subtle activity is regulated by an intricate web of chemical factors that seed, cap, cut, reinforce, and cross-link the polymer filaments. Exploring the chemical facts and biological details of even a small corner of this vast

web would take us far beyond the scope and space constraints of this book. In this chapter you will take the first step by modeling essential ideas of filament layout and assembly organization in the presence of small regulatory factors that modify the rates, or kinetics, of the self-assembly steps. This strategy of regulated self-assembly is a universal motif of cell behavior and so makes a great departure point for your Maya explorations above the level of the single protein.

Rest assured: we are not going to leave behind what we have learned about proteins in Maya. In this chapter you will develop your model around key facts about one of the essential cytoskeleton building blocks: the actin protein molecule, which you met and worked with in the last chapter. You will discover how such a model can incorporate the core data on actin filament assembly and structure in silico. The regulation of growth and shrinkage of actin filaments is central to important cell activities like locomotion and **endocytosis**. In the detailed project steps, we guide you through the methodology and MEL code for a simulation model that provides a striking visual demonstration of actin filament dynamics.

It will be evident how our approach lets you link, step by step as needed, further reactions and regulatory pathways into the basic model. By the end of this chapter, therefore, you will have learned Maya methods and modeling strategies you can extend and modify for larger, more complex systems biochemistry applications than we can accommodate here. It will also be clear that you can apply similar techniques to Maya projects in which the self-assembly events engage 2D and 3D arrays of self-assembled macromolecules, not only the 1D structures of filamentary fame. The chapter's references will take you further afield in the rich biochemistry of actin filaments, cytoskeleton structure and control, and macromolecular self-assembly.

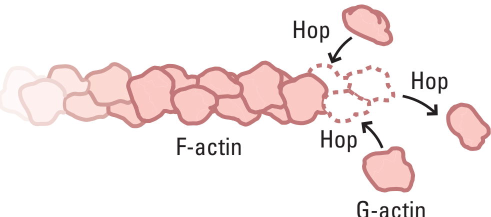
Problem overview

Traditionally mathematical models of chemical systems have often used differential calculus to compute the changing concentrations of the various reactants. This **deterministic** approach neglects the effects that random spatial diffusion of the individual reactants has on the behavior of the system. As you'll see in this and future chapters, we favor a **stochastic** approach in which both spatial diffusion and the effects of uncertainty play a role in determining the outcome of a simulation model. Moreover, this stochastic approach presents an intriguing challenge for the medical artist or scientist who wishes to at once simulate the chemistry involved and visualize its essence in a meaningful way. The challenge arises from the fact that the time intervals between significant diffusion events (see below) and the time intervals between chemical reactions are vastly different. In other words, if one wishes to observe the stepwise wandering of actin molecules as they jostle about in the cytoplasm, one would have to wait a very long time (on average) before a reaction between two molecules was likely to occur. This is not because the molecules wouldn't encounter one another but because, given an encounter, a reaction is not a certainty—there is merely a probability that the reaction will occur. Conversely, to observe chemical reactions with relative frequency, one would forgo watching—and perhaps modeling—molecular diffusion.

In this chapter we wish to model and visualize a small but important bit of biopolymer chemistry—the steady-state turnover of a filament by the balanced addition and removal of filament subunits. This process is often referred to as **treadmilling** because it involves the flux of subunits from addition at one end of the filament to removal at the other. In one treadmilling cycle, a given subunit travels from the plus to the

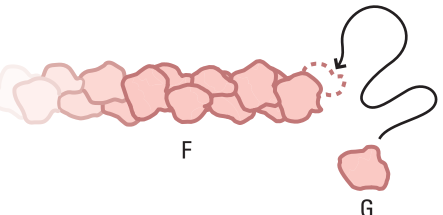
minus end of the filament. This treadmilling property of actin is central to cell locomotion and shape change. In order to capture the essence of steady-state treadmilling, your model will incorporate both time-dependent and spatial diffusion effects. However, before addressing the modeling and visualization challenge of our different time scales—for diffusion and reactions—let's look at the physics involved.

You'll recall from the last chapter that the lone actin protein, before it associates with an actin filament, is often denoted G-actin by biochemists. An actin filament, on the other hand, is referred to as F-actin. A G-actin protein is a subunit of the filament F-actin array when we think of it as coming into contact with, recognizing, and binding to the filament. Textbooks sometimes make F-actin formation look like an easy problem in procedural animation, involving the regular hopping of monomers on to and off from an F-actin filament. Think of railroad cars being added to a freight train in a nice steady order.



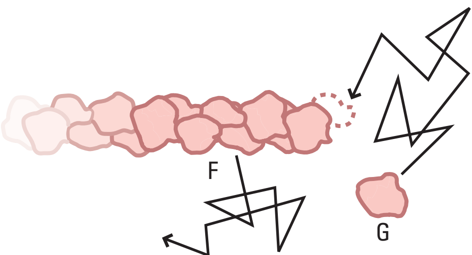
There are several problems with this interpretive visualization view, however, which make it a very limited model of filament formation and destruction:

1. First, the G-actin monomers need not at first be close to the F-actin filament; monomer and filament have to "find" one another within the cell.



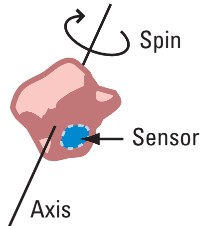
2. The second problem is that neither the F-actin nor the G-actin are equipped with long-range sensors to assist this finding operation. G-actin is not like a dolphin zeroing in on a school of tuna, using echolocation (the dolphin's sensor) to detect its target from far away through the water! The "sensor" parts of proteins are specialized regions of the protein surface, in which the amino acids are folded in very specific spatial arrangements. This lets the regions match up to other protein surfaces with complementary patterns of shape or electric charge—the regions fit together rather like a key fits a lock. Because water molecules very effectively screen the partial charges of amino acid side chains, recognition can take place only over very short range: at about the van der Walls collision distances for molecular contact we discussed in the last chapter. So the monomer has to wander about in the cell interior until it chances to collide with the docking end of an F-actin filament. (Not just any location on the F-actin will do; the bare end of the F filament has "sweet spot" where the G can hook up.)

3. The approach of F and G comprises wandering with a vengeance. In cell biology, the packed interior of the cell jostles all the cytoplasm's molecules, which cannot approach one another in the smooth manner of a space shuttle orbiting in to dock at the International Space Station. This is much more like the mosh pit of a frenzied rock concert. Every 10^{-15} seconds or so, every cytoplasmic molecule is haphazardly bumped by others around it; all the molecules are obliged to make their way through the cytoplasmic crowd by moving in a random walk manner.

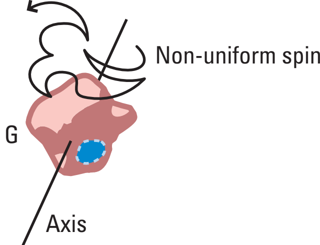


A little math, which we needn't bother to reproduce here, shows that F and G will eventually find one another this way. Indeed a remarkable trait of this 3D "diffusion" in the cell is how quickly it lets macromolecules encounter on another.

4. Getting there, however, is just a fraction of the fun. Before F and G bind together, both must rotate around in space to the extent their sensor regions can detect the "lock–key" fit, letting G bind to the growing end of F. Again we have to be careful about the mental pictures painted by our choice of words. "Rotate", at least for us, suggests a smooth spinning action, like an ice skater twirling round or the majestic turn of those donut-shaped space stations popular in Sci-fi movies of the 1950s.



As with translational movement in the cell, the diffusional jostling destroys any hope of such a nice smooth turning match-up. G and F are kicked through small bits of topsy-turvy spins by their collisions with other molecules.



5. We must think not just about F and G, but about the small regulating molecules that will interact with them and modify the probable outcome of an F–G collision. Some physics math, which we again leave to the thermodynamics texts, does tell us that, over equivalent intervals of time, the distance staggered by these lighter, smaller molecules is considerably greater than what F and G cover. You will use this observation soon.
6. Though we have cartooned them as solid, concrete-looking lumps, F, G, and the other reactants of our model exist in the cell as molecules, that is, as "societies" of chemically bonded atoms replete with all the intricate internal patterns of atomic and molecular motion. Certainly, protein chemists assure us that the recognition of protein by protein can trigger shifts—called conformational changes—in these internal patterns of molecular motion, which enhance the likelihood that the G and F will remain together for a useful period of time—that the key stays in the lock, as it were.

And we must not overlook the plain fact that atomic physics is a world of likelihoods, not certainties. Just because two sensor regions are aligned does not make their chemical binding a certainty. The atomic forces determine a probability that the well-oriented regions will lock and G bind to F. Laboratory chemists refer to this as the "rate" of the reaction given the favored alignment of the molecules.

Enough complexity! Think of all of this, and more, transpiring in every meaningful event between molecules within the cell. To set up our game plan, we will start from a fascinating and very convenient property of astrobiology: although life evolved in crowded aqueous environments on our planet, the forces of gravity, electromagnetism, and atomic packing density are not so high as to scramble together all the seven or so classes of events we just discussed. Thus, for example, the cell is not so crowded and the molecular jostle not so highly energetic that the movement of G to F strips atoms right off G (and F) and so changes their reaction likelihoods once they are together and lined up. Similarly, the intermolecular forces are so short ranged that we can, at least to start, not worry about the effects of sensor–sensor interaction on molecular rotation rate in the rotational diffusion. We can therefore impose some helpful simplifications:

1. The time scales of intramolecular rotation and vibration are very short compared to those of interest. We ignore them, and so are concerned with "average" behavior seen on the time scale of cell physiology.
2. The time a protein needs to change its shape is very fast on the time scales of interest to actin treadmilling, and may be treated as instantaneous.
3. Using steps 1 and 2 we treat the monomers as stable "lumps" which approximate atomic force surfaces. Such a coarse-grained view of the actin molecule is well suited to simulating the interaction of cell volumes in which hundreds or thousands of monomers are present, which is the direction we want you to start with this project. When atomic events inside the protein molecules are important, current supercomputers can rack up impressive results with model filaments in which a half million or more atoms (about a dozen actin subunits) are monitored at once using specialized software for molecular dynamics simulation.
4. The regulating molecules are small and low molecular weight, diffusing through large distances over the relevant time scales. We therefore treat them as a uniform chemical background against which the F–G interactions take place.

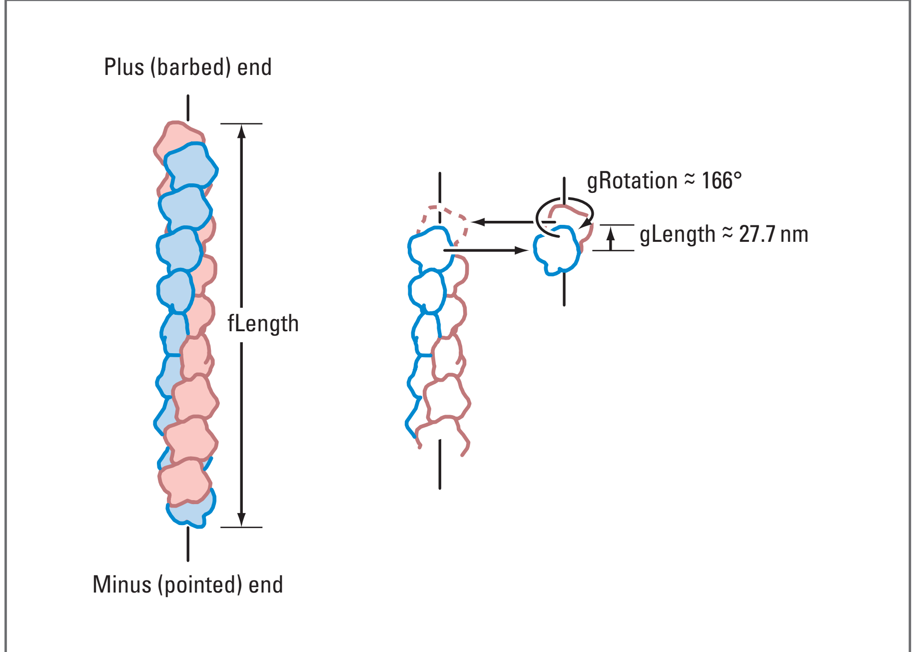


FIGURE 15.02

Moving lengthwise along the F-actin filament, each subunit is rotated by 166° and shifted 27.7 Å relative to its neighbor.

5. The macromolecules translate (diffuse) through space by a 3D random walk. The statistical properties of the walk embody the resultant impacts of those myriad small jostlings taking place too fast for any single one, on the average to impact cell events. You'll use a small set of MEL commands to conveniently generate random spatial movement that is consistent with the statistical properties of molecular diffusion.
6. Similarly for rotation.
7. When F–G sensor surfaces are aligned, a reaction probability determines the stochastic outcome of each simulated binding encounter. The mathematical problem of predicting such reaction probabilities from the first principles of quantum physics is unsolved (and likely unsolvable) in general and is certainly impractical for F–G, at least with current methods and technology. In place of fundamental theorems, models can use estimates based on chemical reaction data. You will develop the model along these latter lines.

Let's therefore look more closely at the molecules and chemistry your model will cover.

The structure of F-actin

Like the monomers themselves, the F-actin polymer (Figure 15.02) is a polar structure with a *barbed* and a *pointed* end—terms coined from an early observation of how filaments appeared in electron micrographs when bound to another protein, myosin. Under physiological conditions, filaments grow more rapidly at their barbed than

at their pointed ends, a feature that led to the terms *plus* and *minus end* which are used interchangeably with *barbed* and *pointed*, respectively.

The F-actin filament can be described both as a right-hand double helix of protofilaments and as a single helix of subunits[1] that are rotated relative to one another about the longitudinal axis of the filament. Using the latter description, as we move from the filament minus end toward the plus end, each subunit is rotated by 166° in a clockwise direction relative to the one immediately preceding it. There are 370 subunits per µm along the length of a filament, which equates to a distance of ≈ 27.7 Å from the center of one subunit to the next.

Actin reactions

Intracellular actin chemistry is a rich mixture of processes including filament nucleation, growth, shrinkage, capping, branching, and cross-linking, each consisting of one or more chemical reactions. The cell regulates these processes via accessory (or helper) molecules which influence the chemical reaction rates. Any number of these reactions can be incorporated into a model of regulated self-assembly. In this project we'll focus on the following:

1. **Plus-end association: the addition of actin monomers to a filament.**

   $G + F_i \rightarrow F_{i+1}$

2. **The hydrolysis of each F-actin subunit's bound nucleotide molecule and the subsequent release of inorganic phosphate.**

   ATP → ADP•Pi → ADP + Pi

3. **Minus-end dissociation: the removal of actin monomers from a filament.**

   $F_i \rightarrow F_{i-1} + G$

These steps are illustrated in Figure 15.03.

Actin's bound nucleotide: ATP, ADP•Pi, or ADP

An important factor in actin chemistry is the **nucleotide** that is bound deep in a cleft in the center of G-actin monomers and F-actin subunits. The nucleotide takes the form of ATP (adenosine triphosphate) or its de-energized state ADP (adenosine diphosphate). Although the mechanism of action remains unclear, the type of bound nucleotide has been linked to actin binding affinities and is therefore a regulatory factor in filament dynamics. Some time after a G-actin monomer binds to a filament, its ATP releases energy through the breaking of one of its three phosphate bonds—a process called **hydrolysis**—becoming ADP•Pi (ADP with an associated inorganic phosphate):

ATP → ADP•Pi

It is currently unclear whether or not this reaction stabilizes the subunit on the filament—that is, increases the binding affinity. Therefore reactions involving ATP-actin and ADP•Pi-actin molecules are often treated with the same reaction parameters. In other words, an ATP-actin subunit is as likely to dissociate from a filament as a subunit

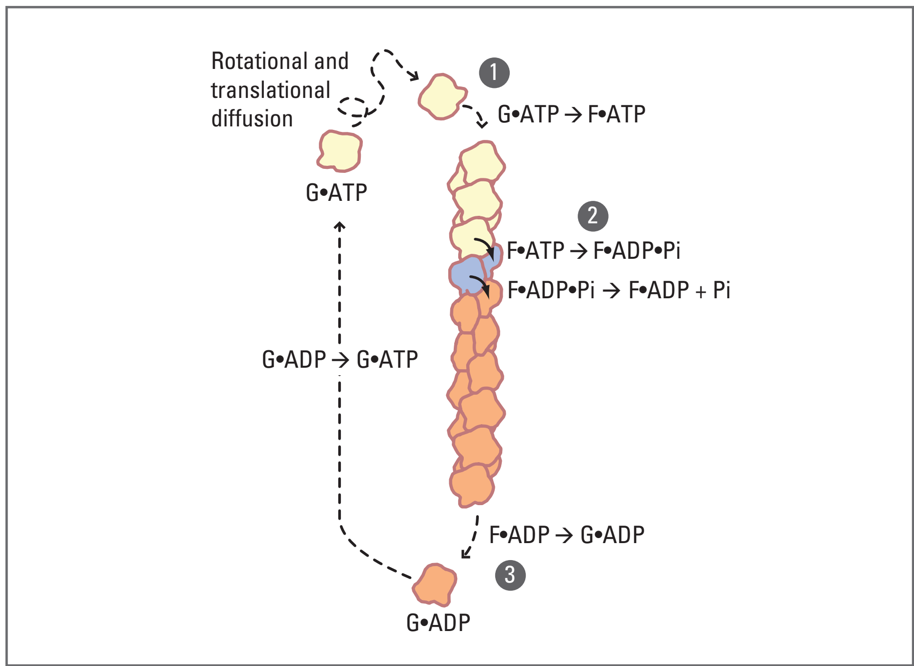


FIGURE 15.03

The diffusion and reaction events modeled in your simulation.

whose ADP has been hydrolyzed to ADP•Pi. After a period of time, however, Pi dissociates from ADP:

$$\mathbf{ADP{\bullet}Pi \rightarrow ADP + Pi}$$

This ADP-actin subunit becomes more tightly bound and is therefore much less likely to dissociate from the filament than its ATP or ADP•Pi relatives. The relative effects of the actin bound nucleotides on filament dynamics are reflected in their **reaction rate constants**—empirically derived numbers used to make predictions about chemical reactions.

Reaction rates

The chemical reaction

$$F_i \rightarrow F_{i-1} + G \qquad \text{[Reaction 15.1]}$$

in which a G-actin monomer leaves a filament can be described mathematically using a reaction rate constant (k^- in units of seconds^{-1}) to yield the number of reactions that can occur in a time interval Δt. Let N represent the number of G molecules that will leave a filament during this interval:

$$N = k^- \times \Delta t \qquad \text{[Equation 15.1]}$$

Likewise, the monomer binding reaction

$$F_i + G \rightarrow F_{i+1} \qquad \text{[Reaction 15.2]}$$

can be described using the reaction rate constant k^+ (in units of $\mu M^{-1} s^{-1}$) and the concentrations (denoted with square brackets) of either reacting species, for example:

$$N = k^+ \times \Delta t \times [G] \qquad \text{[Equation 15.2]}$$

where N is the number of monomers that bind the filament in time Δt.

Empirical studies of actin *in vitro* have yielded rate constants (k^- and k^+ values) that can serve as a guide in constructing your actin simulation in silico. One must use caution when adapting such numbers to a stochastic model in which both reaction *and* diffusion are considered. Reaction rates are thermodynamic statistical averages derived for largely homogeneous solutions of reactants. Research is showing that the widely accepted rate constants for actin don't apply without modification to the heterogeneous mixture within living cells. Furthermore, studies into the roles of actin accessory molecules such as profilin, ADF (or actin depolymerizing factor), and formin to name a few, interact in complex ways with actin to provide tight control over reaction rates in vivo. Therefore, we'll use the established rate constants as a point of departure for choosing parameter values to use in silico. Equally important in choosing these values are the scientific and didactic objectives of the model you're building. We'll come to these shortly.

Pairing reaction and diffusion: A visualization challenge

As stated earlier, if your model is to capture the essence of dynamic actin filament turnover, it will incorporate both time-dependent reactions and spatial diffusion effects.

Reaction timing

Ignoring diffusion for a moment, you could describe the association and dissociation reaction events in terms of Equations (15.1) and (15.2) above. Adapting rate constants from the actin science literature and choosing a reasonable concentration of G-ATP-actin you could quickly determine the number of reactions occurring in each time step Δt. By carefully choosing the number of Maya frames per Δt, you could set up a nice simulated visualization of actin treadmilling. For example, let's say:

1. **$k^+ = 11.6\ \mu M^{-1} s^{-1}$, the established in vitro G-ATP plus end association rate constant.**
2. **[G-ATP] = 0.1 μM, the accepted in vitro critical concentration for G-ATP-actin.**
3. **Rearranging Equation 15.2 to solve for Δt gives:**

   $\Delta t = N/([\text{G-ATP}] \times k^+)$

   The time interval required for one association reaction per filament is therefore

   $\Delta t = 1/(0.1 \times 11.6)$ s = 0.862 seconds

The critical concentration (C_c) is the minimum concentration of units needed before a polymer will form. It can be expressed mathematically as the ratio of off and on rate constants: $C_c = k^-/k^+$.

Suppose you wish to represent an association reaction in your Maya model on average once every second. A frame rate of 30 fps gives a reaction rate of 1/30 reactions per frame.

4. **We can now express Maya frames in terms of reaction time:**

   1 Maya frame = 1/30 reactions × 0.862 seconds/reaction
   ≈ 0.029 seconds

The reaction rates for hydrolysis and dissociation follow similar logic with the exception that they depend only on time and not concentration. So far so good. We have a comfortable time increment of 0.04 seconds per Maya frame—set by the reaction rate constant and a reasonable G-actin concentration—that allows us to observe about 1 association reaction per animation second (at 30 fps).

Diffusion timing

Within a cell, macromolecules such as G-actin undergo random walks due to collisions with other macromolecules, small molecules, and water. For time steps that are large compared to those between intermolecular collisions, these random walks result in diffusive motion. For their ChemCell[4] program, Steven J. Plimpton and Alex Slepoy derived the following equation to calculate the distance, r, that a macromolecule with a diffusion coefficient, D, diffuses in time, Δt:

$$r = 4(D \cdot \Delta t/\pi)^{1/2} \qquad \text{[Equation 15.3]}^4$$

ChemCell[4] is a computer program that uses particles to simulate the protein chemistry of biological cells (www.sandia.gov/~sjplimp/cell.html). It was developed by Steven Plimpton and Alex Slepoy at the Computation, Computer, Information and Mathematics Center at Sandia National Laboratories.

Plimpton and Slepoy's approach maps the effect of intermolecular jostling to times and distances relevant to events we're interested in for this model: chemical reactions between macromolecules. Calculating an approximate value for D or choosing one from the literature, we can set r to reasonable fraction of the diameter of G-actin, and then calculate the corresponding time step Δt. For example:

A value of r on the order of a few Angstroms would be appropriate when steric effects on interaction are considered. In the present model, however, nanometer diffusion step sizes work nicely with the temporal and spatial scales used.

1. **r = 1 nm (1/5 the approximate diameter of G-actin)**
2. **$D = 1.65 \times 10^{-12}\ m^2/s$**
3. **Rearranging equation 15.3 to solve for Δt gives**

$$\Delta t = \pi r^2/(16D)$$
$$= (3.14)(10^{-9})^2/(16 \times 1.65 \times 10^{-12})s$$
$$= 1.19 \times 10^{-7}\ s$$

Now, suppose we let one Maya frame equal 1.19×10^{-7} seconds—in order to visualize and animate the effects of reasonable diffusion step sizes—and use our k^+ and concentration values from above. In this case you'll be waiting some 0.862 s/reaction / $(1.19 \times 10^{-7}$s/frame) ≈ 7.25 million frames (or ≈ 3 days at 30 fps!) between association reactions. This is clearly not a good outcome if the purpose of your model is to both simulate *and* visualize actin steady-state treadmilling. After your monomer associated with its filament, you'd then have to wait another 3 days times the number of subunits in your filament for one treadmilling cycle to finish!

Therefore, in order to represent actin treadmilling behavior within reasonable observer time scales you'll take a novel, yet straightforward approach to simulation and interpretive visualization.

Model conditions

Before leaving the problem overview behind, let's state some initial conditions that will help you get your Maya model up and running as quickly as possible. Rest

assured, the modular approach we're taking will allow you to increase the model's complexity if you wish to explore actin dynamics further.

1. Association reactions occur at the filament plus end only. This models the discovery that, in vivo, G-actin is associated with helper proteins, such as profilin, that inhibit it interaction with filament minus ends. In contrast, profilin bound G-actin has a high affinity for the plus end of filaments. While you won't model profilin explicitly, its effects are implicit in this condition for the model.
2. The environment created by additional helper proteins, such as formin, at the plus end, inhibits the off rates there in comparison with the on rates and with the helper protein stimulated off rates at the minus end (ADF/cofilin).
3. The model features one filament and a small number of G-actin monomers. Since you're considering G-actin binding at the filament plus end only, the monomers are shown (for clarity) to move within a bounding region (the *concentration volume*) centered at the plus end. The monomer number and their concentration volume determine the G-actin concentration.
4. It's difficult to make out treadmilling activity when filaments are themselves buffeted about by diffusion. Therefore, in this model you'll limit diffusive motion G-actin monomers, keeping the filament aligned with a major axis and stationary. In this way, the filament's local axes serve as the frame of reference for observing subunit flux.
5. The only G-actin is G-ATP-actin. Physiologic concentrations of ATP are much higher than ADP and any free G-ADP-actin is quickly changed to ATP-G-actin.
6. Once G-ADP dissociates from the filament minus end, its ADP is thus exchanged for ATP almost immediately and it rejoins the pool of G-ATP-actin. This allows you to maintain a constant supply of G-ATP-actin ready for association at the filament plus end. As well, due to the action of sequestering proteins and the fact that G-ADP has a very low association affinity for F-actin, there is no practical reason you should be consider free G-ADP-actin in your model at this time.

Conditions like the ones above are common to mathematical modeling of complex systems. They set reasonable boundaries within which you can get things working. Once you're satisfied the model is working well under the current conditions you can explore different sets of assumptions, relaxing as many of these conditions as you wish to model actin filament dynamics in the test tube or in the cell.

Methods: Actin geometry

There are several possible approaches to simulating molecules diffusing in a Maya scene, including particle systems and rigid body dynamics, both of which we've explored in our research. For this project we favor an intuitive approach that uses polygon models to represent the atomic contact surfaces of actin molecules, combined with diffusion and collision engines you'll code in MEL. While this approach foregoes some of the efficiencies of Maya's particle systems and the collision detection capabilities built into Maya's rigid body dynamics, it also bypasses their shortcomings and ultimately lessens the amount of MEL code required to implement this model.

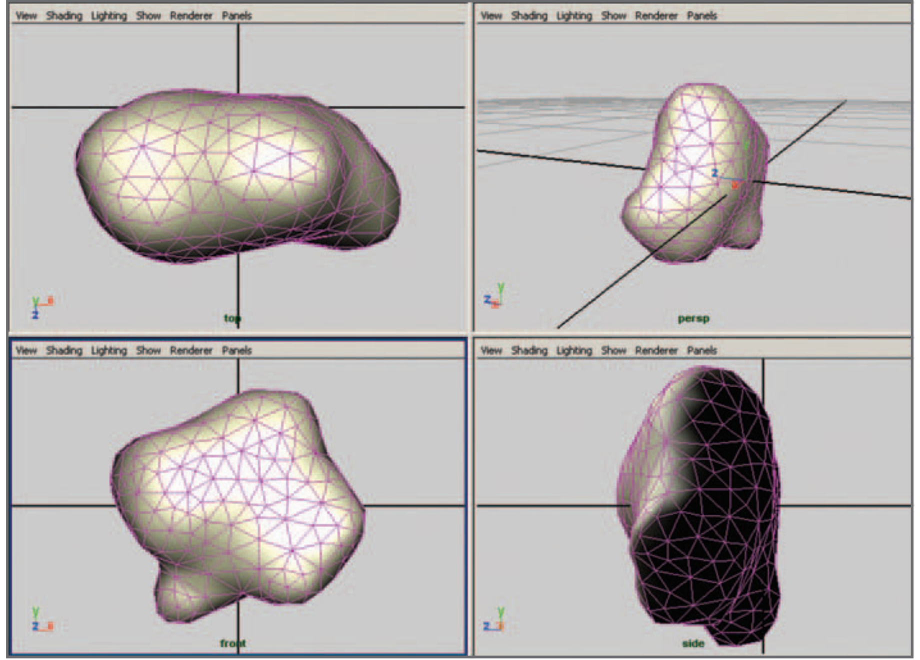


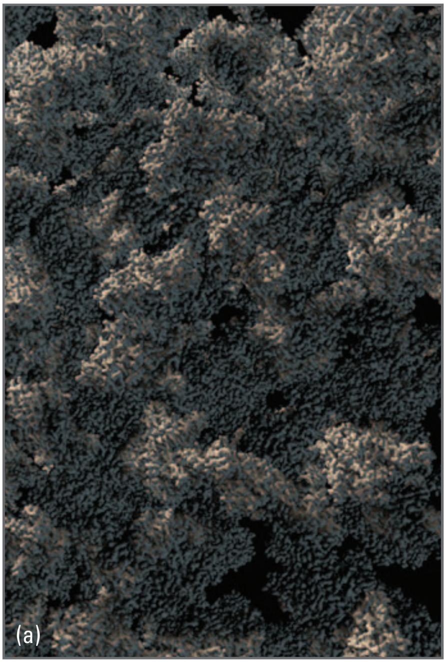
FIGURE 15.04

The G-actin template is a low level-of-detail (LOD) model comprised of 520 polygons. It was created with UCSF Chimera, using Protein Data Bank file 1j6z.pdb.[2]

The G-actin template model

Individual's G-actin monomers and an F-actin subunits were created by instancing a template model (Figure 15.04) created from molecular structure data—the same data you applied in the previous chapter to build the CPK model of actin. However, unlike a space-filling CPK model, in which every atom is instanced with a sphere, your camera distance in this scene is great enough that you won't need the exact location of every atom—recall it's the complementarity of surface shapes that's key to protein–protein recognition. Your template model will be a low level-of-detail (LOD) polygonal surface expressing this surface shape idea (the concept of level-of-detail was discussed in the previous chapter, beginning on page 344). For the CPK model, we were interested in representing the location and species of the individual atoms (nearly 3000 in total) that comprised the actin molecule.

A low-LOD surface model of a molecule is a geometric object that captures the general features of a molecule's surface—it's lobes and clefts—with a minimal number of polygons. A high-LOD model, in contrast, captures surface characteristics in more detail using a greater number of polygons. When you choose a low-LOD over a high-LOD model you gain processing speed in favor of structural detail. A lower degree of surface detail can also benefit your viewers. There is a point, which you must judge for each of your projects, beyond which additional surface detail no longer aids understanding, and can even interfere with a grasp of the subject being illustrated. In a particularly complex visual scene, with many moving and interacting objects—such as your G-/F-actin scene—low-LOD surfaces can make it easier for the viewer to tell objects apart. This principle applies equally to still and animated images and is illustrated in Figure 15.05, which compares two versions of the same Maya scene at different LOD.



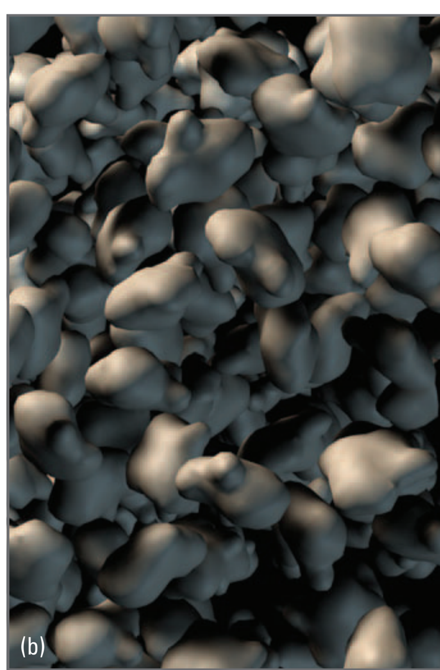


FIGURE 15.05

Too much surface detail can make an image difficult to read. Both (a) and (b) show the same 250 G-actin monomers, however (a) depicts the van der Waals surfaces at atomic-radius resolution, whereas (b) shows the low poly-count surfaces that average over this fine, atom-scale detail—using the template surface model we provided on the CD-ROM. In (a) it is difficult to tell one actin molecule from another, whereas the individual objects can be clearly distinguished in (b).

At the time this book was published, the Maya developers had yet to incorporate tools (like those embedded in UCSF Chimera) for making a polygonal surface model from a cloud of points like the atomic coordinates in a PDB file.

To help you focus on the MEL development, we have prebuilt the actin template model and included it in the book's companion CD-ROM. The model was created using the Multiscale Models tool in UCSF Chimera software, which can be downloaded from `http://www.cgl.ucsf.edu/chimera/` and used free of charge for a variety of structural modeling purposes (UCSF Chimera was introduced in the previous chapter). The model is based on a Protein Data Bank file for the actin monomer, `1j6z.pdb`.[2] The original PDB file can be found by searching `1j6z` on the Protein Data Bank at www.pdb.org. You'll find our low-LOD actin model saved in a Maya scene file on the CD-ROM:

15_Self_Assembly/scenes/actinTemplate.ma

We created the template model with enough surface detail to capture the general morphology of the G-actin (or F-actin subunit) molecule—its characteristic shape (Figure 15.04). This lower-LOD surface model also satisfies our desire for a low polygon count (520 polygons) relative to a van der Waals surface model of actin at atomic resolution (≈33,000 polygons). You could easily go lower than 500 polygons, but your actin model would begin to look blocky.

The template model was correctly oriented relative to the filament axes—which is aligned with the Y-axis in Maya—and offset its pivot point so that it can be duplicated and positioned properly on a growing fiber simply by rotating it 166° and moving it lengthwise by 28 Å relative to the preceding subunit. The degree of offset and the rotational orientation of the model were determined by comparison with the Holmes/Lorenz model of F-actin.[3]

Item	What the item looks like in Maya	Description
F_0		F-actin filament (NURBS cylinder)
G_#*		G-actin monomer (instance of template model)
GF_group†	NA	Group of all F-actin subunits within a filament
Plus		Plus end locator
Minus		Minus end locator
pmGroup	NA	Group of plus and minus locators within the filament
plusConcVol		Plus end concentration volume
plus RxnVol		Plus end reaction volume

*The # symbol represents an actual number such as 1, 2, 3, etc.
†F-actin subunits are grouped (parented) under a null transform node GF-group prior to parenting under the F-actin model.
This makes it easy to shift all GF models at once to make room for a new binding subunit in the dynamic simulation.

TABLE 15.01

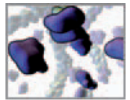
Nomenclature used in the actin simulation model.

One surface model for all species of actin?

The different types (or species) of actin we're considering, G•ATP, F•ATP, F•ADP•Pi, and F•ADP are almost identical to one another in overall shape but—and this is crucial to the biological significance of your model—have different chemical properties. The conformational (shape) changes in the monomers and subunits associated with binding and nucleotide events, while small relative to the overall size and shape of actin, help trigger changes in those chemical properties. For instance, the release of the inorganic phosphate molecule (Pi) from ADP•Pi in an F-actin subunit induces a shape change that increases its binding affinity for its neighbors. However, the effects of these conformational changes are encompassed nicely in the reaction probabilities. As a result, such subtle structural changes need not be considered in this model. Instead the differences between actin states can be more effectively represented through naming, shaders, and custom attribute values.

The F-actin model

Table 15.01 lists the Maya components that make up the F-actin model. They are shown in context in Figure 15.06. The collision surface of the filament is a NURBS cylinder with



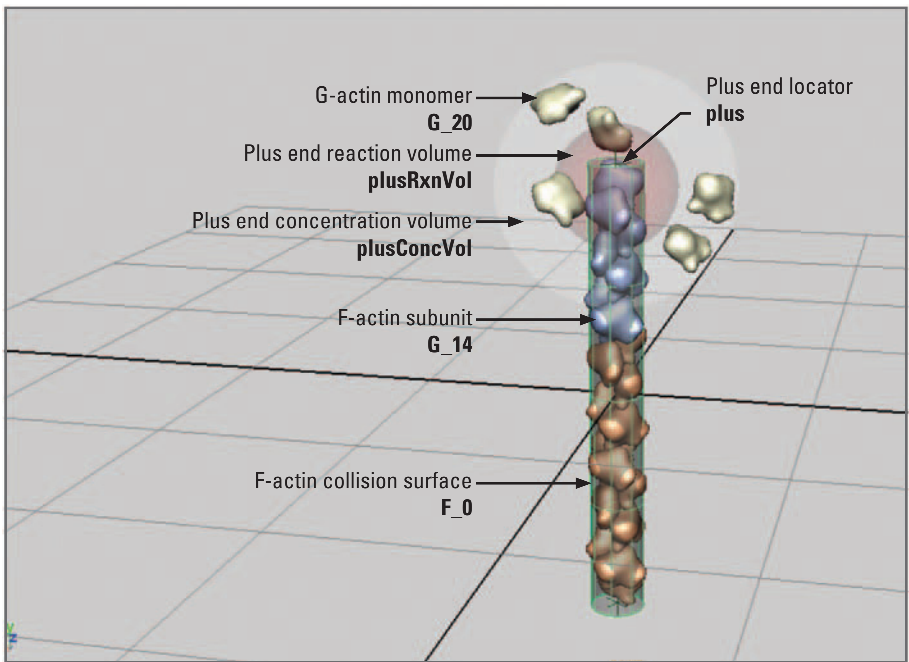


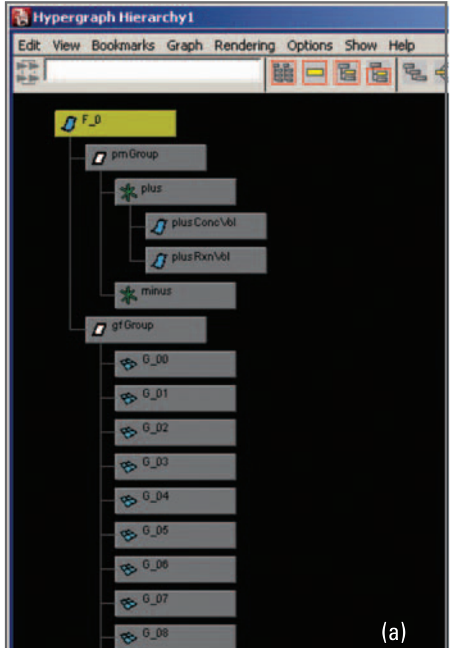
FIGURE 15.06

The initial setup for the actin treadmilling model contains a ready-made filament and five G-actin monomers. The translucent cylinder surrounding the filament subunits is used both as the filament transform node and its collision surface.

its `Visibility` attribute set to off or 0. Its 7 nm diameter defines the approximate cross-sectional area of an actin filament. Think of the cylinder as an invisible shell surrounding the constituent subunits. It grows in length as monomers are added to it. What you'll *see* of the filament in the scene are its subunits, which are visible duplicates of the G-actin template model. Every time the filament reacts with a monomer it lengthens and the monomer changes status to a *subunit* and becomes a child of the filament. The subunit then no longer diffuses—its movement instead dictated by the motion of its parent filament which (in the present model) is stationary relative to the viewer.

Plus and minus ends of the filament are represented by locators—non-rendering Maya objects that show up as cross-hairs in your scene view. When each locator is positioned the distance of one-half subunit ($1/2 \times 27 = 13.5$ Å) beyond its filament end and rotated another 166°, it conveniently marks the location where the next reacting G-actin will be placed when it becomes a subunit. A minus end locator is included in case you wish to extend your model to incorporate minus end association reactions. We'll address the concentration and reaction volumes shortly.

Maya's Scene Hierarchy lends itself conveniently to modeling the nested relationships of biomolecular structure. Figure 15.07 shows how the molecular relationships within an F-actin filament can be modeled using transform nodes in Maya. These relationships differ from those we've discussed in previous chapters, where they involved attribute connections between nodes; the connections were the domain of the Dependency Graph. Instead, parent–child relationships belong to Maya's Scene Hierarchy which can be viewed in the Outliner (Figure 15.07a) and in the Hypergraph (Figure 15.07b).



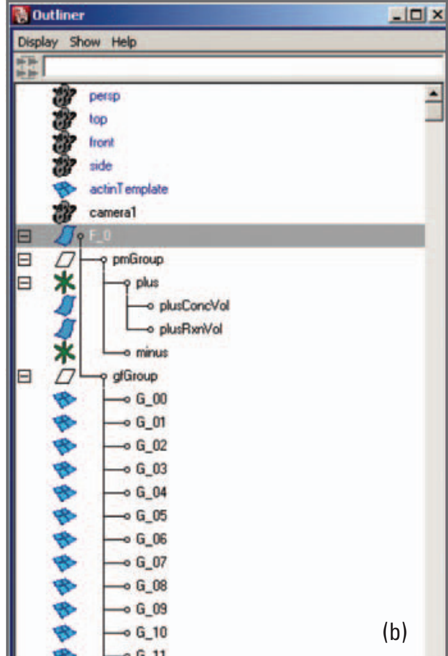


FIGURE 15.07

An F-actin model in the Maya's Scene Hierarchy. It is composed of five F-actin subunits.

(a) Viewed in the Hypergraph.

(b) Viewed in the Outliner.

To save your time—and keep the printed code listings to a minimum here—we've included a ready-made actin scene file on the CD-ROM (called treadmilling.ma), with which to begin building your simulation model. In living cells, actin never exists as either entirely monomeric or entirely filamentous—a balance exists between the two. The simulation file on the CD-ROM is therefore set up with a filament model composed of 20 subunits and five G-actin monomers. The monomer and subunit models are instances of the G-actin template model discussed in the previous section. Spheres have been parented to the filament plus end as a visual cue for the G-actin concentration and reaction volumes (more on this second volume shortly):

15_Self_Assembly/scenes/treadmilling.ma

Now you're set for a strategy to let filaments grow and also change their nucleotide profiles according to the events outlined in Figure 15.03.

Methods: Diffusion and reaction events

This section is divided into the different diffusion and reaction events that together create treadmilling behavior.

Diffusion

In mathematical terms 3D Brownian diffusion can be conveniently described as the product of three Gaussian random number distributions.[4] As a result, the stepwise

x, y, and z displacements of your G-actin monomer can be obtained by sampling a Gaussian number generator with standard deviation

$$\sigma = (2D_T\,\Delta t)^{1/2} \quad \text{(Ref. 4)}$$

where D_T is the translational diffusion constant and Δt, the time increment, both of which were introduced earlier in this chapter. Maya's gauss command takes standard deviation as an argument and returns a random number from Gaussian distribution, with mean value = 0. You will therefore generate your translational diffusion vector in MEL as follows:

```
float $tD, $t, $stdDevTrans, $x, $y, $z;
vector $trans;
$tD = 1.65 x pow(10, -12); // m^2/s;
$t = 1.19 x pow(10, -7); // seconds.
$stdDevTrans = sqrt(2.0 * $tD * $t) * pow(10, 9);
$x =`gauss $stdDevTrans`;
$y =`gauss $stdDevTrans`;
$z =`gauss $stdDevTrans`;
$trans = << $x, $y, $z >>;
```

The standard deviation, `$std Dev-Trans`, was multiplied by 10^9 to scale from meters to nanometers—our Maya model's working units. Rotational diffusion is handled in a similar fashion to translation, with a rotational diffusion constant $D_R \approx 2 \times 10^5$ rad^2/s. Both translational and rotational diffusion will be calculated in a MEL procedure called `diffuse()`.

Collisions

Collisions between G-actin models will be detected by testing their proximity of their pivot points against a threshold value (5.4 nm ≈ G-actin diameter). If molecule A is within the threshold distance of molecule B, A will take a step away from B in the direction of the vector that separates A and B from one another. Likewise, B will step away from A in the opposite direction. The magnitude of this step can be set to any value you wish. For starters, you'll use 1.4 nm (≈ 1/4 G-actin diameter).

Collisions between a monomer and the F-actin filament will be detected by testing the distance from the monomer's pivot point to the closest point on the F-actin NURBS cylinder. You'll use a special node called `closestPointOnSurface` (cpos in our book for short) to perform this test. cpos is connected to the cylinder through Maya's Hypergraph. When assigned an input vector value (G-actin's pivot location), cpos returns (via its `position` attribute) a vector that is the closest surface point to the input vector. If our monomer is within a threshold distance from the filament, the G-actin model will take a step (again 1.4 nm in magnitude) away from the filament.

Given the viscosity of the cytoplasm and the diffusion time steps being considered, you can treat the motion of your actin monomers as highly damped. You can therefore neglect the effects of momentum imparted by collisions. Once two molecules have collided and moved apart they are free to continue diffusing without any physical memory of the recent collision. This is certainly a simplified collision algorithm, but is sufficient for the current model.

All collisions will be evaluated and converted to avoidance vectors ("steps away") in a procedure called `collide()`.

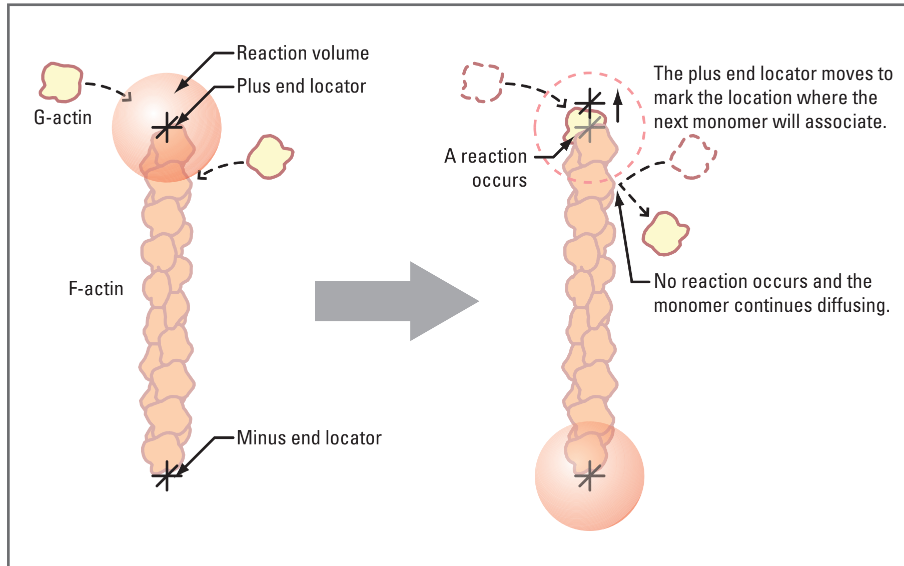


FIGURE 15.08

Collisions between F- and G-actin result in association reaction if two conditions are met:

(1) The collision occurs proximal to the filament plus-end.

(2) The probability of the reaction is greater than a randomly drawn number.

Reaction 1: Association

Figure 15.08 shows two G-actin monomers colliding with F-actin in the time interval of one frame. For a chemical reaction to happen—the binding of G-actin to F-actin—the collision must occur at the plus end of the filament (at least in the present model). To qualify as being "at the plus end" the monomer must breach a spherical reaction volume. We will discuss the reasoning behind this approach shortly in the section entitled *Association reaction rate*. A monomer that enters this reaction volume reacts with the filament and is added to the filament's current group of subunits. A monomer that collides with the filament outside of the reaction volume—that is, not at the filament plus end—deflects off the filament and continues diffusing.

In order to add an associating monomer to the filament, your simulation model must be able to query the location of the plus end relative to Maya's world space. This functionality is provided by the plus end locator. For an association reaction, your MEL code will update the G- and F-actin Maya models in a series of steps to reflect their new status:

1. **Translate the G-actin model to the position of the filament's plus-end locator.**
2. **Rotate the G-actin into its correct orientation relative to its neighboring F-actin subunit.**
3. **Parent G-actin to F-actin.**
4. **Increase the length of F-actin (the NURBS collision surface) by one subunit.**
5. **Reposition the plus or minus-end locator to account for the increased filament length.**
6. **Update custom attributes that store the filament's subunits count and length.**

Association reactions will be evaluated in a procedure called `associate()`.

TABLE 15.02

Actin bound nucleotides are indicated in the model by a custom state attribute and a unique shader.

Actin species	State attribute value	Shader name	Normalized RGB color values (R, G, B)	Color
G•ATP	0	ATP_shader	1, 1, 0.8	
F•ATP	1			
F•ADP•Pi	2	ADPpi_shader	0.65, 0.75, 1	
F•ADP	3	ADP_shader	1, 0.7, 0.5	

Reactions 2 and 3: **Hydrolysis and phosphate release**

The nucleotide profile for F-actin—the distribution of F•ATP, F•ADP•Pi, and F•ADP-actin subunits along a filament—has been widely studied for its implications in the regulation of F-actin length and subunit turnover (treadmilling). By building into the F-actin subunit and G-actin monomer models the ability for each to change the state of its nucleotide, you will have the flexibility to simulate and visualize different hypotheses about the role of the nucleotide profile in filament regulation, should you wish to explore the simulation model further. To set up the model, you need only assign an initial nucleotide state to each F-actin subunit and G-actin monomer. To emulate actin filament conditions encountered in vivo, your Maya filament starts out with a plus end "cap" of ADP•Pi subunits, with the remaining subunits in the ADP state. Each G-actin monomer is assigned an ATP state. These nucleotide states are tracked using a custom `state` attribute and a unique shader (Table 15.02). To assess the nucleotide profile of a given F-actin filament, you need only query the state attributes of its constituent subunit molecules.

Hydrolysis and phosphate release are not directly dependent on molecular concentrations the way G-actin association reactions are. Instead these nucleotide reactions depend on time and rate constants. Based on the rate constant, a probability is calculated and tested against a random number at every time step Δt which you'll treat as one Maya animation frame. The rate constants for each reaction will be discussed below.

Reaction 4: **Dissociation**

In vivo, there are different mechanisms by which actin filaments disassemble and a complete picture of these mechanisms has not yet emerged. Your model treats one of these: the dissociation of G•ADP-actin from the minus end of the filament. When the monomer/subunit dissociates, the following steps are taken:

1. **Move the newly G-actin monomer (former F-actin subunit) away from the filament minus-end.**
2. **Decrease the length of the F-actin NURBS collision object by one subunit.**
3. **Reposition the plus and minus-end locators to account for the decreased filament length.**
4. **Create and assign a temporary shader to the G-actin model.**

The shader assigned in step 4 will allow you to fade out the G-actin model over a specified number of frames. When it's transparent, your MEL code will move the monomer into the filament's plus end concentration volume and assign it the ATP_shader. At this point the monomer has joined the pool of ATP-G-actin and is ready to bind the filament should the opportunity arise.

You'll build a procedure called `dissociate()` to evaluate and execute all dissociation reactions.

Methods: Reaction rates and probabilities

In this section we'll be working quickly through the chemistry math needed to set up the animation events in your MEL code. If you would prefer to move directly to MEL coding, you can take those parameters as established, go to the next section, and return to this section at a later reading.

We'll now address the dual scientific and interpretive visualization requirements of your model:

1. Simulate both the reaction and diffusion events that give rise to self-assembly and treadmilling behavior.
2. Do the above in a way that permits the important events to be appreciated during the animation. The disparate time scales for relevant diffusion and reaction events enhance, as we've seen, the challenge of designing an effective visualization.

You'll follow a workflow in which you establish the visualization requirements of your model and let the reaction rate constants and corresponding probabilities follow naturally from those requirements. Let's derive the numbers to simulate treadmilling behavior in a timely and visually striking fashion.

Visualization requirements of the model

You want to record a 30 second clip of your model in action. Half a minute is long enough to present detail but short enough not to bore viewers.

A frame rate of 30 fps gives you 900 frames to work with. Next, let's say you are asked to show an average of five complete treadmilling cycles per simulation run—being a stochastic model, this number will of course vary from run to run. Five cycles give your audience several opportunities to see the cycle and its components in action.

Bear in mind that for the purpose of seeing treadmilling up close you're using a much shorter filament than one would typically encounter within a cell.

One cycle would therefore last approximately

900 frames/run ÷ 5 cycles per run = 180 frames

Your actin filament in this project is 20 subunits long. One cycle represents the journey of any given subunit from the plus to the minus end of the filament. You therefore have a filament flux rate

subunit flux rate = 20 subunits per 180 frames = 1/9 subunits per frame

At this flux rate, you can expect a subunit to travel the length of the filament in roughly 180 frames, or six seconds of animation playback time.

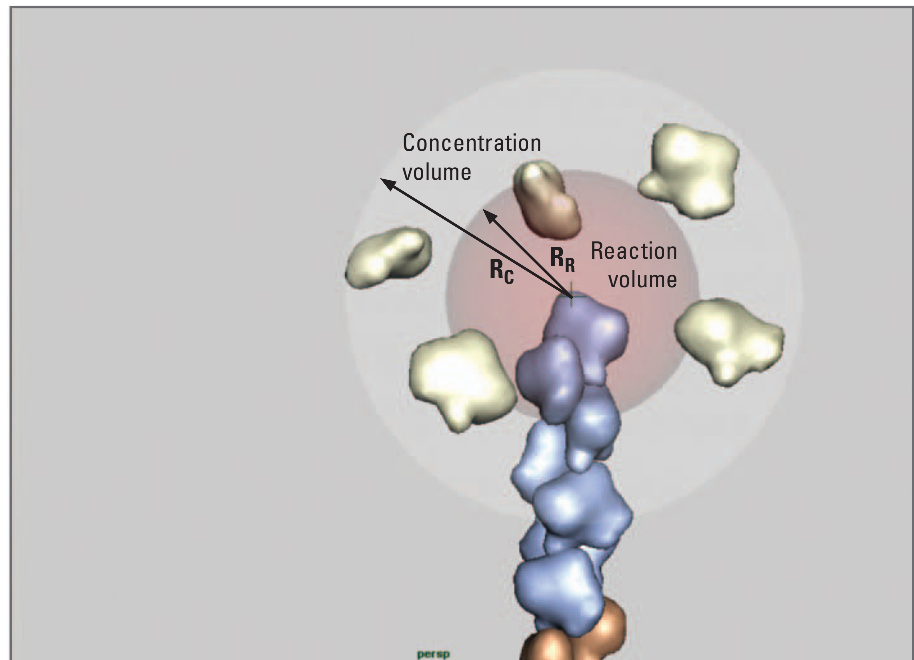


FIGURE 15.09

A concentration volume (light gray sphere) defined by radius R_C lets you calculate the G-actin concentration and contain the monomers to the useful space around the filament plus-end. The red sphere defines the association reaction volume of radius R_R.

Smoothing the flow from G-ADP to G-ATP-actin

You'll recall from Model Condition #6 on page 394 that newly dissociated G-ADP-actin monomers are to rejoin the pool of ATP-G-actin at the filament plus end, ready for reassociation with the filament. Rather than moving these transitioning molecules instantly—which makes the molecules appear as if they're popping out of and into existence—let's take an approach that is less jarring visually, and therefore less distracting from the main action: treadmilling and diffusion. With this approach, you'll fade out each recently dissociated G-actin monomer over several frames, move it to the filament plus end, and then fade it back in to join the pool of association-ready molecules.

Association reaction rate

In steady-state treadmilling, the above subunit flux rate is the same as both your association and dissociation reaction rates. Since only whole monomers (not fractions) can bind the filament, there must be a probability,

$$P^{+} = 1/9 = 0.11$$

that one monomer will associate during each Maya frame to provide the animation pace you need. In your dynamic model, this probability is a function of diffusion as well as chemical reaction. An innovative way to handle this probability was developed by Steven J. Plimpton and Alex Slepoy for their ChemCell program.[4] Implemented in our Maya simulation, their approach involves a *reaction volume* centered at the filament plus end (Figures 15.09 and 15.10). Any monomer entering this volume has a probability of

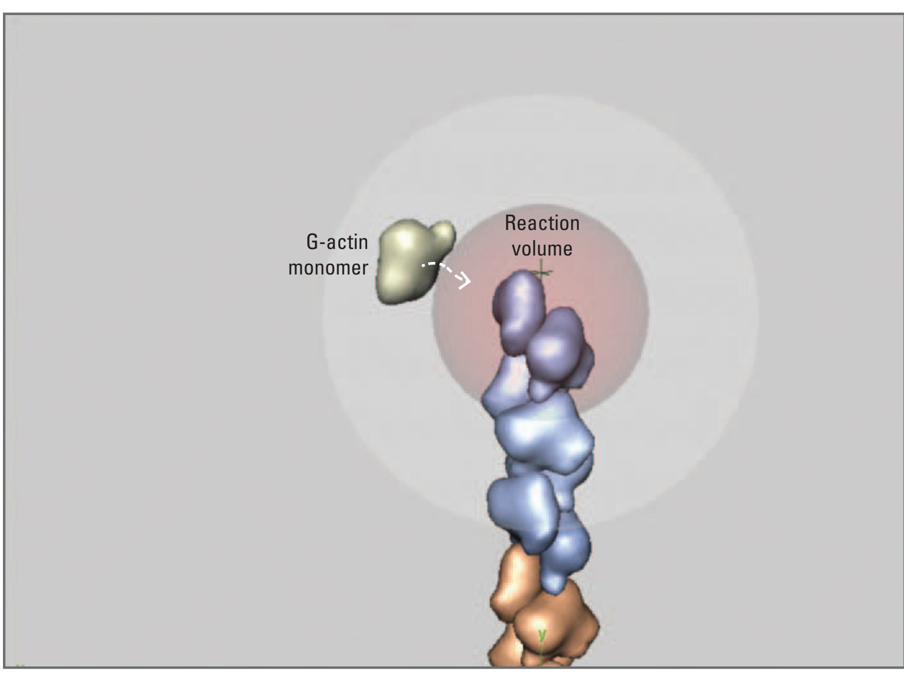


FIGURE 15.10

The red sphere represents the plus-end reaction volume V_R (a sphere of radius R_R). A reaction occurs with probability P when the surface of a G-actin monomer penetrates this volume.

binding to F-actin that is naturally dependent on the size of the reaction volume. To derive this probability, let's start by setting the G-actin concentration and calculating the appropriate rate constant. The subsequent use of subscript and superscript letters is outlined in Table 15.03. Let

[G]	be the micromolar concentration of G-ATP monomer (μM).
$Rate^+$	be the association rate in monomers per second.
k^+	be the association rate constant in monomers per micromolar second ($\mu M^{-1}\ s^{-1}$).
V_C	be the spherical volume in liters of radius R_C used to determine [G] (Figure 15.10).
N	be the number of G-actin monomers in solution.
A_V	be Avogadro's number $= 6.022 \times 10^{23}$.

You first set R_C and N to any values you like. For starters, we chose:

$$R_C = 15\,\text{nm}\ (15 \times 10^{-9}\ \text{meters}),\ N = 5\ \text{monomers}.$$

This gives

$$V_C = 4/3\pi R_C^3 = 1.4 \times 10^{-20}\ \text{liters}.$$

Subscript/ superscript	Meaning	Example
+	of the filament plus end	k^+
C	of the concentration volume	V_C
V	Avogadro's number	A_V
F	of the F-actin filament	R_F
G	of G-actin monomers	R_G
R	of the plus end reaction volume	R_R
'	the distance between object centers	R_R'
D	of the ATP → ADP•Pi hydrolysis reaction	k_D
PI	of the ADP•Pi→ ADP + Pi reaction	k_{PI}

TABLE 15.03
Subscript and superscript notation used in the derivation of reaction rates and probabilities.

$$[G] = N/(V_C A_V) \times 10^6 = 590\,\mu M. \text{ (the free G-actin concentration)}$$

Next we'll calculate the reaction rate and derive k^+:

$$Rate^+ = (1/9 \text{ monomers/frame})/1.19 \times 10^{-7} \text{ s/frame} = 9.26 \times 10^5 \text{ monomers/s}.$$

$$k^+ = R^+/[G] = (9.34 \times 10^5 \text{ monomers})/\text{s}/590\,\mu M = 1.59 \times 10^3\,\mu M^{-1} s^{-1}.$$

Typical reaction rates for in vitro actin preparations are on the order of 10^4 association reactions per second (for $k^+ \approx 12\,\mu M^{-1}\,s^{-1}$ and [G-ATP] $\approx 0.1\,\mu M$).[5] While in vivo G-ATP concentrations have been measured in the neighborhood of $0.1\,\mu M$, in vivo rate constants are a source of ongoing investigation and debate in the cell biology community.

This k^+ value should (in theory!) yield the desired number of association reactions per Maya frame—an average of one every nine frames. What is needed now is a way to relate this rate constant to the bustling diffusion about the filament plus end. Plimpton and Slepoy provide an elegant approach in the following equation. It relates the binding probability of a two-reactant chemical reaction to the reaction volume V_R and incorporates the effect of translational diffusion and the reaction rate constant k^+. The radius R_R of this volume is the maximum separation between the monomer and filament surfaces for which a reaction is considered probable (Figure 15.10). The probability P^+ of this reaction is expressed as

$$P^+ = k\Delta t/(A_V V_R)$$

where $V_R = 4/3\pi\, R_R^3$ (V_R units are liters, R_R units are meters)

Therefore $P^+ = k\Delta t/(A_V 4/3\pi R_R^3)$

Rearranging this equation, setting $P^+ = 1$, and solving for R_R yields the desired reaction radius

$$R_R \approx 4.23\,\text{nm}$$

In other words, if the surface of a G-actin monomer comes within a distance R_R of the filament plus end surface, there is a probability of 1 (a certainty!) that the ensuing chemical events at the top will bind the monomer to the filament. Let

$R_G \approx 2.7\,\text{nm}$ be the approximate radius of a G-actin monomer, and

$R_F \approx 3.5\,\text{nm}$ be the approximate radius of an F-actin filament.

For the G-actin radius we averaged the largest (~6.8 nm) and smallest (~4 nm) dimensions, and divided by 2 to get 2.7 nm.

Then the distance R_R' between the G-actin center and F-actin plus end center is given by

$$R_R' = R_R + R_G + R_F = 4.2 + 2.7 + 3.5 = 10.4\,\text{nm}$$

This value can now serve as the key parameter in your model to evaluate association reactions resulting from diffusive motion. R_R' will be represented by the variable `$RrPrime` in your model. To visualize the reaction volume, the model uses a sphere of radius:

$$R_R + R_F = 7.7\,\text{nm}$$

When a monomer's surface breaches the surface of this volume, it is considered for an association reaction. Let's now look at the time-dependent reactions and derive their rate constants.

Hydrolysis and phosphate release rates

Within the cytoplasm, hydrolysis of ATP-F-actin and the subsequent release of the cleaved inorganic phosphate molecule are closely linked to the activity of profilin and formin at filament plus ends. You'll recall that profilin acts as a chaperone, catalyzing association reactions at the filament plus end while inhibiting them at the minus end. Recent empirical evidence suggests that a filament whose two terminal plus end subunits are profilin-ATP-actin can't grow until ATP has been hydrolyzed and Pi released from the penultimate subunit.[6] This evidence suggests that rapid hydrolysis and Pi release may be essential for rapid filament growth in the cell where actin dynamics is supervised by a myriad of helper proteins. It also provides you with another degree of control over plus end association reactions in your animation. Each association reaction will be contingent upon phosphate release from the penultimate plus end subunit.

The state of a bound nucleotide also affects the likelihood of a subunit dissociating from the filament. Widely cited minus end dissociation rate constants for in vitro actin solutions

$$k^-(\text{F} \rightarrow \text{F} + \text{G-ATP} \bullet \text{Pi}) \approx 0.8\,\text{s}^{-1}$$
$$k^-(\text{F} \rightarrow \text{F} + \text{G-ADP}) \approx 0.3\,\text{s}^{-1}$$

suggest that phosphate release slows depolymerization, thereby stabilizing filaments. Given the above numbers, if a subunit were to make it to the minus end without releasing its phosphate, it would be almost three times more likely to dissociate than if it had lost its phosphate molecule. This property can lead to catastrophic depolymerization in a short filament like the one in your model if you set the phosphate release rate low enough that ADP•Pi subunits are somewhat likely to reach the minus end.

However, since in this project you are interested in emulating actin dynamics as they exist in vivo rather than in vitro—and since there is evidence implying that phosphate

release is necessary for plus end growth in the presence of plus end helper molecules—you will set your nucleotide reaction rates in accordance with your association rate. Therefore, given your associate rate

$$\text{Rate}^{+} = (1/9\,\text{monomers/frame})/1.19\times10^{-7}\,\text{s/frame} = 9.34\times10^{5}\ \text{monomers/s.}$$

and the requirement that the terminal subunit must hydrolyze its ATP and release its phosphate before it becomes the third subunit from the plus end (i.e. in two steps), let's calculate the hydrolysis and phosphate release reaction rate constants (k_D and k_{PI}). Combined both reactions must occur within 9 frames. In other words, each reaction must occur in 4.5 frames. Let Rate_D and Rate_{PI} be the reaction rates in monomers per frame, then

$$\text{Rate}_D = \text{Rate}_{PI} = (1/4.5\,\text{subunits/frame})$$
$$\text{Rate}_D = k_D\Delta t,\ \text{Rate}_{PI} = k_{PI}\Delta t.$$

Therefore

$$(k_D + k_{PI})\,\Delta t = 2(1/4.5) \rightarrow k_D + k_{PI} = (4/9)/\Delta t$$

The typical in vitro ratio of hydrolysis and phosphate release rate is

$$k_D/k_{PI} \approx 0.3/0.0026 \approx 115$$

Since you have nine, and not 115, frames in which to perform both reactions, you'll have to alter this in vitro ratio. For starters, let's go with

$$k_D/k_{PI} = 2/1 \rightarrow k_D = 2k_{PI}$$

Therefore

$$k_D + k_{PI} = 3k_{PI} = 4\,/(9\Delta t) \rightarrow k_{PI} = (4/27)/\Delta t = 1.25\times10^{6}\ \text{s}^{-1}$$

$$k_D = 2k_{PI} = 2.50\times10^{6}\ \text{s}^{-1}$$

Expressing these rate constants in terms of reaction probability (P_D and P_{PI}) per Maya frame gives

$$P_D = k_D\Delta t = (2.5\times10^{5}\ \text{s}^{-1})(1.19\times10^{-7}\ \text{s}) = 0.296$$
$$P_{PI} = k_{PI}\Delta t = (1.25\times10^{5}\ \text{s}^{-1})(1.19\times10^{-7}\ \text{s}) = 0.148$$

When added together,

$$P_D + P_{PI} = 0.444\ (= 4/9)$$

which is of course equal to the desired reaction rate in subunits per Maya frame. Now let's look at the minus end dissociation rate.

Dissociation reaction rate

In order to produce steady-state treadmilling behavior in the model, your dissociation rate must balance the plus end association rate—the net subunit count should remain more or less constant throughout the simulation. Given the brisk pace of the nucleotide reactions estimated above we can expect that all subunits reaching the

minus end will be G-ADP-actin. For generality, however, you'll build into the model the ability to handle dissociation reactions for both types of subunit, in case you wish to experiment with the nucleotide rates. Let $Rate_D$ be the dissociation rate of G-ADP in subunits per frame:

By building the rate constants into your model as variables you'll be able to alter them as you like to test their relative effects on treadmilling behavior.

$$\begin{aligned} Rate_D^- = Rate^+ &= (1/9\ \text{monomers/frame})/\ 1.19 \times 10^{-7}\ \text{s/frame} \\ &= 9.34 \times 10^5\ \text{monomers/s.} \end{aligned}$$

Like the nucleotide reactions, dissociation is time-dependent and therefore its rate constant is equal to the reaction rate:

$$k_D^- = 9.34 \times 10^5\ s^{-1}$$

The probability of minus end dissociation in any one Maya frame is given by

$$P_D^- = k_D^-\ \Delta t = (9.34 \times 10^5\ s^{-1})(1.19 \times 10^{-7} s) = 0.111$$

Now, let's consider the improbable (but not impossible!) dissociation of an ADP•Pi subunit. Going to the published rate constants for in vitro actin preparations gives us the ratio

$$k_{PI}^-/k_D^- \approx 0.8/0.3 \approx 2.7$$

Therefore

$$k_{PI}^- = 2.7(9.34 \times 10^5\ s^{-1}) = 2.49 \times 10^6 s^{-1}$$

Therefore the probability of an ADP•Pi subunit dissociating—provided the minus end terminal subunit is ADP•Pi—is given by

$$P_{PI}^- = k_{PI}^-\ \Delta t = (2.49 \times 10^6 s^{-1})(1.19 \times 10^{-7}\ s) = 0.296$$

Expressing the probability of any reaction as a function of the rate constant k provides some flexibility in your model. You could skip the step of deriving k^- and simply enter the P values in your MEL code as floating point numbers (as opposed to calculating P using k and Δt). However, building in the extra step of deriving P allows you to explore the relationship between reaction probabilities, rate constants, and time. Since biochemical systems are often characterized in terms of rate constants and concentrations, having a k parameter in your model provides a valuable link between your in silico Maya laboratory and the world of empirically based experimental science.

With the numbers in place (Table 15.04), let's start building the model.

Methods: **Algorithm design**

In the previous chapter you used a procedure to build your atomic-detail G-actin model. Here we introduce a method that combines animation expressions (which you met back in *Chapter 13*) and procedures. You'll use this method in the remaining chapters to create dynamic simulations. Since Maya, by default, evaluates animation expressions (just *expressions* from here on) once per frame, they allow you to update your model in a stepwise fashion while Maya displays the changes in the scene view. The algorithm flowchart in Figure 15.11 shows two expressions, five procedures, and the

Reaction	Description	Rate constant	Probability
G•ATP + F → F	Association	$1.59 \times 10^3\ \mu M^{-1}\ s^{-1}$	1
F•ATP → F•ADP•Pi	Hydrolysis	2.5×10^5 seconds^{-1}	0.296
F•ADP•Pi → F•ADP + Pi	Phosphate release	1.25×10^5 seconds^{-1}	0.148
F•ADP → G•ADP	Dissociation	9.34×10^5 seconds^{-1}	0.111
F•ADP•Pi → G•ADP•Pi	Dissociation	2.49×10^6 seconds^{-1}	0.296

TABLE 15.04

Initial reaction rate constants and probabilities for each reaction in the model. Probabilities are the likelihood a reaction will occur in one Maya frame.

flow of information between them. These are the MEL code elements of your treadmilling model. As you work through the code listings below, it may help to refer back to Figure 15.11 when you want an overview of how the different pieces of your algorithm fit together.

To begin, let's state clearly what your algorithm should do:

> **Using ready-made geometric models of G- and F-actin, simulate diffusion and reaction events implicated in biopolymer steady-state treadmilling behavior. The model parameters will be updated at the start of each simulation run. The specific events to simulate are:**
>
> 1. **Translational and rotational diffusion within a concentration volume. Diffusing molecules must respond to collisions with one another and with the filament.**
> 2. **Plus end association of G-actin monomers to the filament.**
> 3. **Hydrolysis of bound ATP on filament subunits.**
> 4. **Release of inorganic phosphate from filament subunits.**
> 5. **Minus-end dissociation of subunits from the filament.**

The first element in Figure 15.11 is `reset`, an expression that executes only on frame 1 and resets your model to its initial conditions. It's within this expression that you'll specify the parameters such as the time increment and the treadmilling cycle. Below `reset` is `selfAssembly`, the command-and-control center of the simulation. It gets run by Maya once per frame and calls the various diffusion and reaction procedures in order to query and then update the state of the model. Within `selfAssembly` is a loop that cycles through the G-actin models (named `G_#`), evaluating boundary conditions, association reactions, diffusion, and collisions. The latter three are each handled by a separate procedure. You'll use a fourth procedure called `faderShader()` to meet the project objective of smoothing the transition from newly dissociated G-ADP-actin to association-ready G-ATP-actin.

Once done with the G-actin models, `selfAssembly` calls the `dissociate()` procedure, which draws a random number and compares it to the dissociation probability to determine if the minus end terminal subunit will dissociate from the filament. Finally `selfAssembly` draws random numbers for comparison with the nucleotide reaction probabilities.

Start and stop on ***press Play***

frame = 1 ?

no / yes

reset expression

- Return models to their initial states
- Initialize global variables (Δt, reaction probabilities, etc.)

selfAssembly expression

- For every G-actin:
  - Evaluate boundary conditions
  - If G-actin is within the plus-end reaction volume call **associate()** procedure
  - If G-actin is recently dissociated from the filament, call the **faderShader()** procedure
  - Call the **diffuse()** procedure
  - Call the **collide()** procedure to react to collisions
- Call the **dissociate()** procedure
- For every F-actin subunit, test the probability of:
  - Nucleotide hydrolysis
  - Phosphate release

end of playback range?

no / yes

End

associate()

- Test the probability of a reaction
- If the reaction is successful, associate the G-actin model with the filament

faderShader()

- Depending on how recent the dissociation:
  - Fade out G-actin
  - Move G-actin to join the plus-end monomers
  - Fade in G-actin

diffuse()

- Calculate and return the diffusion values

collide()

- Test for collisions and return an "avoidance" vector

dissociate()

- Test the probability of a reaction
- If successful, dissociate the subunit from the filament and assign it a "faderShader"

FIGURE 15.11

Algorithm flowchart for the treadmilling model. For model nomenclature refer to Table 15.01.

Methods: Encoding the algorithm

As we've done in previous projects, let's build code in pieces relating to the steps outlined in the flowcharts. You'll recall that for the MEL procedures to work in Maya, all lines of code must be entered together, in the same sequence that they appear in the text. Likewise, the animation expression must be entered whole in Maya's Expression Editor. You can find the complete procedure and expression files on the CD-ROM along with a ready-made scene file. You'll find these useful for checking your work or, if you want a very fast results, seeing the model in action right away!

You may recall from *Chapter 13* that we recommend using the file extension ".txt" for animation expression files and ".mel" for procedures. We will continue to use this convention throughout this book.

15_Self_Assembly\mel\associate.mel
\collide.mel
\diffuse.mel
\dissociate.mel
\faderShader.mel
\reset.txt
\selfAssembly.txt

The reset expression

As usual, we recommend building your MEL script in a text editor other than Maya's Script Editor and saving it often as you follow along with the project steps below. Save the file in your Maya Scripts directory under the following name:

reset.txt

Since this expression is to run only when Maya returns to frame 1, begin with a conditional "if" statement to this effect.

To save space, we've foregone the usual file header information such as *authors*, *creation date*, and so on.

```
/* Description:
This animation expression restores F-actin, subunit, and G-actin
models to their original conditions.
*/

if (frame == 1) {

    /***** DECLARE THE VARIABLES *****/

    /*
    $fName          The name of the filament model.
    $gNames         A list of G-actin model names.
    $tracer         The name of the first G-actin monomer to bind
                    the filament.
    $gfGroup        The name of the group node holding the filament
                    subunits.
    $pName          The name of the plus-end locator.
    $mName          The name of the minus-end locator.

    */
    global string $fName, $gNames[], $tracer, $gfGroup, $pName, $mName;
```

```
/*
$subTrans       The distance separating adjacent subunits.
$subRot         The relative rotation of adjacent subunits
                about the filament's long axis.
$Rg             The effective radius of G-actin ~ 2.7
                nanometers.
$Rf             The approximate filament radius.
$Rc             The plus-end concentration volume radius.
$Rr             The plus-end reaction volume radius.
$Av             Avogadro's number which is used to calculate
                chemical concentrations.
$t              The simulation time increment in seconds.
$r              The average diffusion distance for G-actin.
$stdDevTrans    Standard deviation used with Maya's gauss
                command to calculate translational diffusion.
$stdDevRot      Standard deviation used with Maya's gauss
                command to calculate rotational diffusion.
$kpATP_For      Plus-end association reaction rate of ATP-G-actin.
$kmADPpi_Rev    Minus-end dissociation rate constant of
                ADP•Pi-G-actin.
$kmADP_Rev      Minus-end dissociation rate constant of
                ADP-G-actin.
$kADP           Hydrolysis reaction rate.
$kpi            Pi release reaction rate.
$pATP_ForProb   Plus-end association probability of ATP-G-actin.
$mADPpi_RevProb Minus-end dissociation probability of
                ADP•Pi-G-actin.
$mADP_RevProb   Minus-end dissociation probability of ADP-G-actin.
$adpProb        Hydrolysis reaction rate.
$piProb         Pi release reaction rate.
*/
global float $subTrans, $subRot, $Rg, $Rf, $Rc, $Rr, $Av, $t, $r;
global float $stdDevTrans, $stdDevRot, $kpATP_For, $kmADPpi_Rev,
  $kmADP_Rev, $kADP, $kpi;
global float $pATP_ForProb, $mADPpi_RevProb, $mADP_RevProb,
  $adpProb, $piProb;

/*
$totalOn        Counts association (on) reactions.
$totalOff       Counts dissociation (off) reactions.
$fadeSteps      The number of frames taken to fade out a
                dissociated subunit before it rejoins the
                plus-end pool of G-actin.

*/
global int $totalOn, $totalOff, $fadeSteps;

/*
$name           The element in a list of names.
$parent         Used to determine if G-actin is a subunit of
                F-actin.
$origParent     Used to rejoin dissociated subunits with the
                filament when you reset the simulation.
$currentParent  Used with $origParent (above).
$shadeGrp       A shading group node name.
$relatives      A list of filament children, namely subunits,
                group nodes, and plus/minus-end locators.
```

```
$child        An element of the list $relatives.
*/
string $name, $parent, $origParent, $currentParent, $shadeGrp,
  $relatives[], $child;

/*
$trans        Translation vector.
$rot          Rotation vector.
*/
vector $trans, $rot;

/*
$fLength      The starting length of the F-actin filament.
$yPos         The absolute value of the Y-position of the plus-
              and minus-end locators relative to the filament.
$tD           The translation diffusion constant.
$rD           The rotation diffusion constant.
$pi           The trigonometric constant pi.
$Vc           The plus-end concentration volume.
$Vr           The spherical reaction volume centered at the
              plus-end.
$c            The concentration of free G-actin, that is, the
              time for one subunit to flux through the filament.
$ratio        The ratio of the in vitro reaction rate constants
              $kmADPpi_Rev and $kmATP_Rev.
$fluxRate     The flux rate per frame of subunits through the
              filament.
*/
float $fLength, $yPos, $tD, $rD, $pi, $Vc, $Vr, $c, $ratio,
  $fluxRate;

/*
$cycle        The average number of frames for one treadmilling
              cycle.
$state        The value of the G-actin state attribute.
              = 0 for free G-actin
              = 1 for F-ATP-actin
              = 2 for F-ADP•Pi-actin
              = 3 for F-ADP-actin
$fNum         The filament number. In the present model there is
              one filament, with $fNum == 0.
$gCount       The number of G-actin models.
$subunits     The number of F-actin subunits.
*/
int $cycle, $state, $fNum, $gCount, $subunits;
```

In the next section, you'll set the values used in diffusion, collision, and reaction probability calculations. All distances are given in nanometers and converted to meters where necessary. To simulate collisions in your model, you can treat G-actin as more or less spherical. Below, we assign this "spherical" G-actin an effective collision radius, Rg, of 2.7 nm. If you change this value, be sure to update the reaction radius values as well (see page 407). The F-actin radius R_F is used throughout the simulation to change the length of the filament via the NURBS cylinder `heightRatio` attribute. You'll set the `$totalOn` and `$totalOff` variables to count reaction through each simulation run. This will tell you how close your model is performing to the specifications we outlined earlier in this chapter.

```
/***** INITIALIZE THE VARIABLES *****/

$fName = "F_0";
$gNames =`ls -tr "G_*"`;
$tracer = " ";
$gfGroup = $fName + "|" + "gfGroup";
$pName = $fName + "|" + "pmGroup" + "|" + "plus";
$mName = $fName + "|" + "pmGroup" + "|" + "minus";
$subTrans = 2.8; // Nanometers.
$subRot = -166; // Degrees.
$Rg = 2.7; // Nanometers.
$Rf = 3.5; // Nanometers.
$Rc = 15; // Nanometers.
$Rr = 4.2; // Nanometers.
$Av = 6.022 * `pow 10 23`;
$r = 1.0; // Average diffusion distance.
$totalOn = $totalOff = 0;
$fadeSteps = 10;
$tD = 1.65 * pow(10, -12);
$rD = 1.98 * pow(10, 5);
$pi = 3.14159265;
$Vc = 4.0 / 3.0 * $pi * pow(($Rc*pow(10,-9)), 3) * 1000.0; // Litres.
$Vr = 4.0 / 3.0 * $pi * pow(($Rr*pow(10,-9)), 3) * 1000.0; // Litres.
$cycle = 180;
$subunits =`getAttr ($fName + ".subunitsOrig")`;
$gCount =`size $gNames` - $subunits;

// Time and distance.
float $tmp = $r * pow(10, -9); // $r in meters.
$t = $pi * pow($tmp, 2) / (16 * $tD);
$fluxRate = (float) $subunits/$cycle; // Subunits per second.

// Concentration.
$c = $gCount / $Vc / $Av * pow(10,6); // Micromolar

// Diffusion.
$stdDevTrans = sqrt(2.0 * $tD * $t) * pow(10, 9);
$stdDevRot = sqrt(2.0 * $rD * $t);
// Convert from radians to degrees.
$stdDevRot = $stdDevRot * 360/(2 * $pi);
```

Recall from *Chapter 12* that placing **(float)** in front of a value forces it to be of type float. This is called **explicit typing** and is often necessary if you want a floating point value as the result of an expression involving integers.

The following rate constant and probability calculations are the MEL code version of the derivations we presented earlier in this chapter.

```
/***** REACTION PROBABILITIES *****/

// Plus-end reaction rate constant.
$kpATP_For = $fluxRate / $t / $c; // Molecules per Micromolar
  second.

// Plus-end association probability for monomer entering
  reaction volume.
$pATP_ForProb = $kpATP_For * $t / ($Av * $Vr) / pow(10, -6);

// Minus-end reaction rate constants.
$kmADP_Rev = $fluxRate / $t; // Molecules per second.
```

```
$ratio = 0.8 / 0.3;
$kmADPpi_Rev = $fluxRate / $t * $ratio;

// Minus-end dissociation probabilities.
$mADPpi_RevProb = $kmADPpi_Rev * $t;
$mADP_RevProb = $kmADP_Rev * $t;

// Hydrolysis reaction rate probabilities.
$kADP = 2.5 * pow(10,6);
$adpProb = $kADP * $t;
$kpi  = 1.25 * pow(10,6);
$piProb = $kpi  * $t;
```

We find it helpful to have the key parameters at hand when running a simulation model in Maya. The next lines print to the History panel of the Script Editor at the start of each simulation run.

Remember that the "**\n**" string causes a line break in the printed information.

```
// Print model parameters.
print "\n\n***** RESETTING *****\n\n";
print ("gCount = " + $gCount + "\n");
print ("conc = " + $c + "\n");
print ("concVol  = " + $Vc + "\n");
print ("fluxRate = " + $fluxRate + "\n");
print ("rxnRad = " + $Rr + "\n");
print ("rxnVol  = " + $Vr + "\n");
print ("t = " + $t + "\n");
print ("diffusion step stdDev = " + $stdDevTrans + "\n");
print ("diffusion rotation stdDev = " + $stdDevRot + "\n\n");
print "\n*** Probabilities ***\n\n";
print ("$kpATP_For = " + $kpATP_For + "\t\t$pATP_ForProb =
        " + $pATP_ForProb + "\n");
print ("$kmADPpi_Rev = " + $kmADPpi_Rev + "\t$mADPpi_RevProb =
        " + $mADPpi_RevProb + "\n");
print ("$kmADP_Rev = " + $kmADP_Rev + "\t$mADP_RevProb = " +
        $mADP_RevProb + "\n");
print ("$adpProb = " + $adpProb + "\n");
print ("$piProb = " + $piProb + "\n");
```

The vertical bar "|" is used to construct path names within Maya.

The following command selects a list of all G-actin models in your scene: select ls -tr "G_*";

Substituting the following inside the quotes will select only those that are parented to the F-actin filament (i.e. subunits): "F_0|gfGroup|G_*"

Your filament and G-actin models have been assigned custom attributes which are listed in Table 15.05. The "Orig" attributes are used below to reset the transform and custom attributes to their starting values. The .state attribute tracks the nucleotide state of each G-actin model (G_#). The filament model's .heightRatio attribute is connect to and drives the .heightRatio attribute belonging to the NURBS cylinder creation node. As your F-actin model grows and shrinks, your MEL code sets the .heightRatio value on F_0, which in turn drives the cylinder height.

```
/***** RESET THE FILAMENT MODEL *****/

$gCount = `size $gNames`; // A list of all G-actin and subunits.

// Center the filament's pivot point.
xform -centerPivots ($fName + "|gfGroup.translate");
```

Model	Attribute	Initial value	Description
F_0	transOrigX transOrigY transOrigZ	0 0 0	Initial translate values
	rotOrigX rotOrigY rotOrigZ	0 0 0	Initial rotate values
	Subunits	20	Shows you the number of filament subunits at a glance
	subunitsOrig	20	Initial number of subunits
	heightRatio	16.914	**Ratio of F-actin length to radius** (used to drive the height of the NURBS cylinder model which is the F-actin collision surface)
G_#	transOrigX transOrigY transOrigZ	Varies	Initial translate values
	rotOrigX rotOrigY rotOrigZ	Varies	Initial rotate values
	State stateOrig	Nucleotide state	
		0	Free G-actin
		1	ATP-F-actin
		2	F-ADP•Pi-actin
		3	F-ADP-actin
	filOrig	–1	Initially a free G-actin monomer
		0	Initially an F-actin subunit belonging to F_0

TABLE 15.05

Custom attributes.

```
// Reset the subunit group node.
setAttr ($fName + "|gfGroup.translate") 0 0 0;

// Reset the subunits attribute to its original value.
setAttr ($fName + ".subunits") $subunits;
```

```
/*
Reset the NURBS cylinder length. Length is connected to the
heightRatio attribute of the actin model's history node. The
length of F-actin = 2.8nm per subunit * (original number of
subunits -1). Adding 6nm to enclose the distal subunits gives a
starting length of 59nm for 20 subunits.
*/
$fLength = (float) 6.0 + $subTrans * ($subunits - 1);
setAttr ($fName + ".heightRatio") ($fLength/$Rf);

// Reset plus- and minus-end locators.
$yPos = (float)($subunits + 1)/2*$subTrans;
setAttr ($fName + "|pmGroup|plus.ty") ($yPos);
setAttr ($fName + "|pmGroup|minus.ty") (-$yPos);
setAttr ($fName + "|pmGroup|plus.ry") ($subRot*($subunits + 1)/2);
setAttr ($fName + "|pmGroup|minus.ry") (-$subRot*($subunits + 1)/2);

// Reset the F-actin model to its starting position.
$trans = `getAttr ($fName + ".transOrig")`;
setAttr ($fName + ".translate") -type double3 ($trans.x)
  ($trans.y) ($trans.z);

select -clear; // Clear the current selection.

/***** RESET THE G-ACTIN AND SUBUNIT MODELS *****/

for ($name in $gNames) { // G-actin and F-actin subunits.

    // Get the state of the current molecule.
    $state = `getAttr ($name + ".stateOrig")`;

    if ($state == 0) { // The current model was a G-actin
      monomer to begin.

        $parent = `firstParentOf $name`;
        if ($parent != "") {

            /*
            The model currently has a parent and was therefore
            incorporated into a filament during the previous
            simulation run. Now parent G-actin to world space.
            */
            parent -world $name;
        }
        // Set the shading group.
        $shadeGrp = "ATP_shaderSG";
    }
    else { // The current model was originally an F-actin
      subunit.

         /* Parent the model to its F-actin filament if it
           became dissociated during the last run. */
         $currentParent = `firstParentOf $name`;
         $fNum = `getAttr ($name + ".filOrig")`;
         $origParent = "|F_" + $fNum + "|gfGroup";
```

Parenting an object to world space (parent-world) removes it from the hierarchy of its current parent.

```
            if ($fNum > -1 && $origParent != $currentParent) {
                parent $name $origParent;

            }

            // Determine the appropriate shader for this model.
            if ($state == 1) {
                // The current model was an F-actin ATP subunit.
                // Set the shading group.
                $shadeGrp = "ATP_shaderSG";
            }
            else if ($state == 2) $shadeGrp = "ADPpi_shaderSG";
            else if ($state == 3) $shadeGrp = "ADP_shaderSG";

        } // End else.

        // Reset the G-actin model to its starting position.
        $trans = `getAttr ($name + ".transOrig")`;
        setAttr ($name + ".translate") -type double3 ($trans.x)
          ($trans.y) ($trans.z);

        // Reset the G-actin model to its starting rotation.
        $rot = `getAttr ($name + ".rotOrig")`;
        setAttr ($name + ".rotate") -type double3 ($rot.x) ($rot.y)
          ($rot.z);

        // Reset G-actin custom state attribute.
        setAttr ($name + ".state") $state;

        // Put the G-actin in the appropriate shading group.
        sets -e -forceElement $shadeGrp $name;

        // Clear the current selection.
        select -clear;

    } // End "for ($name in $gNames)".
```

The "select -clear" statement makes it so that the final G-actin to be reset doesn't remain selected when this expression finishes.

This next bit of code uses Maya's reorder command to put the subunit model in alphanumeric order within the filament cylinder model. This is not essential for the simulation to function, but keeps your objects tidy in the Outliner.

```
    // Put F-actin subunits in their proper alphanumeric order.
    $relatives = sort(`listRelatives -path ($fName + "|gfGroup")`);
    for ($child in $relatives) reorder -back $child;
```

Finally, delete any temporary "faderShader" that were in use when the simulation run stopped.

```
    // Delete faderShaders.
    delete `ls "faderShader*"`;

} // End if (frame == 1).
// End expression.
```

Save your text file and start a new one for the next animation expression.

The selfAssembly expression

Save this file in your Maya Scripts directory under the following name:

selfAssembly.txt

Unlike reset, this expression must run once every frame greater than frame 1. Again, this condition can be tested with an "if" statement.

Reminder: variables that have been described previously will not be given a description here or with subsequent occurrences.

```
/* Description:
This is a runtime animation expression that updates the actin
assembly simulation. Diffusion, collision avoidance, and reactions
are parcelled off as procedures that are called when needed from
this expression.
*/

if (frame > 1) {

    /***** DECLARE THE VARIABLES *****/

    global string $fName, $gNames[], $tracer, $pName;
    global float $Rg, $Rf, $Rc, $Rr, $adpProb, $piProb;
    global int $totalOn;

    /*
    $plusSubunit     The current plus-end subunit.
    $minusSubunit    The current minus-end subunit.
    */
    string $plusSubunit, $minusSubunit, $name, $gName,
      $relatives[];

    /*
    $gTrans          Current position of the current G-actin molecule.
    $gRot            Current rotation of the current G-actin molecule.
    $pTrans          Current position of the plus-end locator.
    $separation      The vector separating G-actin and the filament
                     plus-end.
    $trans           The sum of diffusion, collision, and bounding
                     vectors.
    $rot             The diffusional rotation vector.
    $bounding        Nudges the G-actin back inside the reaction
                     volume.
    $diffusion       The return value of the diffuse() procedure.
    $collision       The return value of the collide() procedure.
    */
    vector $gTrans, $gRot, $pTrans, $separation, $trans, $rot,
                     $bounding, $diffusion[], $collision;

    /*
    $dist            The scalar distance of the vector $separation.
    $RrPrime         The distance used to test if G-actin lies within
                     the association reaction volume.
    $rnd             A random number between 0 and 1.
    */
    float $dist, $RrPrime, $rnd;

    /*
    $bindThisFrame    Set to 1 if an association reaction has
                      occurred in the current frame, 0 if not.
```

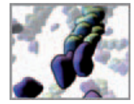
```
$bindThisMonomer Set to 1 if the current G-actin has associated
                 with F-actin, 0 if not.
The above values are set in the associate() procedure.
$mState          The nucleotide state of the minus-end subunit.
*/
int $bindThisFrame, $bindThisMonomer, $mState, $state,
  $subunits;

/***** INITIALIZE THE VARIABLES *****/

$state = 0;
$bindThisFrame = 0;
$bindThisMonomer = 0;

/*
Query the coordinates of the plus- and minus-end locators.
xform queries the world matrix for the specified object.
*/
$pTrans = `xform -query -worldSpace -translation $pName`;

// Refresh the list of G-actin monomers.
$gNames = `ls -tr "|G_*"`;
```

The first time you see a command used in the code listings here (e.g. `xform`) the long flag names will be used. Subsequently, the short names will be used.

Here, your expression will assess bounding, association reactions, diffusion, and collisions for every free G-actin monomer. You'll use the "for in" conditional statement to loop through a list of G-actin names. If a given monomer associates with the filament, you need not calculate its diffusion or check for collisions. Therefore you'll track the "recently associated" status of the monomer using the variable `$bindThisMonomer`. Its value will be set to 1 if an association reaction occurs and 0 if not. A similar variable `$bindThisFrame` tracks whether an association reaction has occurred yet during the present frame. If one has, `$bindThisFrame` = 1 prevents the expression from evaluating any more binding opportunities. This limits the model to binding a maximum of one monomer per frame. Note the totalOn variable which has 1 added to its value each time a successful association reaction occurs.

Bounding, diffusion, and collision each produce a vector (with direction and magnitude) (Figure 15.12). After all three have been evaluated, their effects are combined (vectors added) and used to update the transform of the G-actin model.

```
/***** G-ACTIN BOUNDING, REACTION, AND DIFFUSION *****/
for ($name in $gNames) {

    // Zero the bounding vector for this G-actin.
    $bounding = <<0, 0, 0>>;
    $bindThisMonomer = 0;

    // Query the position and state of monomer $name.
    $gTrans = `getAttr ($name + ".t")`;
    $state = `getAttr ($name + ".state")`;

    if ($state == 0) { // This G-actin is ready for binding.

        // Calculate the distance between G-actin and the
          plus-end.
        $separation = $pTrans - $gTrans;
        $dist = mag($separation);
```



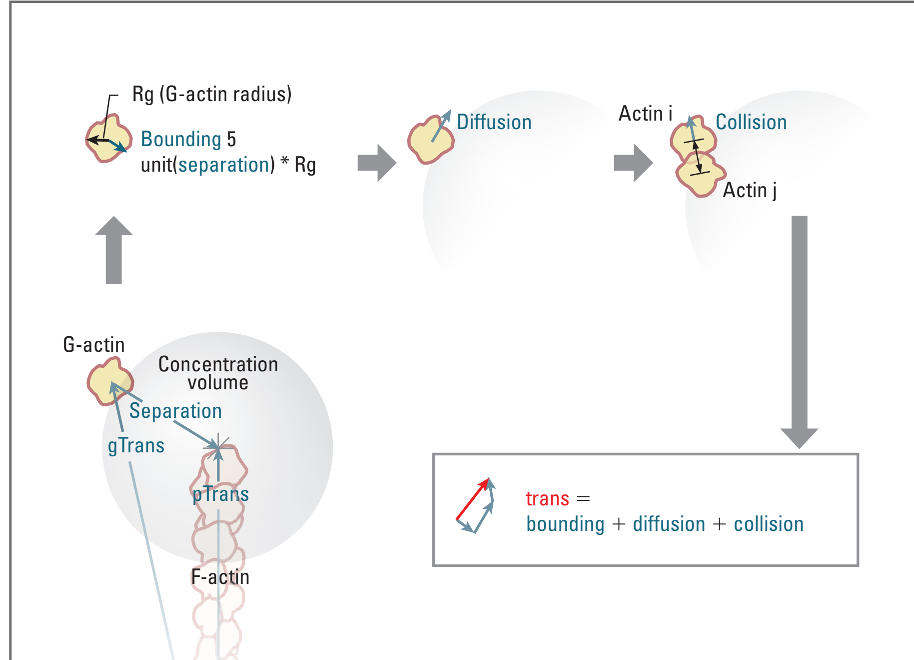


FIGURE 15.12

Bounding, diffusion, and collision evaluations each result in a displacement vector. The three vectors are summed and used

The if($bindThisFrame) notation evaluates $bindThisFrame as if it were of type Boolean with two possible values: 1 (yes) and 0 (no). If the variable being tested has only 0 and 1 as possible values, this notation does the same thing in fewer characters than if($bindThisFrame == 1). The opposite of if($bindThisFrame) is

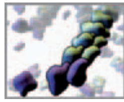
```
        // The distance between the G-actin and reaction volume
          surfaces.
        $RrPrime = $Rr + $Rf + $Rg;

        // Determine if G-actin is outside the concentration
          volume.
        if ($dist > ($Rc - $Rg)) {
            // Calculate a vector to nudge G-actin back inside.
            $bounding = unit($separation) * $Rg;
        }
        else if ($dist <= $RrPrime && $bindThisFrame == 0) {

            /* The monomer can associate with the filament.
            Call the association reaction procedure.
            */
            $bindThisFrame = associate($name);

            if ($bindThisFrame) { // 1 if yes, 0 if no.
                // Increment the "on" reaction counter.
                $totalOn++;
                /*
                The current G-actin has bound the filament
                and will not be considered for diffusion,
                collisions, and bounding below.
                */
                $bindThisMonomer = 1;
            }
        }
    }
    else {
```

```
        /*
        This is a recently dissociated G-actin and is in the
        process of fading in or out. Call the faderShader()
        procedure.
        */
        faderShader($name, $state);
    }
```

If the current G-actin model's state is not 0, then it must have recently dissociated and is either fading out (state < 0) near the filament minus end or fading back in (state > 0) near the plus end. In either case, the `faderShader()` procedure is called to fade out, fade in, or move G-actin to the filament plus end region. We'll explore this procedure a little later in this chapter. Provided no association reaction occurred, the next bit of code calls the `diffusion()` and `collision()` procedures. Their results are added to the `$bounding` vector from above to make a new vector called `$trans`. `$trans` is then used to update the G-actin model.

```
    /*
    If $name did not associate with the filament, calculate its
    diffusion, avoidance and bounding vectors.
    */
    if ($bindThisMonomer == 0) { // No binding occurred.

        // Call the diffusion procedure.
        $diffusion = diffuse();

        // Call the collision avoidance procedure.
        $collision = collide($name);

        // Total the motion for the G-actin molecule.
        $trans = $diffusion[0] + $collision + $bounding;
        $rot = $diffusion[1];

        /*
        Query the current translation and rotation of the
        molecule. Since faderShader() may have moved the
        G-actin to the filament plus-end, it's essential to
        query G-actin's translate attribute again here.
        */
        $gTrans = `getAttr ($name + ".translate")`;
        $gRot = `getAttr ($name + ".rotate")`;

        /* Add the current translation and rotation values to
          the $trans and $rot vectors. */
        $trans = $trans + $gTrans;
        $rot = $rot + $gRot;

        // Set the molecule's new translation and rotation values.
        setAttr ($name + ".translate") -type double3 ($trans.x)
          ($trans.y) ($trans.z);
        setAttr ($name + ".rotate") -type double3 ($rot.x)
          ($rot.y) ($rot.z);
    }
} // End for ($name in $gNames).
```

Here you'll use the `ls` command to get a list of F-actin subunits and count its size. The list begins with the name of the minus end subunit and ends with that of the plus end subunit. In other words, in the array (`$relatives[]`) used to store this list, element 0 is the minus end subunit and element [`$subunits`] is the plus end subunit.




```
/***** INTRA-FILAMENT REACTIONS *****";

// Get a list of F-actin subunits and count its size.
$relatives = `listRelatives -path ($fName + "|gfGroup")`;
$subunits = `size $relatives`;
$minusSubunit = $relatives[0];

// Get the nucleotide state of the minus-end subunit.
$mState = `getAttr ($minusSubunit + ".state")`;
```

Actin oligomers of two subunits are highly unstable, meaning they are quite likely to dissociate into G-actin monomers. However, once a third subunit is added, the structure—now called a *nucleus*—is much more likely to remain stable and grow into a filament. Therefore, we'll set minimum stable filament size to three subunits and not allow further dissociation reactions if your filament happens to reach the three-subunit size. We'll present the dissociate procedure() later in this chapter.

The sets command -fe flag is short for -forceElement which forces the addition of the item ($gName) to the set (the shading group node) if it currently belongs to another set.

```
// The minimum filament size is 3 subunits.
if ($subunits > 3) {
        // Call the dissociate() procedure.
        dissociate($minusSubunit, $mState);
}

for ($gName in $relatives) {

  // Get the state of the current molecule.
  $state = `getAttr ($gName + ".state")`;
  // Generate a random number with which to test reaction
    probabilities.
  $rnd = `rand 1`;
  if ($state == 1) { // ATP subunits.

      if ($rnd < $adpProb) { // A reaction occurs.
          // Change state and set the shading group.
          setAttr ($gName + ".state") 2;
          /* If the current G-actin is not the tracer,
            assign the ADP•PI shader. */
          if ($tracer != $gName) sets -e -fe "ADPpi_
            shaderSG" $gName;
      }
  }
  else if ($state == 2) { // ADP•Pi subunits.


      if ($rnd < $piProb) { // A reaction occurs.
          // Change state and set the shading group.
          setAttr ($gName + ".state") 3;
          /* If the current G-actin is not the tracer, assign
            the ADP shader. */
          if ($tracer != $gName) sets -e -fe "ADP_
            shaderSG" $gName;
      }
  }
}
```

```
        // Clear the current selection.
        select -clear;

    } // End if (frame > 1).
    // End expression.
```

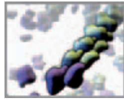
Next we'll cover the five procedures in the order they're called by selfAssembly.

The associate() procedure

This procedure is called only if a G-actin monomer breaches the reaction volume centered at the filament plus end. It takes the monomer name (`$gName`) as its only argument. The test for this "breach" is carried out in the selfAssembly expression. Since we calculated the size of the reaction volume to correspond to a binding probability of one, each call to `associate()` will result in a binding reaction—at least for now. If you choose later to vary the reaction volume and probability, the probability test you'll build into the model below will come in handy. Note that this procedure is of type int, which means it will return an integer to `selfAssembly`: 1 if a reaction is successful, 0 if not.

```
    /* Description:
    This procedure assesses the probability for binding G-actin to
    F-actin If a reaction occurs, G-actin is parented to F-actin and
    both models are updated.
    */

    global proc int associate(string $gName) {

        /***** DECLARE THE VARIABLES *****/

        global float $subTrans, $subRot, $pATP_ForProb, $Rf;
        global string $fName, $tracer, $gfGroup, $pName, $mName;

        /*
        $second   The name of the second subunit from the plus-end.
        */
        string $second;

        /*
        $plusTy   The plus-end translateY value.
        $minusTy  The minus-end translateY value.
        $plusRy   The plus-end rotateY value.
        $minusRy  The minus-end rotateY value.
        The above variables are used to reposition the plus and minus-
        ends after a binding event.
        */
        float $plusTy, $minusTy, $plusRy, $minusRy, $rnd;

        /*
        $bound    Set to 1 if a successful reaction occurs and 0 if not.
        $sState   The nucleotide state of the second subunit from the
                  plus-end. This must be state 3 for an association
                  reaction to be possible.
        */
        int $bound, $sState, $subunits;
```

Below you'll set $bound to 0 and call a random number to test the reaction probability, and query the name and state of the penultimate plus end subunit. Its state must



be 3, indicating an ADP subunit, for a monomer to bind the filament (for a reminder of the rationale for this condition, refer back to page 407).

```
/***** INITIALIZE THE VARIABLES *****/

$bound = 0;
$rnd = `rand 1.0`;
$relatives = `listRelatives -path ($fName + "|gfGroup")`;
$subunits = `size $relatives`;
// Get the state of the penultimate plus-end subunit.
$second = $relatives[($subunits - 2)];
$sState = `getAttr ($second + ".state")`;

// Print a message to the Script Editor.
print ("\nInside associate(), $sState = " + $sState + "\n");
```

The `rand()` command generates a random number between limits specified by its arguments. If only one argument is provided, as we've done above, the other limit is zero by default. The result of the above statement is therefore a decimal number between zero and one, which is the range of probabilities for an event to occur. Below you'll compare $rnd to the probability for G-ATP binding, `$pATP_ForProb`. Since we've set the probability for this reaction to one, the following condition will be met and the step below it will proceed.

```
/***** TEST THE BINDING PROBABILITY *****/
if ($rnd < $pATP_ForProb && $sState == 3) {

    // G-actin will bind to the Plus end of the filament,
      therefore
    $bound = 1;

    print $gName;
    print "will associate with the filament\n";

    // Adjust the plus- and minus-end locators for the next
      reaction.
    $plusTy = `getAttr ($pName + ".ty")`;
    $minusTy = `getAttr ($mName + ".ty")`;
```

Shift the plus and minus end locators by half the subunit spacing (`$subTrans` = 2.8 nm) to account for the addition of a new subunit.

```
    setAttr ($pName + ".ty") ($plusTy + $subTrans/2);
    setAttr ($mName + ".ty") ($minusTy - $subTrans/2);
    $plusRy = `getAttr ($pName + ".ry")`;
    setAttr ($pName + ".ry") ($plusRy + $subRot);
```

Here, you'll put the G-actin at the world origin to zero its translate and rotate attributes. Next, you'll put the G-actin into a new group node that you'll connect to the F-actin translate and rotate attributes. Finally, move the G-actin to its proper position and rotation *relative* to the filament origin, ungroup the G-actin and delete the group. The net result of these steps is that the G-actin translate and rotate attributes will represent the subunit's position and orientation relative to its parent filament's transform node. The translate and rotate values are far simpler to interpret when treated this way when compared to the equivalent values in world space.

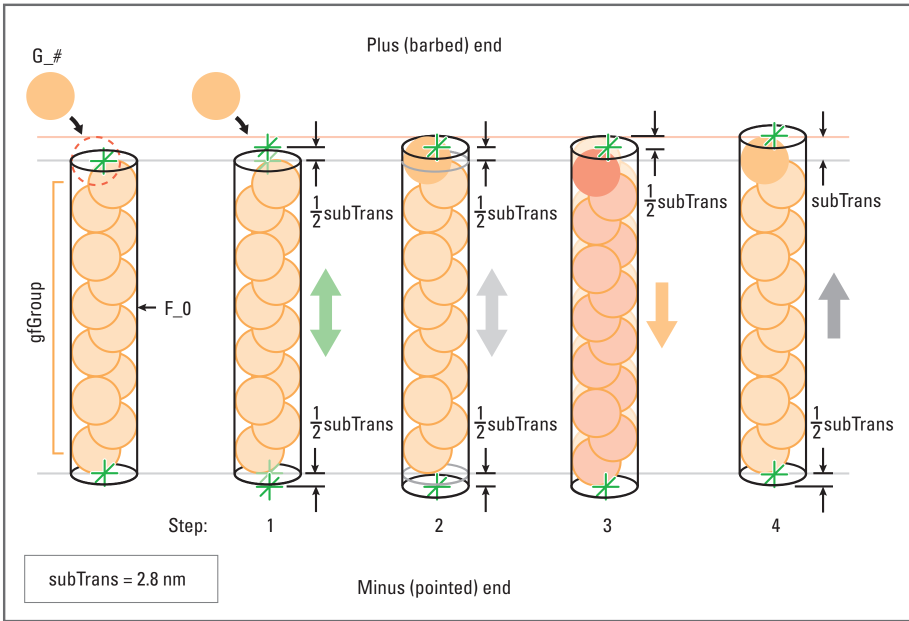


FIGURE 15.13

The plus end binding of a monomer into the Maya filament model is executed as a series of steps within the association() procedure.

Step 1: Move the plus- and minus-end locators relative to the filament (F_0) by 1/2 the subunit spacing (1/2 subTrans).

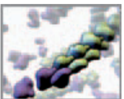
Step 2: Parent the G-actin model (G_#) to the gfGroup which is a child of F_0. Lengthen F_0 by 1/2 subTrans.

Step 3: Move gfGroup toward the minus end to center gfGroup vertically in F_0.

Step 4: Move F_0 in world space by subTrans toward its plus-end. In the end the filament minus-end winds up in the same spot it began, the filament is longer by one subunit, and the locators indicate the new binding locations.

Figure 15.13 describes the parenting and shifting of nodes to accommodate the new subunit that follows below.

```
/***** PUT A G-ACTIN INTO THE FILAMENT *****/

// Locate the G-actin model at the world origin.
setAttr ($gName + ".translate") 0 0 0;
setAttr ($gName + ".rotate") 0 0 0;

// Create a new empty (-em) group to hold the G-actin.
group -n group1 -em;
// Parent the newly bound G-actin to group1.
parent $gName group1;

// Connect the new group's transform node to the F-actin
  transform.
connectAttr -force ($fName + ".translate") group1.
  translate;
connectAttr -f ($fName + ".rotate") group1.rotate;

/*
Rotate and move the G-actin relative to its parent, group1,
using the plus-end locator rotate and translate values.
*/
rotate -r -os 0 $plusRy 0 $gName;
move -r -os -wd 0 $plusTy 0 $gName;

/*
Parent G-actin to gfGroup so that it moves with the other
subunits when the next G-actin monomer is added on.
*/
parent $gName $gfGroup;
```




```
/*
Center the gfGroup's pivot point, now that you've added
another subunit to it.
*/
xform -cp $gfGroup;

// Delete group1 since it's no longer needed.
  delete group1;

// Shift gfGroup by half a subunit height to center it
  within F-actin.
move -r -os -wd 0 (-$subTrans/2) 0 $gfGroup;

// Move the filament by half a subunit to reflect its
  increase in length.
move -r -os -wd 0 ($subTrans/2) 0 $fName;

// Update the F-actin's .subunits and .heightRatio attributes.
  string $tmpStr = $fName + ".subunits";
int $tmp = `getAttr $tmpStr`;
setAttr $tmpStr ($tmp + 1);
setAttr ($fName + ".heightRatio") (((float) 6.0 + $subTrans
  * ($tmp)) /$Rf);

// Update the G-actin's .state attribute.
setAttr ($gName + ".state") 1;
```

To visualize the flux of a subunit through the filament, you can assign a unique shader to one subunit per treadmilling cycle. We'll call this subunit the "tracer" and connect it to a blinking shader called `tracer_shader`. `$tracer` is a global variable that stores that name of the G-actin tracer model. `tracer_shader` exists prebuilt in the `treadmilling.ma` scene file. Once a subunit becomes the tracer, it remains so until it dissociates from the filament, at which point the very next subunit to associate at the plus end become the new tracer. The first tracer will be the first monomer to join the filament once the simulation begins.

```
        /* If no tracer object exists make this subunit the tracer
        and assign it the tracer_shader. */
        if ($tracer == "") {
            $tracer = $gName;
            sets -e -forceElement tracer_shaderSG $gName;

        }
    } else {
        print $gName;
        print "will not associate with the filament\n";
    }

    // Send a return value back to the expression that called this
      procedure.
    return $bound;

} // End procedure.
```

If you decide to vary the association binding probability, the print statement above reports failed binding attempts in the Script Editor History panel. Save this procedure in text file in your Maya Scripts directory under the following name:

associate.mel

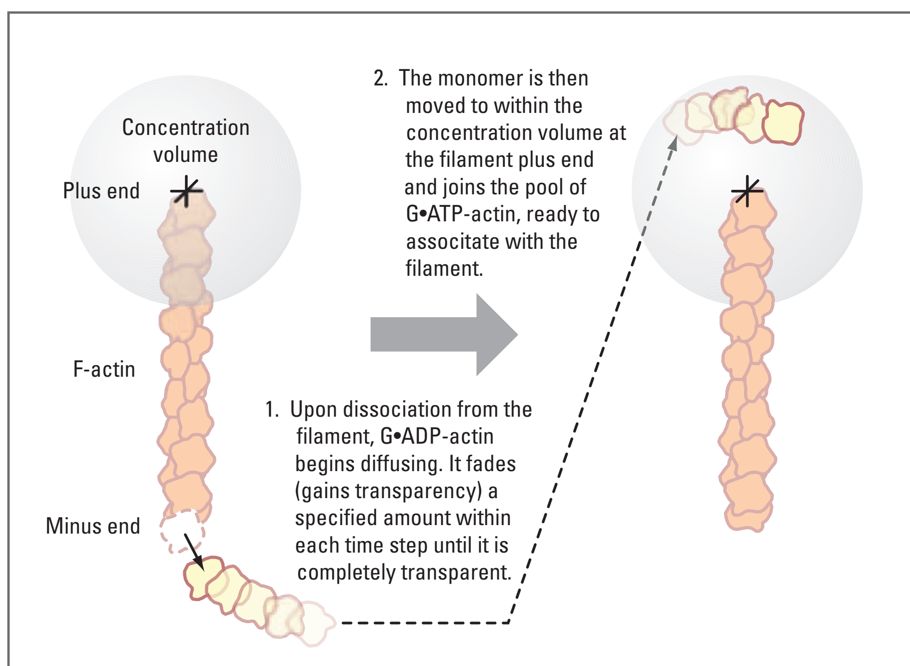


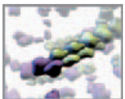
FIGURE 15.14

The transition of a monomer from newly dissociated ADP-actin subunit to ATP-actin ready to bind the plus-end of the filament is governed by the `faderShader()` procedure. The purpose is to smooth the transition visually.

The faderShader() procedure

After a subunit dissociates from the filament it transitions from the G-ADP state to the G-ATP state and joins the group of diffusion monomers near the filament plus end (Figure 15.14). The purpose of this procedure is to ease these transitions visually, which is in keeping with our interpretive visualization goals of this project. You can set the number of transition steps via the `$fadeSteps` variable in the reset expression. Setting its value to 1 causes a dissociated G-actin to join the plus end group one frame after it dissociates, without fading out and back in.

```
/* Description:
This procedure fades out newly dissociated monomers, moves them to the
plus end of the filament, then fades them back in.
*/

global proc faderShader(string $gName, int $state) {

    /***** DECLARE THE VARIABLES *****/

    global float $Rc;
    global string $fName, $pName;
    global int $fadeSteps;

    /*
    $transp             The transparency attribute value for the
                        faderShader.
    $newTrans           The new position of G-actin after being
                        transported to plus-end.
    */
    vector $transp, $newTrans, $pTrans;
```




```
/*
$increment      The incremental increase or decrease in
                transparency.
*/
float $increment;

/*
$shaderName     The name of the faderShader for the current
                G-actin.
*/
string $shaderName;
```

Each faderShader will be unique to the monomer for which it was created. This makes it possible to fade multiple monomers at once, each at a different stage in its transition to the plus end. The shader name will therefore include the name of its monomer. When the monomer has faded back in and joined the other diffusing plus end monomers, it will be assigned the default ATP shader, at which point its custom faderShader can be deleted from the scene.

```
/***** INITIALIZE THE VARIABLES *****/

$increment = 1.0/ $fadeSteps;
$shaderName = "faderShader" + $gName;

/***** TEST THE MONOMER'S STATE *****/
```

Free G-actin with a state attribute value other than 0 is either fading out (state <-1), transferring to the filament plus end (state $==-1$), or fading in (state >0).

```
if ($state < -1) { // Fade out.
    $transp = `getAttr ($shaderName+".transparency")`;
    $transp = $transp + <<$increment, $increment, $increment>>;
    setAttr ($shaderName + ".transparency") -type double3
      ($transp.x) ($transp.y) ($transp.z);
    setAttr ($gName + ".state") ($state + 1);
}
```

Once the G-actin's state attribute has incremented to –1, it's time to move it to a new location near the filament plus end. Rather than picking a location anywhere within the concentration bounding volume—which could likely land it within the reaction volume—let's introduce it to the plus end region as if it had wandered in from afar. To do this, you can take advantage of Maya's sphrand command. sphrand generates random numbers (single numbers and vectors) that lie within a spherical volume of radius R, where R is the command's argument.

```
else if ($state == -1) { // Move to the plus-end.

    // Query the location of the plus-end locator.
    $pTrans = `getAttr ($pName+".worldPosition")`;
    // Generate a random vector within the concentration
      sphere.
    $newTrans = sphrand($Rc);
    // Move the vector's origin to the plus-end.
    $newTrans = $newTrans+$pTrans;

    // Move the monomer and set its .state attribute.
    setAttr ($gName + ".translate") -type double3 ($newTrans.x)
            ($newTrans.y) ($newTrans.z);
    setAttr ($gName + ".state") ($fadeSteps + 1);
```

```
        // Set the shader colour to match the ATP_shader.
        setAttr ($shaderName + ".color") 1 1 0.8;
    }
    else if ($state > 1) { // Fade in.

        $transp = `getAttr ($shaderName+".transparency")`;
        $transp = $transp - <<$increment, $increment, $increment>>;
        setAttr ($shaderName + ".transparency") -type double3
          ($transp.x) ($transp.y) ($transp.z);
        setAttr ($gName + ".state") ($state - 1);
    }
```

Once the monomer has faded back in, assign it the default ATP shader and set its state attribute to 0. The latter step makes this G-actin once again a candidate for association reactions.

```
    // Swap shaders and set G-actin's state to 0.
    else if ($state == 1) {

        // Replace faderShader with ATP_shader and set .state
          attribute to 0.
        sets -e -forceElement ATP_shaderSG $gName;
        setAttr ($gName+".state") 0;
        delete $shaderName; // No longer needed.
    }
} // End procedure.
```

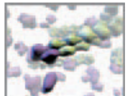
Save this procedure in text file in your Maya Scripts directory under the following name:

faderShader.mel

The diffuse() procedure

This procedure is called for every G-actin monomer in the simulation and returns a translation and a rotation vector. It employs the method described starting on page 399 to query random diffusion values via Maya's gauss command. The arguments passed to gauss were calculated using Δt and the diffusion coefficients, `$stdDevTrans` and `$stdDevRot`, which were initialized in the reset expression. Unlike the `associate()` procedure this one returns an array of vectors: `$trans` and `$rot`.

```
/* Description:
This procedure returns translation and rotation diffusion vectors
via Maya's gauss command.
*/

global proc vector[] diffuse() {

    /***** DECLARE THE VARIABLES *****/

    global float $stdDevTrans, $stdDevRot;

    /*
    $trans          The translational diffusion vector.
    $rot            The rotational diffusion vector.
    $both[]         An array used to return $trans and $rot to the
                    selfAssembly expression.
```




```
    */
    vector $trans, $rot, $both[];

    /*
    $x              Stores the x-component of the $trans and $rot
                    vectors.
    $y              Same as above for the y-component.
    $z              Same as above for the z-component.
    */
    float $x, $y, $z;

    /***** TRANSLATIONAL DIFFUSION *****/

    $x = `gauss $stdDevTrans`;
    $y = `gauss $stdDevTrans`;
    $z = `gauss $stdDevTrans`;
    $trans = <<$x, $y, $z>>;

    /***** ROTATIONAL DIFFUSION *****/

    $x = `gauss $stdDevRot`;
    $y = `gauss $stdDevRot`;
    $z = `gauss $stdDevRot`;
    $rot = <<$x, $y, $z>>;

    // Store both vectors $both, then return $both to the
      selfAssembly expression.
    $both = {$trans, $rot};
    return $both;

} // End diffuse.
```

Save this procedure in text file in your Maya Scripts directory under the following name:

diffuse.mel

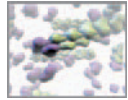
The collide() procedure

This procedure returns a single vector `$collide` which represents the sum total steps taken by the current G-actin in response to collisions with neighboring monomers and the filament. As we stated earlier, the purpose here is not to emulate the true dynamics of intermolecular collisions but rather to embody their net effects.

Moreover, while there exist more sophisticated collision detection and avoidance algorithms, the approach used here is intuitive to grasp, straightforward to implement, and runs quickly for the present model. Nonetheless it has shortcomings which will become apparent when you study the simulation carefully. In particular, it assesses collisions for each monomer only once per time step. As a result, the net response of a monomer to its current collisions can at times cause it to intersect another model with which it was not previously in contact. If you're interested, we encourage you to explore additional collision avoidance strategies and compare their merits and limitations.

```
/* Description:
This procedure detects collisions and moves monomers accordingly.
*/

global proc vector collide(string $gName) {
```

```
/***** DECLARE THE VARIABLES *****/

global string $gNames[];
global float $Rg;
string $name;

/*
$otherTrans     The position of a G-actin being considered for
                collision with the current G-actin.
$separation     The vector separating two G-actins.
$collide        The vector used to avoid a collision between two
                G-actins.
*/
vector $otherTrans, $separation, $collide, $gTrans;

/*
$dist           The scalar magnitude of $separation.
$contactRange   The range within which molecules are considered
                in contact.
$collideScale   Used to scale the unit collision vector.
*/
float $dist, $contactRange, $collideScale;

/***** INITIALIZE THE VARIABLES *****/

$gTrans = `getAttr ($gName + ".translate")`;
$collide = <<0, 0, 0>>;
$dist = 0;
$contactRange = $Rg*2.0;
$collideScale = $Rg/2.0;

/***** CHECK FOR COLLISIONS *****/

// Check the proximity of $gName to every other G-actin.
for ($name in $gNames) {

    if ($name != $gName) { // Test all but the current G-actin.

        // Get the other cell's translate value.
        $otherTrans = `getAttr ($name + ".translate")`;

        $separation = $gTrans - $otherTrans;
        $dist = mag($separation);


        // Determine if $dist is less than the $contactRange
          value.
        if ($dist < $contactRange) { // $gName is in contact
          with $name.

            // The unit separation vector.
            vector $tmp = unit($separation) * $collideScale;

            // The return value will be the sum total of
              collision steps.
            $collide = $collide + $tmp;
```




```
                // Move $name away from $gName.
                $tmp = $otherTrans- <<($tmp.x), ($tmp.y), ($tmp.z)>>;
                setAttr ($name + ".translate") ($tmp.x) ($tmp.y)
                  ($tmp.z);
            }
        }
    } // End for ($name in $gNames).
```

closestPointOnSurface (cpos for short) is an example of an **underworld node**, a type of DG node associated with the parameter space of NURBS objects. Underworld nodes that are used to query points on curves and surfaces (such as cpos) cannot be grouped or parented, and therefore must be named uniquely. Other examples of underworld nodes include:

pointOnCurve

pointOnCurveInfo

pointOnSurface

pointOnSurfaceInfo

You'll get experience with a few of these later in this book.

To detect collisions with the filament model, you'll use the closestPointOnSurface node we presented on page 400. This node was made and connected to the F-actin NURBS cylinder model in the treadmilling.ma scene file on the CD-ROM.

```
    // Check for a collision with the filament.
    $cposName = "F_0_cpos";

    setAttr ($cposName+".inPosition") -type double3 ($gTrans.x)
      $gTrans.y) ($gTrans.z);
    $otherTrans = `getAttr ($cposName + ".position")`;

    $separation = $gTrans - $otherTrans;
    $dist = mag($separation);

    if ($dist < $Rg) { // $gName is in contact with the filament.

        vector $tmp = unit($separation) * $collideScale * 2;
        $tmp = <<($tmp.x), ($tmp.y), ($tmp.z)>>;

        // Add $tmp to the total collision vector.
        $collide = $collide + $tmp;

    }
    // Return the collision avoidance vector to the selfAssembly
      expression.
    return $collide.

} // End procedure.
```

Save this procedure in a text file in your Maya Scripts directory under the following name:

collide.mel

The dissociate() procedure

This is the fifth and final procedure for the project. You can think of it essentially as the reverse of associate(), with the exception that two reactions are under consideration, not just one. The procedure arguments are the name and state of the subunit being considered for reaction. You'll see some print statements as well. These provide helpful information about the outcome of the probability test.

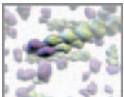
```
/* Description:
This procedure dissociates G-actin molecules from F-actin
filaments.
*/

global proc dissociate(string $subunit, int $state) {
```

```
/***** DECLARE THE VARIABLES *****/

global float $subTrans, $gRot, $Rf;
global float $mADPpi_RevProb, $mADP_RevProb;
global string $fName, $tracer, $gfGroup, $pName, $mName;
global int $totalOff, $fadeSteps;
float $rnd, $plusTy, $minusTy, $plusRy, $minusRy;

/*
$shaderGroup    Used to create the faderShader group node.
*/
string $shaderGroup[], $gName, $gfGroup, $shaderName[];

/*
$react          Set to 1 if a reaction occurs, 0 if not.
*/
int $react;

/***** TEST THE REACTION PROBABILITY *****/

$rnd = `rand 1.0`;
print ("\nInside dissociate(), STATE: " + $state + "\n");

if ($state == 3) { // An ADP subunit.
    print ("$mADP_RevProb = " + $mADP_RevProb + ", $rnd =
      " + $rnd + "\n");

    if ($rnd < $mADP_RevProb) {
        $react = 1;
        print $subunit;
        print (", an ADP subunit will dissociate from the
          filament.\n");
    }
}
else { // An ADP-Pi subunit.
    print ("$mADPpi_RevProb = " + $mADPpi_RevProb + ", $rnd =
           " + $rnd + "\n");

    if ($rnd < $mADPpi_RevProb) {
        $react = 1;
        print $subunit;
        print (", an ADP•Pi subunit will dissociate from the
          Minus-end of the filament.\n");
    }
}
```

The `$react` variable here serves a similar purpose to `$bound` in the `associate()` procedure; it stores the outcome of the reaction probability test: 1 if successful, 0 if not. In the latter case the following code will be skipped.

```
/***** UPDATE THE MODELS *****/

if ($react == 1) { // A reaction occurred at the Minus-end.
```

The following steps are similar to but achieve the opposite results to those outlined in Figure 15.13. The first statement below un-parents the subunit from F-actin and moves it to a random location just below the filament minus end. After that, you'll reset the various F-actin model elements and update its custom attributes.




```
// Move G-actin and parent it to world space.
move -r -os ($subTrans/2*rand(-1,1)) (-$subTrans * 1.5)
  ($subTrans/2*rand(-1,1)) $subunit;
parent -world $subunit;

// Rotate the Minus end locator into position for the next
  bind.
$minusRy = `getAttr ($mName + ".ry")`;
setAttr ($mName + ".ry") ($minusRy + $gRot);

/* Shift gfGroup by half a monomer height within F to
account for the subunit loss. */
move -r -os -wd 0 (-$subTrans/2) 0 $gfGroup;
// Move the filament by half a subunit spacing.
move -r -os -wd 0 ($subTrans/2) 0 $fName;

// Query the local Plus and Minus end coordinates.
$plusTy = `getAttr ($pName + ".ty")`;
$minusTy = `getAttr ($mName + ".ty")`;

// Reposition the Plus and Minus ends.
setAttr ($pName + ".ty") ($plusTy - $subTrans/2);
setAttr ($mName + ".ty") ($minusTy + $subTrans/2);

// Update the F-actin's .subunits and .heightRatio attributes.
string $tmpStr = $fName + ".subunits";
int $tmp = `getAttr $tmpStr`;
setAttr $tmpStr ($tmp–1);
setAttr ($fName + ".heightRatio") (((float) 6.0 +
  $subTrans * ($tmp-2))/$Rf);
```

Next, you'll set the state attribute equal to `$fadeSteps` × −1. This marks the monomer for consideration by the `faderShader()` procedure. Next, you'll create the `faderShader` and assign it to the monomer. `faderShader` is made by duplicating of `ADP_shaderSG`. When the `–ic` (short for `inputConnections`) flag is used, the upstream shader node is duplicated as well. Duplicating an existing shader rather than creating one from scratch saves you the hassle of setting attribute values for the new shader—the attributes are duplicated along with the node.

```
// Update the G-actin's .state attribute.
setAttr ($subunit+".state") (-$fadeSteps);

// Create a fader_shader.
$shaderGroup = `duplicate -un -ic -name
                ("faderShader"+$subunit+"SG") ADP_shaderSG`;
```

Naming the duplicated shading group node does not give a corresponding unique name to the shader node (Maya simply adds an integer to the end of the original shader name). Let's name the new shader distinctly as `faderShader`—so you can pick it out easily in the Hypergraph. To rename the shader you must first get its current name. Do this using the listConnections command as follows:

```
// Rename the upstream shader node.
$shaderName = listConnections ($shaderGroup[0] +
  ".surfaceShader");
rename $shaderName[0] ("faderShader" + $subunit);
```

```
        // Assign the shader to the newly dissociated monomer.
        sets -e -forceElement $shaderGroup[0] $subunit;
```

Finally, check if this subunit is the flux tracer. If it is, clear the $tracer variable so that a new tracer can be assigned when the next plus end association reaction occurs. Like `$totalOn`, `$totalOff` counts reactions so you can check your treadmilling rate at the end of a simulation run.

```
        // If this subunit was the firstBind subunit, reset $tracer.
        if ($tracer == $subunit) $tracer = "";

        // Increment the "off" reaction counter.
        $totalOff++;

    } // End if ($react == 1).

    else {
        print $subunit;
        print "will not dissociate.\n";
    }

} // End procedure.
```

Save this procedure in text file in your Maya Scripts directory under the following name:

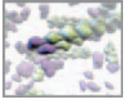
dissociate.mel

That's it! Your code is complete. Make sure you saved each procedure in your Maya's Scripts directory with appropriate names. Now it's time to get your model running!

Results: Running your simulation

Prepare your scene file

1. **Start Maya.**
2. **Choose Window → Settings/Preferences → Preferences.**
3. **Choose Categories → Settings and make the following settings:**
   Under Working Units → Linear: centimeter.
   → Angular: degrees.
   → Time: NTSC.
4. **Choose Categories → Timeline and make the following settings:**
   Under Timeline → Playback Start: 1.
   → Playback End: 900.
   → Time, select NTSC.



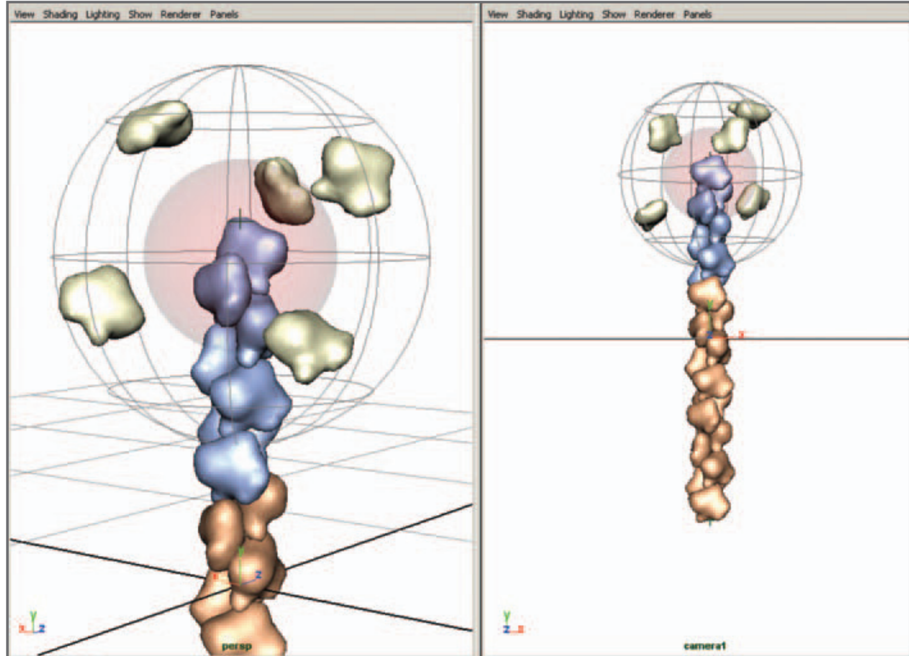


FIGURE 15.15

A two-panel view of the scene. The persp camera on the left is used to move around within your scene. camera1 on the right travels vertically with the actin filament as it treadmills.

Under Playback → Looping: once.

→ Playback Speed: Play every frame.

→ Playback by 1.

5. **Press the Save button to set your preferences.**

Open the `treadmilling.ma` scene file included on the book's CD-ROM:

1. **Copy the scene file from the CD-ROM to your Maya Scenes directory**

   15_Self_Assembly/scenes/treadmilling.ma

2. **In Maya, choose File → Open Scene**
3. **Browse to your Maya Scenes directory and choose treadmilling.ma.**

Take a moment to inspect the scene. Explore the hierarchy relationships in the Outliner and check out the shaders in the Hypergraph. Notice that we've created a camera for you called camera1. Its attributes are locked—to prevent accidental movement—and its translateY attribute is connected to the F-actin model's translateY attribute. Next, set up a panel layout that uses both camera1 and the default persp camera (Figure 15.15). Use persp to move around your scene and camera1 to capture a consisted view of the model's behavior each time you run the simulation.

4. **Choose Window → View Arrangement → Two Panes Side by Side**
5. **In the left-hand view panel choose Panels → Perspective → persp**
6. **In the right-hand view panel choose Panels → Perspective → camera1**

Your scene is now ready for the custom expressions and procedures.

Load the script files

Create the expressions

First you'll create the reset and selfAssembly expressions. If you didn't build the expression scripts earlier in this chapter, copy them from the CD-ROM to your Maya Scripts directory:

15_Self_Assembly/MEL/reset.txt

/selfAssembly.txt

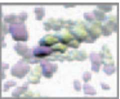
1. **Open reset.txt (either the file you created or the one on the CD-ROM) in the text editor of your choice.**
2. **Ensure that the text editor is set to not use typographer's quotation marks.**
3. **Select and copy the entire script to the clipboard.**
4. **In Maya, enter ExpressionEditor in the Command Line to launch the Expression Editor, or select it from the menu Windows → Animation Editors → Expression Editor.**
5. **Press the New Expression button.**
6. **LMB + click in the Expression text field.**
7. **Press Ctrl + V to paste your expression into the text field.**
8. **Press the Create button at the bottom of the Expression Editor.**
9. **In the Expression Name field, replace the default name with reset and press Enter.**
10. **Repeat steps 5 through 9 for the selfAssembly expression, but name it selfAssembly in the Expression Name field.**

If Maya generates one or more errors when you press the Create button, you will need to debug the expression: open the Script Editor to view the specific error messages and to read the line number(s) that generated the error(s). If your text editor can display line numbers, use this feature to cross-reference the error messages to the offending lines in your expression. If you are unable to resolve the errors, you can compare your script to the appropriate file (**reset.txt** or **selfAssembly.txt**) included on the CD-ROM.

11. **Press Ctrl + S to save your scene with the expressions in it.**

Prepare the procedures

In the previous chapter, you loaded the cpk.mel procedure by "sourcing" it through the Script Editor. In this project you'll let Maya search for and load the procedures automatically when they're called by the selfAssembly expression. In order for this method to work, it's essential that your five procedures are each contained within a separate file and saved in your default Maya Scripts directory. For example, the associate() procedure must reside within a file named associate.mel. You can query



the name of your Maya Scripts directory by entering the following statement in the Script Editor:

```
internalVar -userScriptDir;
```

You may recall from *Chapter 12* that Maya only creates the contents list (file names) of its search path at startup. If you add files to the Scripts directory when Maya is running, you will have to refresh the search path contents, using the rehash command, in order to have access to those files and their contents.

Refresh the search path contents. In the Script Editor, enter:

```
rehash;
```

Running and debugging your simulation

When you're ready to run the simulation, press the Play button in Maya's timeline controls.

Errors

Unless you've already tested your procedures for syntax errors and corrected them, chances are you'll get error messages when you go to run the simulation for the first time. If this happens, we recommend sourcing one procedure script at a time through the Script Editor and fixing the bugs as they're flagged by Maya. After you've debugged syntax errors you may discover that your expressions and procedures contain runtime errors—errors that appear only upon execution—that didn't appear when you loaded or sourced the scripts. You can tackle these by reading the error description in the Script Editor and then tracking down the sources one at a time. Alternately you can compare your scripts line-by-line with those we've included on the CD-ROM and search for discrepancies:

15_Self_Assembly/MEL/reset.txt

/selfAssembly.txt

/associate().mel

/collide().mel

/diffuse().mel

/dissociate().mel

/faderShader().mel

One common runtime error occurs when Maya is unable to locate a procedure that has been called. If you get an error such as

```
// Error: line 62: Cannot find procedure "associate". //
```

It means that Maya is unable to locate your associate().mel file—or that you misnamed the procedure either in the MEL script file itself or in the procedure call

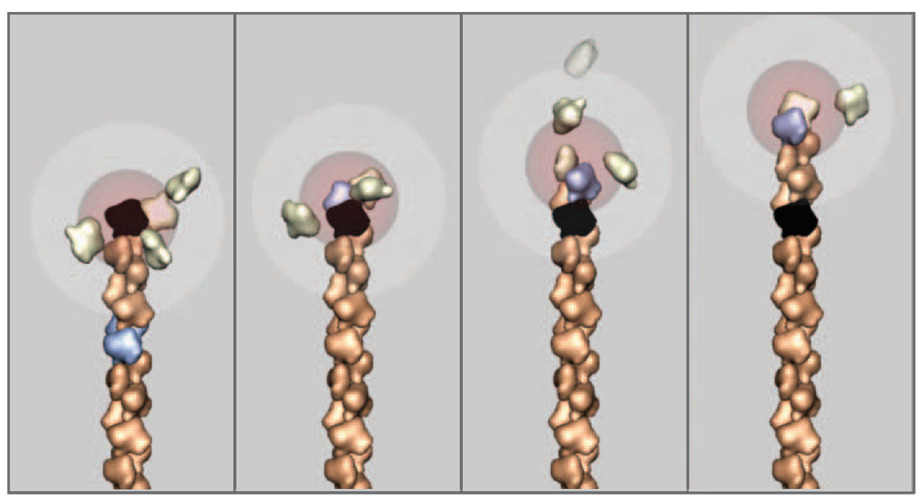


FIGURE 15.16

Four frames from a simulation run (approxmiately 10 frames apart).

within selfAssembly. It may also be that you added the file containing the procedure to your Scripts directory while Maya was already running. In this case, executing the rehash command refreshes Maya's search path contents.

When your scripts are error free, you'll be able to play and stop the simulation as you like. Each time you rewind to frame 1, your model should reset to its original state.

Summary

Figure 15.16 shows four frames from a simulation we ran using the current model. After each run, we entered the following code in the Script Editor to see how many on and off reactions occurred:

```
print ("$totalOn: " + $totalOn + "\n$totalOff: " + $totalOff + "\n");
```

The average results were:

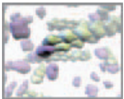
```
$totalOn: 107
$totalOff: 106
```

which are well within range of our target number:

Number of reactions = $P^+ \times$ Frames = (1/9) (900) = 100

We encourage you to vary the model's parameters and observe their effects on treadmilling behavior. There are many ways you could take the model further, including the incorporation of plus end dissociation and minus end association. Figure 15.17 shows a still from one of our multi-filament simulations in which both the G- and F-actin diffuse through space.

The regulated self-assembly of actin molecules you've emulated in this project is central to cell locomotion—the growth and shrinkage of actin filaments pushing and retracting the cell membrane extensions called pseudopodia. In the next chapter you'll take a natural next step and explore some of the principles of cell locomotion via Maya.



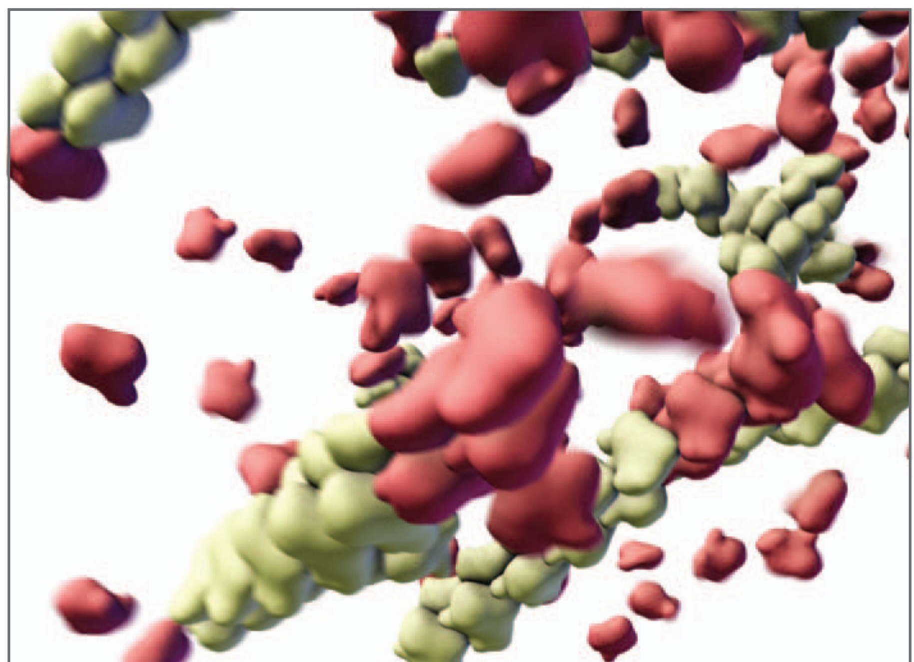


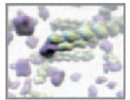
FIGURE 15.17

Still from one of our multi-filament actin assembly simulations. The filaments diffuse along with the G-actin molecules. Free G-actin and recently bound subunits are red. This image was rendered using Maya's 3D motion blur to give the impression of movement in a still image.

References

1. Alberts B, Johnson A, Lewis J, Raff M, Roberts K, Walter P: *Molecular Biology of the Cell*, 5th edn. Garland Science, Boca Raton, FL, 2007.
2. Otterbein LR, Graceffa P, Dominguez R: The crystal structure of uncomplexed actin in the ADP state. *Science* 293: 708–711, 2001.
3. Lorenz M, Popp D, Holmes KC: Refinement of the F-actin model against X-ray fiber diffraction data by the use of a directed mutation algorithm. *Journal of Molecular Biology* 234: 826–836, 1993.
4. Plimpton S, Slepoy A: ChemCell: A particle-based model of protein chemistry and diffusion in microbial cells, Sandia National Laboratories Report Number SAND2003-4509 (technical report online): infoserve.sandia.gov/sand_doc/2003/034509.pdf, accessed August 3, 2007.
5. Pollard TD: Rate constants for the reactions of ATP- and ADP-actin with the ends of actin filaments. *Journal of Cell Biology* 103: 2747–2754, 1986.
6. Romero S, Didry D, Larquet E, Boisset N, Pantaloni D, Carlier MF: How ATP hydrolysis controls filament assembly from profilin–actin: Implication for formin processivity. *Journal of Biological Chemistry* 282: 8435–8445, 2007.

16 Modeling a mobile cell



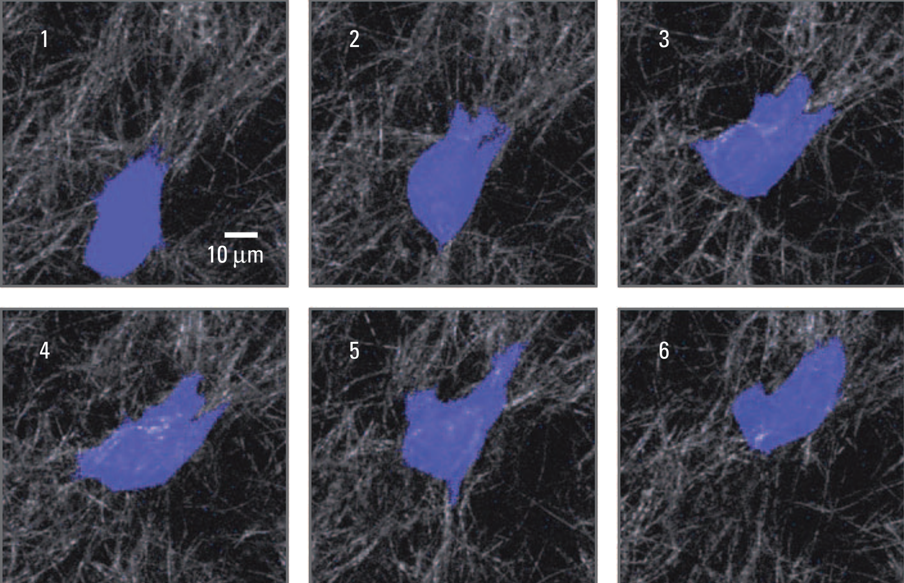


FIGURE 16.01

Through innovations in experimental techniques and imaging technology, in vitro and in vivo cell studies will increasingly enter the realm of 3D. Shown here is a migrating cancer cell captured on video within a 3D scaffold of collagen fibers.[1]

Courtesy of Katarina Wolf and Peter Friedl, University of Würzburg, Germany.

Introduction

In previous chapters we have focused on the atomic, nano, molecular, macromolecular, and multimer levels of biological organization, using both hemoglobin and actin as working examples. Among other roles in cell structure and function, dynamic cycles of actin polymerization and depolymerization, and of actin filament bending, are now widely accepted as the engine that drives cell locomotion. This self-propelled translocation of a whole cell is implicated in both normal physiological and disease processes. Notable examples include fertilization, embryonic development, wound healing, and the spread of cancer cells (Figure 16.01). There are several primary mechanisms that drive cell locomotion: spinning flagella, which propel human sperm cells (a 3D computer model of a sperm cell is shown in Figure 16.02) and many species of bacteria; beating cilia, also responsible for propulsion in many types of single-cell organism; and crawling, which is the mechanism commonly employed by mobile cells in animals.

In this chapter we will introduce you to a method we created for the purpose of procedurally animating the crawling movement of a cell. You will build the cell behavior simulator using a very powerful Maya construct: character rigging, by which 3D CGI animation models are induced to change their spatial configuration over time. In conventional character rigging (think of the animated behavior of your favorite 3D CGI character) joint deformers are bound to the deformable surface mesh before animation takes place; the relationship between the deformers and the surface doesn't change once the animation begins. In contrast, our rig is created *during* the animation—on the fly—not before it. The deformers are continually destroyed and recreated by the MEL script, in new positions relative to the mesh in order to deform the cell in any required direction; the cell deformations are not limited to the initial placement of the deformers.

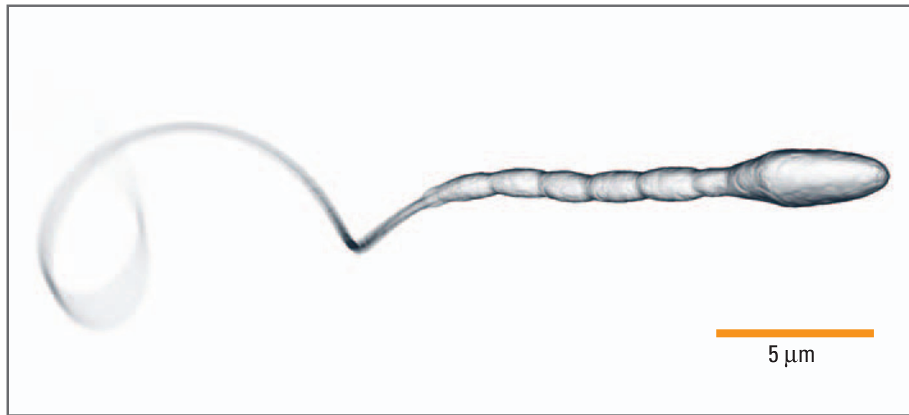


FIGURE 16.02

A 3D computer model of a human sperm cell.

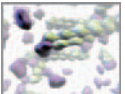
Your model will consist of a single cell crawling over a surface, in a combination of random and directed movement. The surface might be a plane of tissue in a developing embryo or part of a blood vessel into which cancer cells will escape and invade distant organs. The plane might be the glass slide in a microscope through which a cell biologist is tracking and measuring migration paths. All are interesting and important cases of cell motion. In the next chapter we will look at what it takes to begin modeling the even more complex world of the 3D tissue environments in which cells move when they are not zeroing in on surface motion. Presently, your cell is free to move about the plane without needing to work around collisions with other cells or objects. This is the simplest scenario that a cell could face and makes an excellent point from which to begin simulating cell locomotion. In the subsequent chapters we will explore collision detection strategies that allow us to locomote cells in response to, not in spite of, their environment.

In this chapter you will first create a scene file which includes the cell geometry. Then you will build a Maya expression to animate the cell during playback of the scene. If you wish to dive right into the model, you can skip ahead to the *Model Definition* on page 449. The next section provides background on the fascinating world of cell locomotion and why it's suited to computer modeling. By the end of this chapter you will have created a prototype cell locomotion model. Along the way you'll learn how to rig and deform a character model, both with the Maya UI tools, and procedurally using MEL commands. You can then apply these tools and techniques to your own cell behavior modeling tasks.

Problem overview

Crawling in context

In this section we are going to take a closer look at cell crawling, which is of central importance in the cell biology of both health and disease. To see why, think of your body as a society of about 100 trillion cells, all of which have descended from the fertilized ovum at the beginning of your life. Like members of any cooperative society, there is a complex division of labor by which those cells are organized to perform specific tasks: nerve cells transmit and process information, gut cells transport nutrients from digested food to the blood stream, kidney cells filter waste products from the circulation, and so on. Most of these cellular specialists are immobile: their lives



are spent at or near to the spot where cell division splits them from their progenitor cells. The developing embryo, from which those more sessile descendents finally arise, attains its form as successive generations of progenitor cells divide and move, by cell crawling, into the 3D configurations that become working brains, kidneys, and the other familiar organs and tissues of our bodies. While many of the cell types comprising the developed tissues have minimal inclination to move around, several groups of highly mobile specialists are crucial to our health: wriggling sperm (Figure 16.02) seek unfertilized ova; red blood cells, passively adrift in the flowing blood, carry life-giving oxygen to the tissues and carry off poisonous carbon dioxide metabolic waste; cells of the immune system crawl through the tissues, patrolling for invading viruses and bacteria; traction specialists called fibroblasts activate in wounds and contract, helping pull the regenerating tissue shut. Crawling cells can, by disruptions in their motility control, also become deadly threats as cancerous mutations change otherwise sessile specialists into determined movers unable to cease an endless cycle of division and invasion: the growing tumor edging into healthy parts of the body.

Crawls and walks

The project you'll undertake in this chapter is to create an interesting initial model of a cell crawling behavior. The science and the MEL you'll explore in carrying out this project will draw on a remarkable discovery: despite the impressive complexity of the biochemistry behind cell crawling, and its seemingly endless web of chemical detail, the resulting behavior the crawling of the cell has simple, beautiful mathematical properties. When biologists watch through microscopes and trace the pattern of crawling cell movement as demarcated, say, by the position of the cell center or the cell surface, the meanders traced out by cell crawling in two and three dimensions have the traits of random walks!

The details of the biochemistry behind this random walk behavior are intricate and not yet fully worked out. If you find the subject intriguing, this chapter's References will take you beyond the capsule survey we have room for here. And while questions remain, the basics of the cell crawl mechanism are well understood (Figure 16.03): actin filaments like the one you dealt with in the last chapter begin to grow just beneath the cell surface, shifting a balance of chemical and mechanical forces so that the cell surface protrudes, and may ultimately elongate in the direction of the filament-rich protrusion. In the latter stage of the cycle, the cell shifts the bulk of its mass in the direction of that protrusion and prepares to repeat the cycle. Repeated over and over as the cell thrusts protrusions out in numerous directions and follows their advance, the cell slides along in a stochastic meander.

The motion of your model cell certainly will be simpler than observed in real cells. Cell crawling often belongs to the very interesting class of persistent random walks[6–10], which are probabilistic wanders in which the mover carries on rather longer in one direction, once movement begins, than we would expect based on an acquaintance with the Brownian motion random walks of chemical diffusion (Figure 16.04). This chapter's project will not embrace all these mathematical subtleties, but will get you started with a method you can easily refine and extend. In fact you'll be doing some of this in later chapters, when you let the cells escape from two into three dimensions. And while acknowledging the role of actin and other proteins in the chemistry behind the crawl, we'll focus on the resulting behavior itself—the repeating cycles of protrusion, traction, and advance.

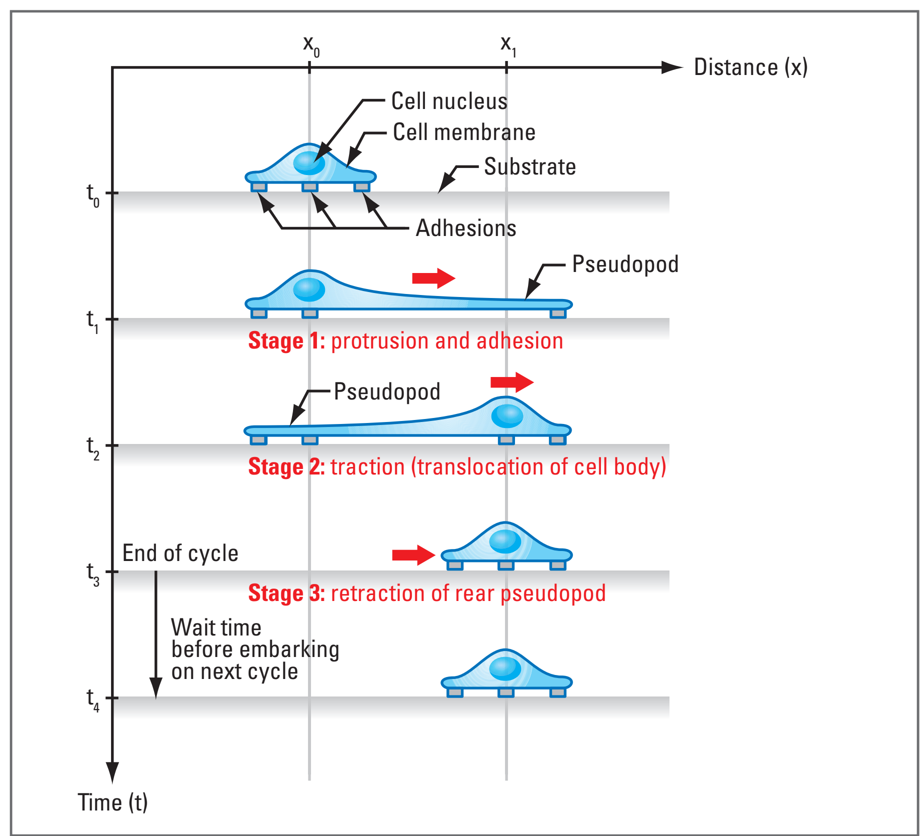
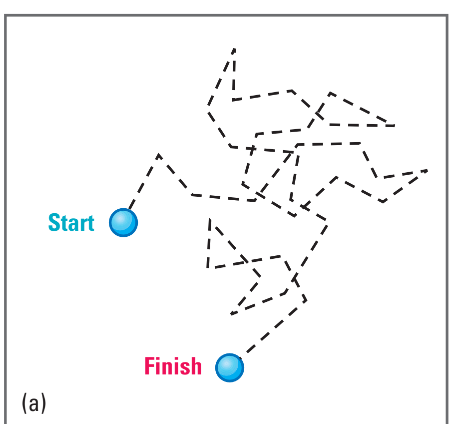


FIGURE 16.03

The stages of cell locomotion on a flat substrate.



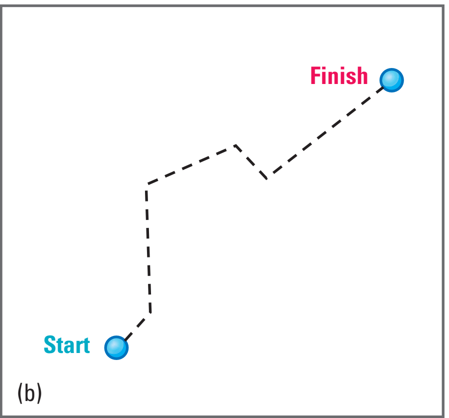


FIGURE 16.04

Random motion can be characterized as (a) Brownian and (b) persistent.

Your model will incorporate an additional element from chemistry that makes them aware of their environment. Sensor molecules in the cellular wanderers let them detect chemical concentration gradients. Detection can trigger changes in the probability characteristics of the cell crawling, causing their random walk to become

biased in spatial orientation. The crawling cell can then navigate along these chemical gradients.

Navigation steps produced by chemically oriented cell crawling are thought to be essential to the normal formation of tissues and organs. They may also be keys to understanding how a few cancer cells can mobilize into an effective invading force, tracking outside their place of origin and mounting a deadly threat to the entire body. Your model will include the cell's navigation response to chemical signals.

Fast and slow movers

Different body cell types adapt different strategies for crawling locomotion based on their makeup and that of their surrounding tissues. For example, fibroblasts (connective tissue cells) and some tumor cells abundantly express proteins of the integrin family which enable the cells to make strong adhesions with the **extracellular matrix** (**ECM**), the "glue" holding cells together in ordered patterns of healthy tissue. These cells are relatively large and slow moving (≈0.01 μm/s), and rely on constant reorganization of the actin cytoskeleton for locomotion. Endothelial cells are another cell type that rely on integrin-mediated locomotion to do their job during **angiogenesis**, the growth of new capillaries. In contrast, lymphocytes and neutrophils, both white blood cell types, are smaller and express much lower integrin levels. Their locomotion is characterized by shorter-lived integrin-independent contacts with the ECM and the ability to adapt cell shape to pre-formed cytoskeleton architecture rather than having to constantly reorganize it. As a result, lymphocytes can move relatively quickly (≈0.1 μm/s) compared with their larger, integrin-dependent cousins like fibroblasts.

Protrusion nomenclature

For both fast and slow movers, those membrane protrusions observed in the first stage of the crawl cycle are often called **pseudopodia** and adhere to the substrate via adhesion molecules such as integrins. On a flat substrate, cells tend to send out flat protrusions. These are often called *lamellopodia*, in place of pseudopodia, in reference to their flattened shape. Furthermore, in the scientific literature on lymphocyte cell migration, rearward membrane projections are sometimes called *uropodia* rather than pseudopodia. In this book we'll simply use of the words pseudopod and pseudopodia to refer to motility-related projections of the cell surface involved in movement behavior. In the next stage the cell body translocates toward the front-end adhesion: that process termed **traction**.

As the body moves, it leaves behind rearward protrusions, also called pseudopodia. In the final stage, de-adhesion and **retraction**, the rearward adhesions are released and the rear pseudopodia retract to rejoin the cell body. All three motility processes—**protrusion, traction**, and **retraction**—involve the actin cytoskeleton. There is arguably a fourth stage of locomotion which involves no locomotion at all! This is when the cell has no pseudopods extended yet.

Navigation nomenclature

We saw that cells move in response to chemical signals. When this movement is nondirectional, or random, the process is called **chemokinesis**. When the motion is directional, we call the process **chemotaxis**. Chemotaxis is the mechanism employed during

embryonic development as cells migrate to different regions and become specialized for specific tissues. Similarly, chemotaxis is believed to be largely responsible for the migration of fibroblast cells into a wound environment in order to begin healing. As well, cell-cell signaling among cancer cells creates a chemotactic environment for metastasis. Your cell model in this chapter will undergo random motility on a homogeneous substrate, but with a chemotactic stimulus to provide a directional bias to its movement.

Cells don't need the stimulus of chemical gradients to set them in motion. Cells also move in response to contact with a substrate, a process called **haptotaxis**. Haptotaxis can be a random process (in which cells wander aimlessly), or a directed one, depending on the organization of the substrate. In the absence of external chemical stimuli, cells on a homogeneous substrate such as a smooth coverslip will tend to wander aimlessly. Conversely, a non-homogeneous substrate—on which there is a directional pattern to the surface-bound chemicals which they can detect—cells can be influenced to migrate in specific directions.

A **ligand** is any molecule that binds to another. In biochemistry the term usually refers to a small molecule that binds to a receptor or other kind of protein. In cell migration it is the binding of membrane receptor molecules to ECM ligands—small side-chains of large ECM molecules like collagen—that enable a cell to generate a traction force by contracting its cytoskeleton.

Model definition

The cell model

Your goal is to make a model of a crawling cell that migrates on a flat substrate via random motility and under the influence of a chemotactic signal, or **chemoattractant**. The cell is inclined to move from a lower to a higher concentration of chemoattractants, or signaling molecule. To account for the probability effects involved in cell locomotion, you will build randomness into the speed and direction calculations for your cell. Therefore as your cell crawls up the chemoattractant gradient, it won't necessarily do so in a straight line.

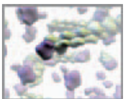
Like a living cell beginning its migration on a coverslip, your cell will start off as a 10 μm spheroid that has been flattened around the edges. You could call it a lymphocyte, but it could just as easily be another type of motile cell that crawls using the three-stage mechanism shown in Figure 16.03. For simplicity, the cell will produce a single pseudopod at the start of each locomotion cycle and retract a single one at end of the cycle. We will introduce the technique shortly for setting up your cell to deform. This will be the key Maya technique you'll learn and apply in this chapter. Below is a list of parameters you'll build into the model:

1. **Leading pseudopod protrusion rate.**
2. **Traction (cell center translocation) rate.**
3. **Trailing pseudopod retraction rate.**
4. **Wait time—the time spent sitting still between crawl cycles.**

From these four parameters, a fifth will emerge:

5. **Linear migration speed of the whole cell.**

Without adhering to the substrate, a cell cannot generate the traction required to move. We can represent cell-substrate adhesions in Maya by fixing the location of different portions of the cell surface—by fixing the joints—at appropriate times during



the crawl cycle. Finally, to make the cell center clearly visible as it moves about the scene, you will couple a smaller sphere—the nucleus—to the cell body.

Cell behavior

Your cell will start its journey at the Maya world origin, decide on a direction and incremental distance to move, and then execute a crawl cycle that encompasses the different stages shown in Figure 16.03. A living, motile cell has no definite front and back, or left and right. When it changes direction, it does so by sending out a pseudopod in the new direction, not by turning its body around. It has the ability to extend processes in *any* direction meaning that any location on the cell surface may become the new cell "front". This is an important characteristic of living cells and therefore one we want to build into the model.

Furthermore, to make the model flexible, you'll want the ability to easily adjust the rates of the different motility stages: protrusion; traction, and retraction. This will allow you to tailor the model with experimental data on unique motility characteristics for different cell types and locomotion scenarios.

For the determination of random motility, the calculation of direction, which is denoted by the angle alpha (α), and the distance to be traveled, which we'll call reach (for the reach of a pseudopod) should vary randomly from crawl cycle to crawl cycle. The specific numbers involved in this random variability are not important at this stage; they can be tailored in later, more advanced stages of the model—if you wish to take it further—in order to test different hypotheses about cell locomotion.

The chemotactic signal

You're including in the model a chemotactic influence, or **chemoattractant** gradient, to bias the random motility of the cell. This can be characterized as an angle, theta (θ), to represent the gradient orientation (see Figure 16.05), and a magnitude, c, to represent its strength. These two parameters will be used to bias the cell's direction by altering its motility angle, alpha.

The substrate

Using a flat (horizontal) substrate allows you to limit the complexity of the model to two dimensions in Maya: X and Z. Such a substrate can be modeled implicitly by limiting all Y-translation values to zero throughout the locomotion simulation. If you like, you can place a geometric plane object (NURBS or polygon) on the view plane made by the XZ world axes to depict the substrate plane explicitly and receive cast shadows from the cell when you render the scene. Furthermore, as with an in vitro study of locomotion, you will need a point of reference on the substrate from which all distance measurements will be made. The Maya world origin makes for a logical reference point.

The cellular scale

When working with the actin model in the previous two chapters, you set 1 Maya unit equivalent to 1 Å or 0.1 nm. In moving from the scale of individual molecules to that of a whole cell, we make a 1,000-fold leap in scale; whereas one G-actin monomer is approximately 7 nm across, a motile lymphocyte cell, for example, is about 10,000 nm

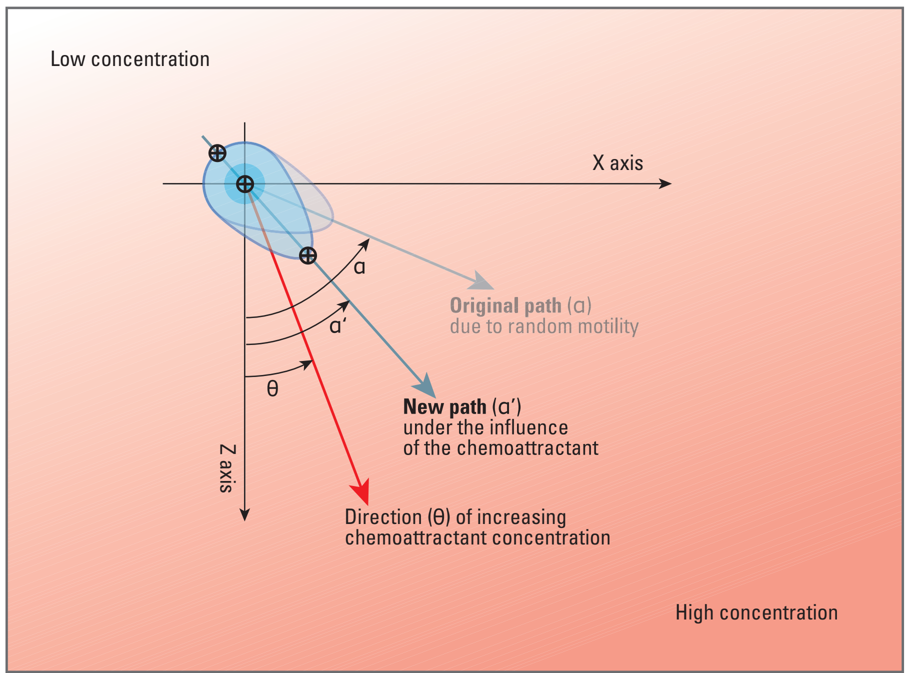


FIGURE 16.05

Chemotaxis occurs in the presence of a gradient of signaling molecules, a chemoattractant or chemorepellant, which is represented here by the color ramp. You will represent such a gradient in Maya using an angle, **theta**, for its orientation and a magnitude, **c**, which will bias the random motility angle, **alpha**, chosen by the cell.

(or 10 µm) in diameter. For simplicity, let's make 1 Maya unit represent 1 µm for this project. As for time scale, we know that a eukaryotic fast mover like a lymphocyte cell migrates at an average speed of 0.1 µm/s. What you call a Maya frame in terms cell migration time will depend on the speed parameters you set for the cell. In other words, once the model is functioning, you can work backwards to determine how many seconds of cell time is represented by 1 frame on the Maya timeline.

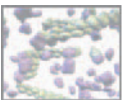
Methods: Generating pseudopods

A principle creative element of your project is the way your cell changes shape, extruding and retracting pseudopodia, as it moves. This cycle of deformations and shape dynamics gives your model enhanced realism as a simulation of real cell activity. It also sets the stage for more advanced applications, such as the MEL code that links the mechanical properties of the cell and the locomotion forces to exact predictions of protrusion size and shape.

But how to coax such smooth biomorphic deformations and extrusions out of Maya geometry models? In this chapter you will learn about and apply a powerful approach to this problem based on the concept of the rig and its atomic constituents, bones and joints.

Animation using joints

Underlying the movements of animated characters—the articulated CG actors you see in films and on television—are what animators refer to as character rigs. A rig is a set of deformers and other tools that an animator uses to control everything from



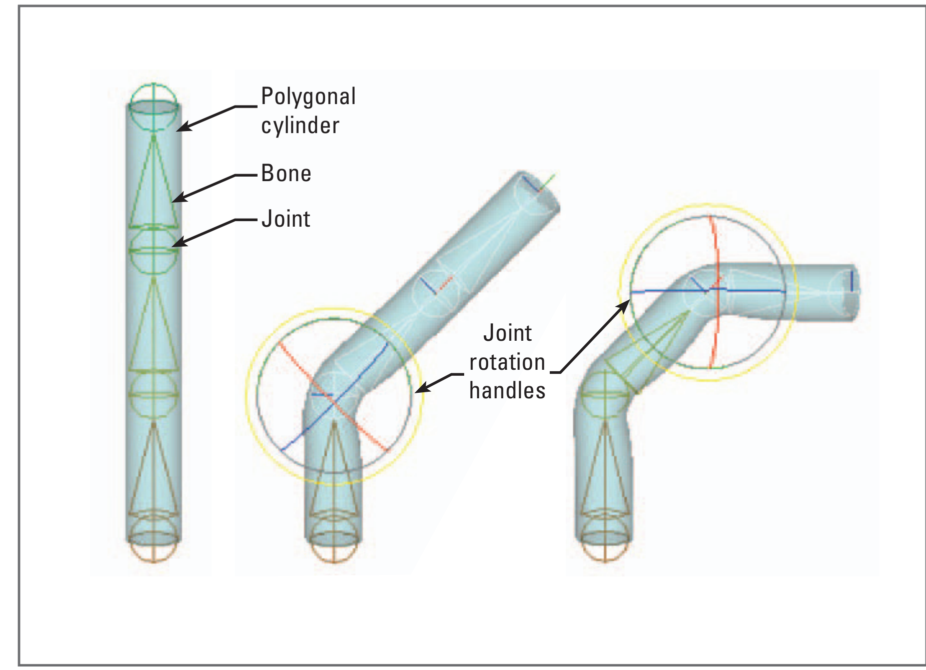


FIGURE 16.06

A skeleton rig applied to a polygonal cylinder. The surface CVs are under the influence of the joints. As the joints rotate, the surface deforms. Bones relate the joints to one another throughout the Scene Hierarchy and constrain the locations of joints in space.

facial expression to gait, and all gestures in between. A common type of character rig involves a skin that is bound to a skeleton (animation curves provide the muscle). The skin is the surface mesh to be animated. The skeleton is comprised of joints which are connected by bones, much like a real animal skeleton. The joints do the actual deforming, while the bones locate the joints relative to one another and constrain their movement in space. When you bind a skin to a skeleton, you are putting the mesh control vertices (CVs) under the influence of the nearby joints. The total influence on all CVs is distributed among all the joints in the skeleton. When it comes time to animate, the animator rotates the joints to position a limb, for example, and the skin deforms accordingly. Figure 16.06 shows a simple skeleton rig applied to a polygon primitive cylinder in Maya; in your hands for a biomedical project, the mesh could just an easily be an arm or a leg, or a chain of amino acids folding to form a protein!

When you build a skeleton the traditional way, joints are related to one another through the Scene Hierarchy via parent/child relationships with bones. Like bones in the human body, a Maya bone is rigid and of a fixed length. While these characteristics are essential to creating a typical character animation rig, they can be a hindrance when animating models that need to change size and shape continually, such as cells in dynamic simulation models.

Your cell will need to alter its rigging "on the fly" to accommodate different pseudopods advancing in different directions. When rigging the cell, therefore, you will take advantage of the fact that joints can be used independently of bones, existing all at the same level in the Scene Hierarchy; in other words, transforming one joint has no influence on the other joints since none of them are related by parent/child relationships.

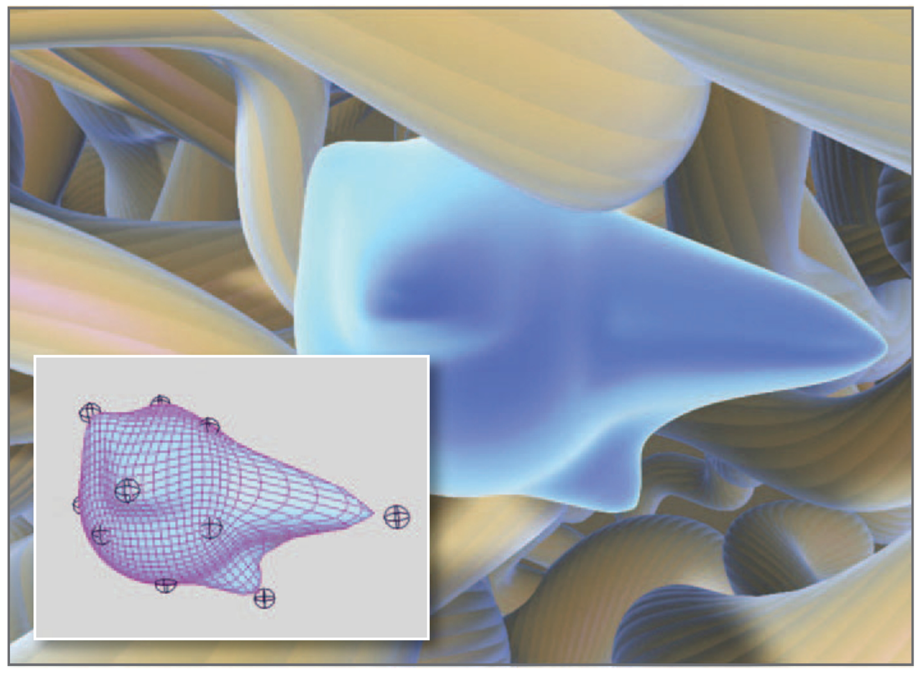


FIGURE 16.07

When joints are used without bones to deform a surface, they can be moved about in space, unconstrained by the hierarchical relationships imposed by bones in a typical skeleton rig. This boneless skeleton is useful for rigging amorphous surfaces like the crawling cell model shown here. The large picture is a still from a Maya simulation of cell motility in a 3D tissue environment. The inset shows the cell model polygonal mesh along with joints used to animate it.

Image courtesy and copyright 2006 Donald Ly, University of Toronto.

What this leaves you, the animator, with is a collection of joints that can be moved independently about the scene, while exerting a distributed influence in their target geometry—in this case a cell. This type of rig (shown in Figure 16.07) is well suited to procedural control of a surface since the deformations are related directly to the translation values of the joints and are therefore predictable and easy to manage. You will see just how useful this simple rig can be for a cell model shortly.

Methods: **Algorithm design**

Figure 16.08 shows the Maya elements comprising your cell locomotion model. Assuming these pieces are in place (which they will be shortly!) let's design an algorithm to make the cell crawl. Following our in silico workflow, let's first lay out the algorithm in flowchart form and then encode the flowchart into a Maya expression. You will find it helpful to state clearly and succinctly the goal of this algorithm:

> Make a polygonal object deform, frame by frame during Maya playback, to resemble a crawling cell. The direction and magnitude of the joint-induced deformations are to be calculated using pseudorandom numbers to emulate the required probabilities, taking into account a chemotactic angle and magnitude.

The flowchart in Figure 16.09 expands this statement into a series of steps describing one complete crawl cycle. Figure 16.10 shows a diagrammatic translation of the flowchart and Table 16.01 lists the nomenclature you'll use to name and refer to the various elements of your model. Now that we have a step-by-step plan to simulate and visualize the crawl, let's build your cell model!

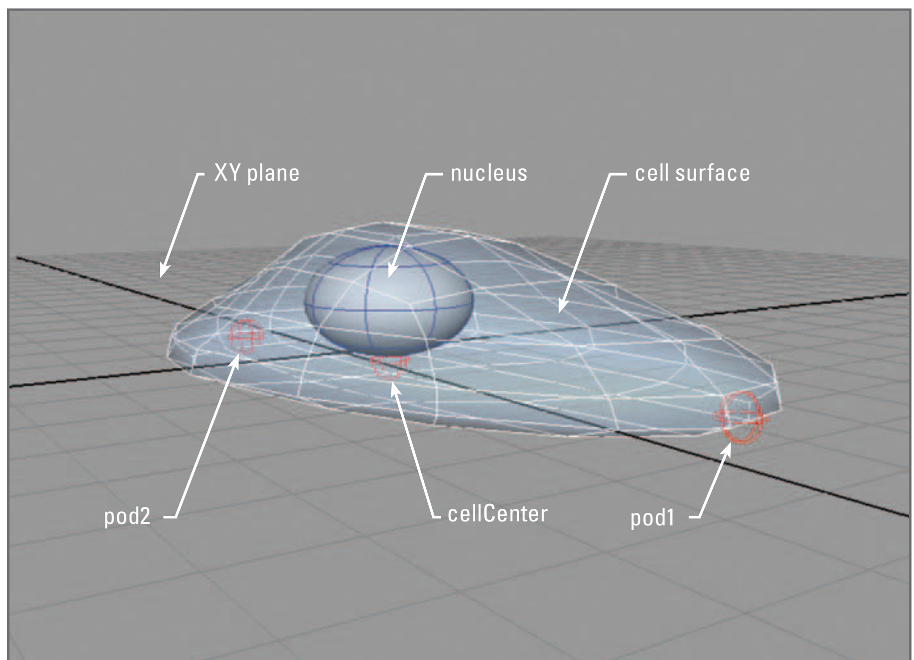


FIGURE 16.08
The crawling cell model in terms of its Maya components: a polygonal mesh for the cell surface, two joints (pod1 and pod2) bound to the mesh. The pod1 and pod2 will draw out the membrane protrusions, or pseudopodia. A third joint represents the cell center. The XZ plane represents a flat substrate for the cell to crawl on.

Methods: A cell locomotion engine

In this section you will set up your Maya scene file, make the cell geometry, and build a Maya expression to animate it. If you wish to begin experimenting with the model right away, you can open the complete Maya scene file for this chapter. It is located on the accompanying CD-ROM:

16_Mobile_Cell/scenes/cellCrawl.ma

Please note that there are two versions of the scene: one tested for Windows and one tested for Max OS X 10.4 (Tiger). If you're using Maya for Mac OS X, refer to the `readMe.txt` file, which is in the same directory as the Maya scene file, for a brief description of the differences between the two scene file versions. If you're building the scene from scratch, you'll find a small modification described at the end of this section that will enable the scene file to be opened in Maya for Mac OS X.

Prepare your scene file

Start Maya and make the following settings. If Maya is already running, save your work and start a new scene file.

1. **Choose Window → Settings/Preferences → Preferences.**
2. **Choose Categories → Settings and make the following settings:**

   Under Working Units → Linear: centimeter.
   → Angular: degrees.
   → Time: NTSC.

Start and Stop on ***press Play***

frame = 1 ?

yes

Reset the cell model

- reset **$i** to 0
- delete existing cell rig elements: **joints**, **skinCluster**, and **bindPose** nodes
- move the **cellCenter** joint and the cell to the origin
- bind the cell to the **cellCenter** joint, creating **skinCluster** and **bindPose** nodes

no

frame > 1 && **$i** == 0 ?

yes

Set up the next crawl cycle

- calculate the random motility direction, **$alpha**, and distance, **$reach**
- add the effect of chemotattractant direction, **$theta**, and magnitude, **$c**
- calculate the location for the pseudopod joints **pod1** and **pod2**, and for the destination of **pod1**
- create and position **pod1** and **pod2**
- add **pod1** and **pod2** to the existing **skinCluster** node
- calculate the step sizes (μm/frame) for each stage of the crawl
- calculate the waiting time between this and the next crawl cycle

no

frame > 1 ?

yes

Increment the crawl cycle

Depending on which crawl stage the cell is in, increment the position of one of the following:

- **pod1**
- **cellCenter**
- **pod2**

no

$i −= 1

frame > 3 && **$i** == 0 ?

yes

Prepare for the next crawl cycle

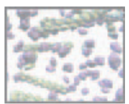
- delete **pod1** and **pod2**
- unbind cell from **skinCluster** node

no

frame += 1

FIGURE 16.09

Flowchart for the cell crawl model. The variable $i is the number of frames left in the current crawl cycle. Because the algorithm runs as an expression, it starts and stops when the Play button is pressed.



Frame = 1

cellCenter joint
Cell membrane
Cell nucleus
X axis
cellRadius
α
γ
Randomly chosen cell direction (α)
$\gamma = \alpha - \theta$
θ
Z axis
Direction of increasing chemoattractant concentration (θ)

Frame > 1 (setup)

pod2 joint
pod1 joint
α
γ
α'
θ
ligandVect (resultant cell motility vector)
ligand position (pseudopod destination)
$\alpha' = \alpha - \gamma \times c$ (resultant cell direction)
where c = chemoattractant strength
(0.1 < c < 1)

STAGE 1 Protrusion

pod1Vect
α'
ligand position
pod1Speed
= rate of protrusion (µm/frame)

STAGE 2 Traction

centerVect
centerSpeed
= rate of translocation (µm/frame)

STAGE 3 Retraction

pod2Vect
pod2Speed
= rate of retraction (µm/frame)

Stage 4 Waiting: pod1 and pod2 are deleted in preparation for the next cycle.

local cell coordinates
next cell direction
α'
next ligandVect
θ

FIGURE 16.10
A diagrammatic version of the cell crawl algorithm.

Name	What it looks like in Maya	Description
cell		The cell body (polygonal mesh). In our rig it gets coupled to the joints below through a **skinCluster** node.
cellCenter		This joint anchors the membrane, proximal to the cell center, to the substrate during leading-end protrusion and rear-end retraction. During the traction stage, this joint moves the bulk of the cell body forward.
pod1		This joint deforms the cell to create the leading pseudopod. After the extension of the pseudopod, this joint adheres it to the substratum during the traction and retraction stages of locomotion. Theoretically, **pod1** provides the anchor against which the traction force is generated.
pod2		This joint anchors the rear of the cell to the substratum, creating the trailing pseudopod as the cell body advances during the traction stage. In the retraction stage, **pod2** returns to the periphery of the cell body, bringing with it the deformed cell surface.
nucleus		The cell nucleus (squashed NURBS sphere).

TABLE 16.01

Nomenclature used in our cell locomotion model.

3. **Choose Categories → Timeline and make the following settings:**

   Under Timeline → **Playback Start: 1.**
   → **Playback End: 1000.**
   → **Time, select NTSC.**

   Under Playback → **Looping: once.**
   → **Playback Speed: Play every frame.**
   → **Playback by 1.**

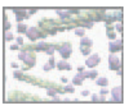
4. **Press Save.**

5. **Select a Four-View of your scene by pressing the button in the Toolbox.**

Because you're dealing with an expression that is evaluated at each frame, it is especially important to set **Playback Speed** to **Play every frame**.

Build the geometric model

Here you'll create a polygon cube and then smooth it to turn it into a sphere. This technique is handy for making spherical surfaces with roughly equal sized polygons. Figure 16.11 compares a Maya primitive polygon sphere with one created by the smoothing a cube. Quite a difference! The CVs are more evenly distributed on the latter, making surface deformations more predictable and easier to control, and also making it easier to map textures onto the surface.



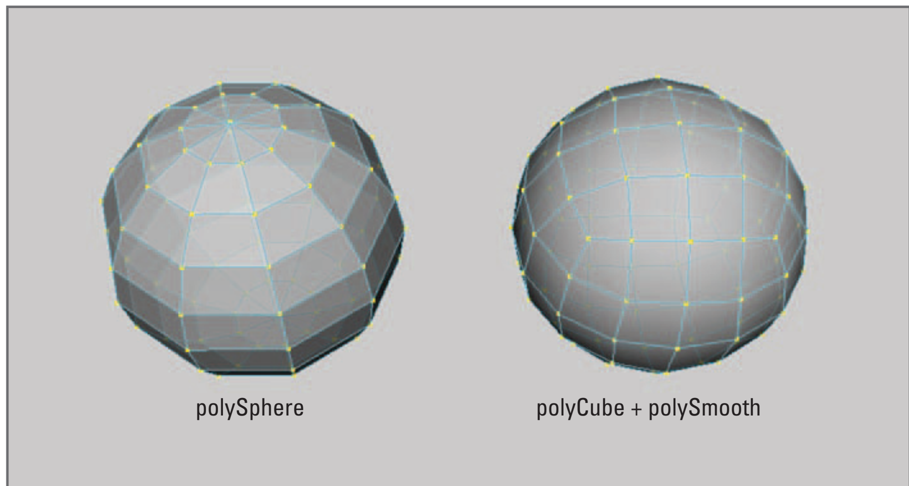


FIGURE 16.11

Two types of polygonal sphere model in Maya. On the left is a primitive sphere created with the **polySphere** command. On the right is a **polyCube** (with width, height, and depth subdivisions all set to 1) with a **polySmooth** node applied. Note that the CVs are more evenly distributed over the surface of the smoothed cube.

Up until this point in this book you have relied on the Maya UI menus to access many of the tools, commands, and settings. At this point, we will begin introducing MEL commands in place of menu selections. To economize on words in the step-by-step instructions, we will write simply **Enter** in place of saying **Enter the following in the Command Line entry field or the Script Editor input pane**. We'll also from here on refrain from elaborating on the MEL commands and their many flags here in the text with the provision that such details are well documented and easily accessed in Maya's Help Library. In general we will use the long name for a flag the first time it appears and then employ its short name for subsequent uses. Let's begin using MEL commands in the following steps which will take you through creating the cell geometry model.

Make the cube

First, you'll use the `polyCube` command to make an 11 μm cube which will shrink slightly to make a 10 μm diameter cell after smoothing.

Enter:

```
polyCube -width 11 -h 11 -d 11 -subdivisionsX 1 -sy 1 -sz 1 -n cell;
```

Smooth the cube

The `polySmooth` command will subdivide the cube (a hexahedron) into a polyhedron with **n** sides according to the following internal Maya formula:

$$n = 6 \times 2 \exp(2 \times d)$$

where **d** is the value of the polySmooth node `divisions` attribute. In practical terms, a `divisions` setting of **2** gives us **96** sides (or quadrilateral polygons), which is sufficient detail for the deformations required of this initial model.

Enter:

```
polySmooth -continuity 1 -divisions 2 cell;
```

Create and apply a shader

Here you'll make a blue, translucent Phong shader called `cellShader` and apply it to the cell. The translucence will allow you to see the nucleus through the cell's outer membrane (or surface). You can make the shader in the Hypershade and set its color and transparency in the Attribute Editor if you like. Otherwise, you can create and apply the shader by entering the following lines of code in the Script Editor:

```
shadingNode -asShader phong -n cellShader;
setAttr "cellShader.color" -type double3 0.5 0.5 1;
setAttr "cellShader.transparency" -type double3 0.5 0.5 0.5;
select cell;
hyperShade -assign cellShader;
```

Shape the cell with a Lattice Deformer

Lattice Deformers are powerful tools for shaping polygonal and NURBS surfaces and clusters of particles in Maya. Here you'll use a Lattice Deformer to flatten the cell bottom and spread out its edges.

Enter:

```
lattice -divisions 2 5 2 -objectCentered true -ldivisions 2 2 2;
```

Figure 16.12 shows the lattice and cell together and the subsequent steps you'll take, manipulating **lattice points** to deform the cell. A lattice is like a hexahedron and its points like CVs: as you translate, scale, or rotate them, you change the shape of the lattice. This in turn deforms the object under the influence of the lattice. The following steps are guidelines only. In all likelihood your lattice-deformed cell will look a bit different than ours.

1. **Press the hotkey, W, to activate the Move Tool.**
2. **RMB + click over a portion of the lattice and choose Lattice Points.**
3. **Select, move, and scale groups of lattice points to flare and flatten the bottom of the cell as shown in Figure 16.12.**

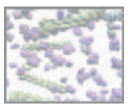
This just gives you a taste of what lattices are capable of. To learn more about the nodes that comprise a Lattice Deformer and its various uses, refer to Maya Help:

Lattice Deformers

Maya Help → Using Maya → Animation, Character Setup, and Deformers → Deformers → Lattice Deformer

Delete history

When you are satisfied with the shape of your cell, delete its history to remove the lattice nodes and make permanent the surface deformations that give the cell its default "resting" shape. You want the Dependency Graph (**DG**) cleared of any nodes



(a) Select the lower eight lattice points.

(b) Use the Move tool to move the lattice points up.

(c) Select the bottom four lattice points.

(d) Move the lattice points up to flatten the bottom of your cell model.

(e) Select the lower eight lattice points and use the Scale tool to splay them apart.

(f) Select the bottom four points and continue to spread the cell bottom using the Scale tool.

FIGURE 16.12

Step by step deformation of the sphere, by moving and scaling the lattice points, in order to create the basic cell geometry. Using the Scale tool doesn't scale the lattice points themselves, but scales the distance between selected points. To scale uniformly in X, Y, and Z, click on the central manipulator (yellow) cube and MMB-drag your mouse.

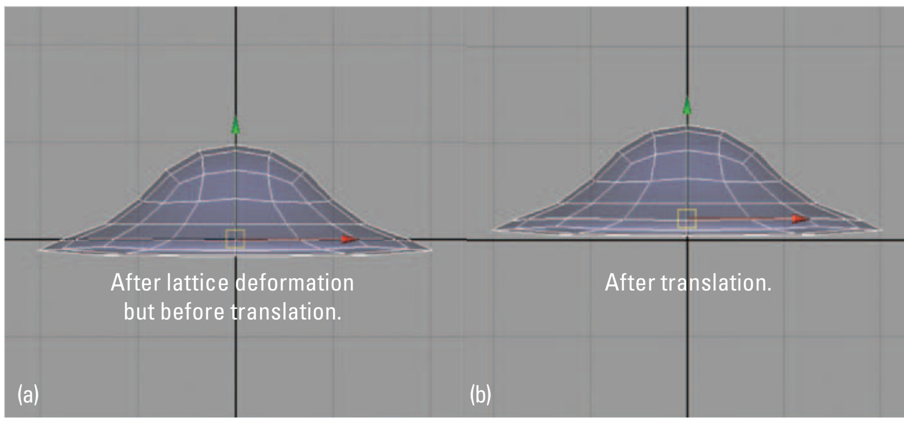


FIGURE 16.13

The XZ plane represents a flat substrate for your cell. After deforming the geometry with a Lattice Deformer, you may need to translate it in Y so that it sits on or slightly above the XZ plane and doesn't intersect it.

that aren't essential to the cell crawl cycle in order to minimize the number of steps needed in the expression to make the cell deform predictably.

Enter:

```
delete -constructionHistory cell
```

Reset the cell's position

Depending on how much you moved the lattice points, your cell may be intersecting the XZ plane, which represents the flat substrate on which you want the cell to crawl. If this is the case, take the following steps (Figure 16.13):

1. **Activate a side or front orthographic view.**
2. **Select the cell and hit the hotkey, W, to activate the Move Tool.**
3. **LMB + click the Y manipulator handle and drag it in the positive Y direction until the cell surface clears the XZ plane** (Figure 16.13 b).

Next, freeze the translate values so that the cell's current position will be its zero position.

Enter:

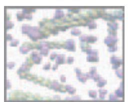
```
makeIdentity -apply true -translate 1 cell;
```

Freeze Transformations

Help → Using Maya → Tools, Menus, and Nodes → Menus → Modify → Modify > Reset Transformations, Freeze Transformations

Rig the cell

Initially, your model requires only one joint, which we've been calling `cellCenter`. The other, `pod1` and `pod2`, will be created by the expression which you will soon build. You can make the `cellCenter` joint easily using the MEL command, joint, which makes a joint and places it at the world origin—right where you want `cellCenter` to start off.



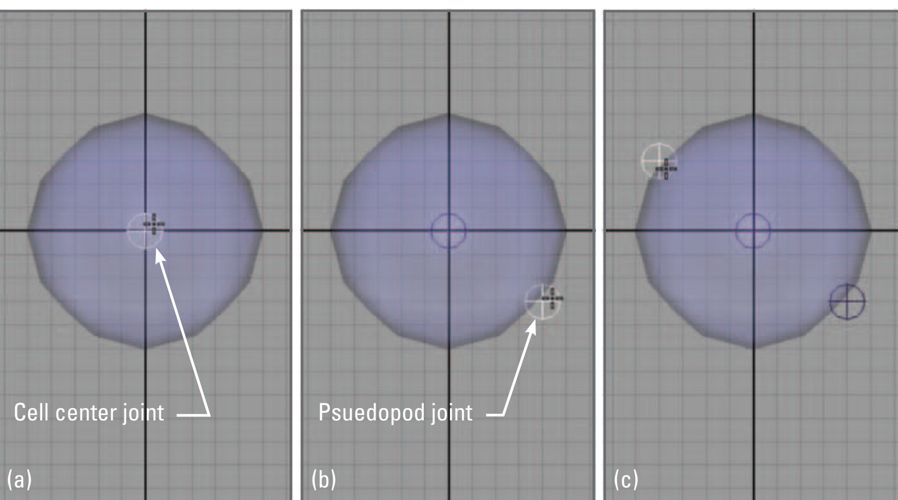


FIGURE 16.14

A top view of the cell geometry as joints are created. With the Joint tool active, a joint is created each time you LMB+click in the scene view. Hold down the X key when you click to constrain the joint to a point on the scene grid. Press Enter on your numeric keypad after each joint click to deactivate the joint tool—otherwise subsequent joints will be parented to the first. (a) Click at the world origin to make the cellCenter joint then press Enter. (b) and (c) Click at the perimeter of the cell to make the pseudopod joints, pressing Enter between each click.

However, we believe it would be useful for you to create `cellCenter` and two other joints using Maya's **Joint tool** in the UI and then bind them to the cell mesh in order to experience manually what your expression script will be doing many times per second automatically. Think of these next steps as building a prototype by hand to see what your rig is capable of before setting it loose through an expression. This will help you better understand the expression as you build it.

Make the joints

1. **Activate the orthographic top view of your scene and adjust your view to appear similar to that in** Figure 16.14**.**

2. **Press the F2 hotkey to activate the Animation menu set.**

3. **Choose Skeleton → Joint Tool ❐.**

4. **In the Joint tool options:**

   (a) **Press Reset Tool.**
   (b) **Under Bone Radius Settings → Short Bone Radius, enter 2.**
   (c) **Press the Close button.**

Your cursor will change to cross hairs to indicate that the Joint tool is enabled. The Short Bone Radius setting determines how large the joints you make will *appear* in the scene view—it has no bearing on a joint's functional properties.

5. **While holding down the X key (to constrain the joint to a point on the grid) click at the world origin to make the cellCenter joint.**

6. **Press Enter on your numeric key pad to complete the joint creation and disable the Joint tool.**

7. **In the Channel Box, rename the joint cellCenter.**

8. **Adjust the Joint Display Scale:**
   (a) **From the main menu set choose Display → Joint Size → Custom.**
   (b) **Enter 5 and then close the Joint Display Scale window.**
9. **Press the repeatLast hotkey, G, to re-enable the Joint tool.**
10. **While holding down the X key click somewhere near the cell perimeter.**
11. **Press Enter on your numeric key pad.**
12. **In the Channel Box, rename the joint pod 1.**
13. **Press G to re-enable the Joint tool.**
14. **While holding down the X key click near the cell perimeter opposite to pod1.**
15. **Press Enter on your numeric key pad.**
16. **In the Channel Box, rename the joint pod 2.**

If you don't press Enter after clicking to make a joint, the Joint tool remains active. If you click again to make a second joint, Maya automatically creates a bone linking the first and second joint through the Scene Hierarchy (the second joint becomes a child of the first). Pressing Enter after the first joint disables the Joint tool: if you then re-enable it and click again, the second joint will have no relationship to the first, which is the scenario we want for the cell animation rig.

Bind the skin to the joints

A skin can be bound to a skeleton in one of two ways: a smooth bind and a rigid bind. In smooth binding, the influence over a given set of mesh CVs is shared among proximal joints. This type of binding is great for smooth, organic-looking deformations of the kind we're after to represent cell surfaces. In rigid binding each CV is influence by only one joint. This produces mechanistic-looking deformations, useful for articulated limbs that are rigid, such as those found on a striding robot or a crustacean for example, where you don't want the surface mesh stretching and bending like an elastic membrane.

When applied, the **Smooth Bind tool** executes the `skinCluster` MEL command behind the scenes. This in turn creates a `skinCluster` node, which stores information on how joint influences are distributed throughout the mesh. In the expression you'll build shortly, you will use the `skinCluster` command to smooth bind the cell to its joints. At this point, however, just use the Smooth Bind tool:

1. **Select the cell and the three joints you made. The selection order isn't important for the Smooth Bind tool.**
2. **In the Animation menu set, choose Skin → Smooth Bind ❐.**
3. **In the Smooth Bind options window, choose Edit → Reset Settings.**
4. **Press the Bind Skin button.**

If Maya generates an error, make sure that all four objects are selected and repeat steps 2–4.

Deform the cell

Your cell is now rigged! Take a minute or two to move each of the joints around using the Move Tool and observe the resulting changes to the cell surface (Figure 16.15).

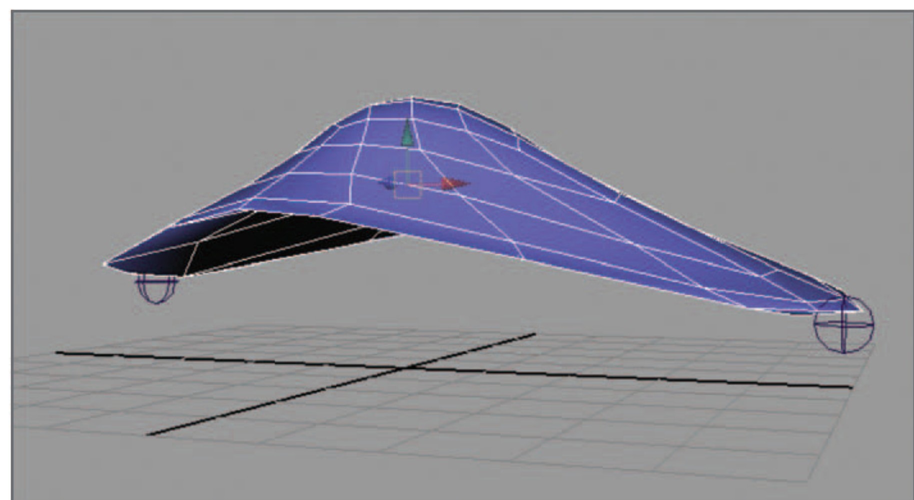


FIGURE 16.15
Once you've rigged your cell by binding the mesh to the joints, select and move each joint using the Move Tool to see the deformation effects on the surface. The centerCell joint is selected in this picture.

Note that each joint has keyable transform attributes which are displayed in the Channel Box. Your cell crawl expression will set those attribute values (but not key them) according to the numbers it computes for the random walk, the chemotactic influence, and the duration of each stage—protrusion, traction, and retraction—of the cell crawl cycle.

Reset the cell

Before moving on to building the expression reset your cell by returning the three joints to their original locations and unbinding (or detaching) the cell from the joints and then deleting pod1 and pod2:

1. **Select the three joints.**
2. **Choose Skin → Go To Bind Pose.**
3. **Select the cell.**
4. **Choose Skin → Detach Skin ❐.**
5. **In the Detach Skin options window choose Edit → Reset Settings.**
6. **Press the Detach button.**
7. **Select and pod1 and pod2. Press the Delete key on your keyboard.**

Add custom attributes

Now you'll add six custom attributes to the cell transform node. They will be a means for quickly entering key expression variables rather than having to search for them in the Expression Editor. All you'll have to do to assign the variables is select the cell

in the Scene View or Outliner and enter values in the appropriate attribute fields in the Channel Box. Note the `minimum (min)` and `maximum (max)` values. These are used to prevent the user from entering values that would result in errors, such as a zero speed value for one of the joints.

Enter the following lines in the Script Editor:

```
addAttr -longName pod1Speed -attributeType double -min 1 -max 10
  -dv 2 |cell;
setAttr -edit -keyable true |cell.pod1Speed;
addAttr -ln centerSpeed -at double -min 1 -max 10 -dv 1 |cell;
setAttr -e -keyable true |cell.centerSpeed;
addAttr -ln pod2Speed -at double -min 1 -max 10 -dv 2 |cell;
setAttr -e -keyable true |cell.pod2Speed;
addAttr -ln waitFactor -at double -min 0.1 -max 10 -dv 1 |cell;
setAttr -e -keyable true |cell.waitFactor;
addAttr -ln chemoMagnitude -at double -min 0 -max 10 -dv 5 |cell;
setAttr -e -keyable true |cell.chemoMagnitude;
addAttr -ln chemoTheta -at double -min 0 -max 360 -dv 0 |cell;
setAttr -e -keyable true |cell.chemoTheta;
```

-dv is short for defaultValue. You can later change the values of these attributes in the Channel Box.

You can change the minimum and maximum limits on these attributes by using the addAttr command with the edit flag. For example:

```
addAttr -edit -min 3 -max 6 cell.pod1Speed; // Change the min and
  max limits.
```

Add the cell nucleus

For the cell nucleus, create, scale, and position a NURBS sphere so that it fits comfortably within the cell model (Figure 16.16).

1. **Enter the following line in the Script Editor:**

   ```
   sphere -radius 5 -axis 0 1 0 -name nucleus;
   ```

2. **Press W to activate the Move Tool then drag the nucleus in the positive Y-direction until it's mostly inside the cell.**
3. **Press R to activate the Scale tool then scale the nucleus in the Y-axis so that it fits within the boundaries of the cell.**

To make the nucleus move as the cell does, connect its translate attribute to that of the `cellCenter` joint.

4. **Enter the following line in the Script Editor:**

   ```
   connectAttr cellCenter.translate |nucleus.translate;
   ```

Now that your cell geometry is ready—complete with custom attributes—save your scene. You will use it again shortly. First you'll apply your knowledge of rigging and animating with joints to the cell crawl expression.

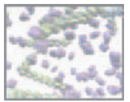


FIGURE 16.16

After creating the nucleus model using the **sphere** command: (a) Move the nucleus into position within the cell model. (b) Scale the nucleus so that it fits within the boundaries of the cell surface.

Methods: Encoding the algorithm

Before beginning, you may wish to refer back to Figures 16.09 and 16.10, the flowchart and schematic illustration for a quick refresher on the steps you're about to encode. Unlike the earlier projects you will not use procedures here, but rather create one expression that calculates the crawl parameters and makes, moves, and destroys the pieces needed to make your cell crawl, on a frame by frame basis. That's not to say the code in your expression couldn't be broken out into several subroutines that are then called when needed. However, this script is sufficiently concise that we find it simpler to manage in one chunk.

The cellCrawl expression

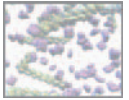
As you saw back in *Chapter 12*, an expression can be made in one of two ways. The first uses the MEL command `expression` and a single line of code. If the code is sufficiently long, you can break it up into multiple lines as long as you use the escape line break notation, `\n`, at the end of each line. In longer expressions, escaping line breaks becomes tedious and can lead to errors when you add new lines to the code and forget to escape them. For this project we recommend the second way to make an expression: compose it in multiple lines—as you would a procedure—then copy/paste it into Maya's Expression Editor and press Create. Making an expression this way avoids the errors associated with the extra step of building it as a single MEL statement. As in previous chapters, we also suggest that you compose your expression in a text editor (other than Maya's Script Editor), saving it periodically as you follow along with the instructions below. When you want to test bits of code in Maya, just copy and paste them from your text editor into Maya's Script Editor.

Some header information will be helpful when you refer back to this expression at a later date and to help others understand your code.

```
/***** cellCrawl.txt *****/
/*
Date: April 2006.
Authors: Jason Sharpe, Charles Lumsden, Nick Woolridge.

Description:
This expression animates a crawling cell.

To use this script:
Your Maya scene must contain a NURBS or polgon model called cell
and a joint called cellcenter, both located at the world origin.
The cell model must have the following attributes (sample default
values are given):
pod1Speed = 2; centerSpeed = 1; pod2Speed = 2; waitFactor = 1,
chemoMagnitude = 5; chemoTheta = 0.
Note: All angles are in degrees, measured CCW from the positive
z-axis.
Copy and paste this entire script into Maya's Expression Editor and
press the Create button. Press play to animate the cell.
*/
```

Next, declare the main variables. You'll use global variables for the parameters that are set at the start of a cycle and then must be read at each frame in the cycle. Being global, once these variables are set, they can be read at any time and anywhere in the Maya environment (i.e. in an expression, a procedure, or in the Script Editor).

```
/***** DECLARE THE VARIABLES *****/
/*
$pod1Pos       XYZ position of the leading pseudopod joint, pod1.
$pod2Pos       XYZ position of the trailing pseudopod
               joint, pod2.
$centerPos     XYZ position of the cell center joint.
$ligandPos     Location where the leading pseudopod binds to the
               substrate.
*/
global vector $pod1Pos, $pod2Pos, $centerPos, $ligandPos;

/*
$theta         Chemoattractant direction (an angle in degrees).
$c             Magnitude of the chemoattractant concentration.
$alpha         Random motility direction (an angle).
$gamma         Difference between $alpha and $theta.
$alphaPrime    Final motility direction, accounting for both
               random motility and chemotaxis.
$cellRadius    Radius of the flattened cell (~10µm).
$reach         Distance travelled the cell in a given cycle.
$waitAverage   The average time, in frames, the cell waits before
               moving.
$wait          A random wait time.
*/
float $theta, $c, $alpha, $gamma, $alphaPrime, $cellRadius, $reach;
float $waitAverage, $wait;

/*
$minDist       Minimum possible value of $reach.
$maxDist       Maximum possible value of $reach.
```




```
$pod1Speed          Rate of protrusion in µm/frame.
$pod2Speed          Rate of retraction in µm/frame.
$waitFactor         Multiplier to scale the waiting time.
$stateZero          Number of frames spent waiting.
*/
float $minDist, $maxDist, $pod1Speed, $pod2Speed, $waitFactor,
  $stateZero;

/*
$pod1Vect           Vector added to the position of pod1 for each
                    step in a crawl cycle.
$pod2Vect           Same as above for pod2.
$centerSpeed        Rate of cell center translocation in µm/frame.
$centerVect         Vector added to the position of cellCenter for
                    each step in a crawl cycle.
*/
vector $pod1Vect, $pod2Vect, $centerSpeed, $centerVect;

/*
$pod1Steps          Number of frames needed for protrusion.
$centerSteps        Number of frames needed for translocation.
$pod2Steps          Number of frames needed for retraction.
*/
global int $pod1Steps, $centerSteps, $pod2Steps;

/*
$clusterNames[]     A list of skinCluster node names.
$bindPoseNames[]    A list of bindPose node names.
$podNames[]         A list of pods names.
*/
string $clusterNames[], $bindPoseNames[], $podNames[];

/*
$i                  Counts the steps in a crawl cycle
*/
int $i;
```

A bindPose node is created when you bind a skin (surface) to a joint. It stores the joint's world matrix (translation and rotation) at the time of binding.

Initialize the variables

At this point, the expression uses the `getAttr` command to read the custom attribute values you assigned to the `cell` transform node.

```
/***** INITIALIZE THE VARIABLES *****/

$theta = `getAttr cell.chemoTheta`;
$c = `getAttr cell.chemoMagnitude`;
$pod1Speed = `getAttr cell.pod1Speed`;
$pod2Speed = `getAttr cell.pod2Speed`;
$centerSpeed = `getAttr cell.centerSpeed`;
$waitFactor = `getAttr cell.waitFactor`;
$cellRadius = 10;
$minDist = 20;
$maxDist = 40;
```

Reset the cell model

Each time you set the current time indicator to frame 1 the cell should return to the world origin and prepare for the next sequence of crawl cycles. This preparation includes deleting the skinCluster, bindPose, and pseudopod joint nodes. To do this you'll use a string array to store a list (using the ls command) of each type of node by name and then delete the array contents. This is a safer technique than deleting items by name, as in delete pod1. If no item named pod1 exists, Maya will generate an error and halt the expression. On the other hand, if the array of objects called pod1* is empty and you delete the array contents, Maya will only issue a warning,

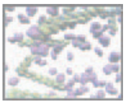
```
Warning: Nothing is selected. Select objects or components to
  delete.
```

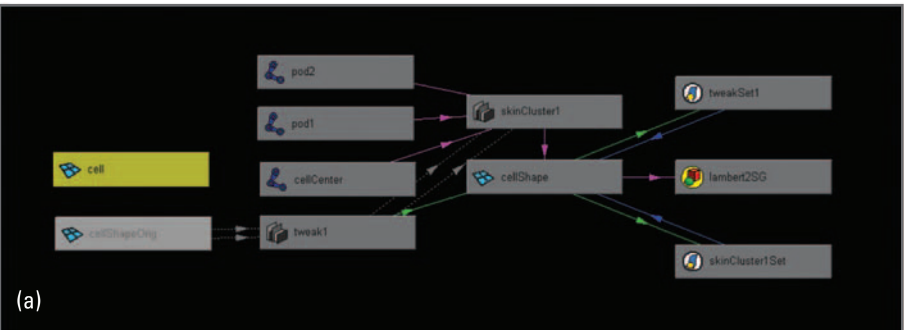
and will continue to execute the expression (do not enter the above line as part of the expression).

```
/***** RESET THE CELL MODEL *****/

if (frame == 1) {

    // Reset the crawl step counter.
    $i = 0;

    // Delect existing skinCluster nodes when you return to frame 1.
    $clusterNames = `ls -long "skinCluster*"`;
    delete $clusterNames;

    // Delect existing bindPose nodes when you return to frame 1.
    $bindPoseNames = `ls -long "bindPose*"`;
    delete $bindPoseNames;

    // Delect existing pod objects when you return to frame 1.
    $podNames = `ls -tr "pod*"`;
    delete $podNames;

    // Center the cell center joint.
    $centerPos = <<0, 0, 0>>;
    setAttr cellCenter.translate ($centerPos.x) ($centerPos.y)
      ($centerPos.z);
```

If you delete a skinCluster node when it is connected to an object, Maya locks the object's connected attributes. The following lines unlock the cell's transform attributes, ensuring they are free to be connected to a new skinCluster node.

```
    // Unlock the cell's transform attributes.
    setAttr -lock 0 "cell.tx";
    setAttr -lock 0 "cell.ty";
    setAttr -lock 0 "cell.tz";
    setAttr -lock 0 "cell.rx";
    setAttr -lock 0 "cell.ry";
    setAttr -lock 0 "cell.rz";
    setAttr -lock 0 "cell.sx";
    setAttr -lock 0 "cell.sy";
    setAttr -lock 0 "cell.sz";

    // Center the cell.
    setAttr cell.translate ($centerPos.x) ($centerPos.y)
      ($centerPos.z);
```





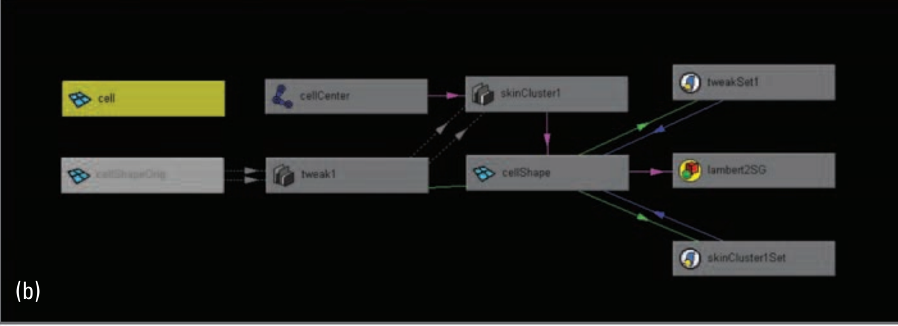


FIGURE 16.17

These images of the Hypergraph show the DG nodes of the cell model in different stages of operation: (a) At the start of each crawl cycle, the joint nodes pod1 and pod2 are made. They remain in existence during the cycle. (b) After each cycle is completed, and at frame 1 when the cell is reset, pod1 and pod2 are deleted and the cell is bound only to the cellCenter joint.

```
    // Bind the cell to the cell center joint.
    print "Cell is set to go!\n";
    $clusterName1 = `skinCluster cellCenter cell`;
    // Select the cell to make its attributes available in the
      channel box.
    select cell;
  } // End if (frame == 1). Cell is reset.
```

Figure 16.17 shows the effect of the above section of code on the DG for the cell.

Print commands

As you learned in *Chapter 12*, the `print` command in the above code section sends a message to the Command Line and the Script Editor telling you that this part of the expression executed and the cell is now ready to crawl. Printing information about an expression or MEL script can be very helpful when debugging code. Printing allows you to see if variables and attribute values are being assigned the values that you think they are. We will continue to use `print` throughout the code. When the expression is done and you run the cell crawl simulation, the Script Editor will display a running report of what's going on behind the scenes.

Set up the next crawl cycle

In this section of code you will calculate the values necessary to move the cell, including the random motility direction, the effect of the chemoattractant, and the step

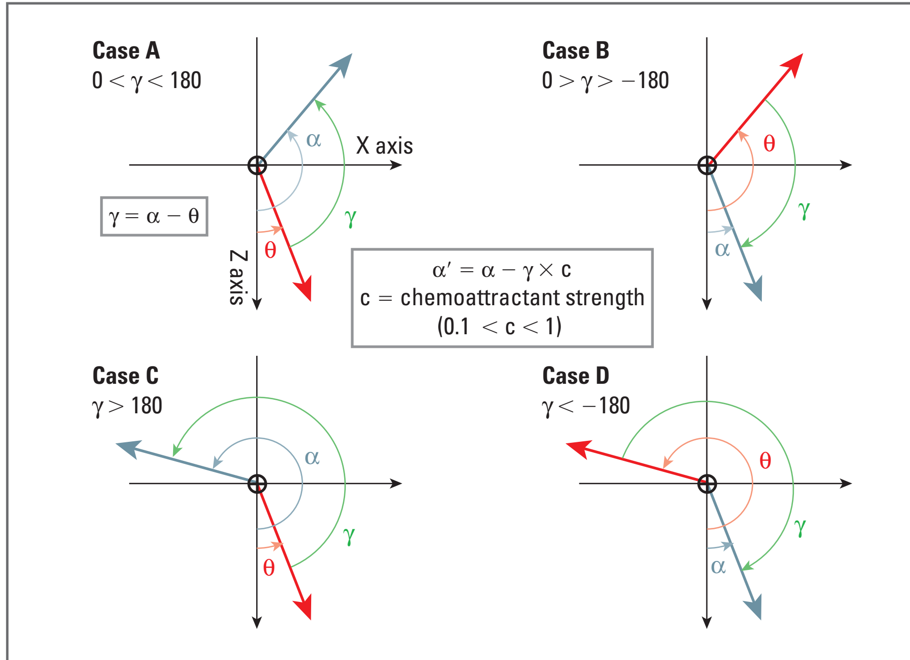


FIGURE 16.18

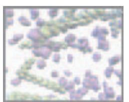
Four cases to consider for the angle ($\gamma = \alpha - \theta$) when calculating alphaPrime, the direction resulting from both random motility and chemotaxis.

sizes for each of the three motility stages: protrusion, traction, and retraction. In order to execute this code, the following conditions must exist:

1. **The cell must be at the beginning of a crawl cycle, i.e. $i == 0.**
2. **The frame number must be > 1 (we're reserving frame 1 for resetting the model).**
3. **The variable skinCluster[0] must have an assigned value, the name of the existing skinCluster node. If it doesn't, it's because the scene file has just been opened and hasn't been reset to frame 1 yet, or you edited the expression after stopping the cell in mid-crawl. In such cases, checking if a skinCluster node exists prevents an execution error.**

You'll use Maya's rand() function to generate the random motility angle, $alpha and pseudopod protrusion distance. Feel free to experiment with more complex probability models once you have the basic code up and running, for example a Gaussian (bell-curve) form. The chemoattractant angle $theta and magnitude $c were read in from the cell attribute chemoMagnitude when you initialized variables. We have given the name $gamma to the difference between $alpha and $theta. $gamma will be used with $c to calculate $alphaPrime, the cell direction that accounts for both random motility and chemotaxis (see Figures 16.05 and 16.10). There are four possible cases we must consider for the value of the angle $gamma which are shown in Figure 16.18.

```
/***** SET UP THE NEXT CRAWL CYCLE *****/
if ($i == 0 && frame > 1 && `objExists $clusterNames[0]`) {

    // Calculate the random angle $alpha.
    $alpha = rand(0,360);
```




```
float $gamma = $alpha - $theta;
print ("\nStarting new crawl\n$gamma = " + $gamma + "\n");

// CASE A.
if ($gamma > 0 && $gamma <= 180){
   $alphaPrime = $alpha - $gamma*$c/10; print "CASE A\n";
}
// CASE B.
else if ($gamma < 0 && $gamma >= -180){
   $alphaPrime = $alpha - $gamma*$c/10; print "CASE B\n";
}
// CASE C.
else if ($gamma > 180){
   $alphaPrime = $alpha + (360 - $gamma)*$c/10; print "CASE C\n";
}
// CASE D.
else if ($gamma < -180){
   $alphaPrime = (360 + $alpha) - (360 + $gamma)*$c/10; print
     "CASE D\n";
}

// Calculate the pseudopod reach.
$reach = rand($minDist, $maxDist);
print ("$alpha = " + $alpha);
print (", $alphaPrime = " + $alphaPrime + ", $reach = " +
  $reach + "\n");

// cellCenter position.
$centerPos =`getAttr cellCenter.translate`;

// pod1 position.
$tmpX = $centerPos.x + $cellRadius * sin(deg_to_
  rad($alphaPrime));
$tmpZ = $centerPos.z + $cellRadius * cos(deg_to_
  rad($alphaPrime));
$pod1Pos = <<$tmpX, 0, $tmpZ>>;

// pod1 position.
$tmpX = $centerPos.x + $reach * sin(deg_to_rad($alphaPrime));
$tmpZ = $centerPos.z + $reach * cos(deg_to_rad($alphaPrime));
$ligandPos = <<$tmpX, 0, $tmpZ>>;

// pod1 position.
$tmpX = $centerPos.x + $cellRadius * sin(deg_to_rad
  ($alphaPrime + 180));
$tmpZ = $centerPos.z + $cellRadius * cos(deg_to_rad
  ($alphaPrime + 180));
$pod2Pos = <<$tmpX, 0, $tmpZ>>;

/* Delete existing pseudopod joints. This is a safeguard for
when the file is saved in mid-crawl, and then reopened and
played.*/
string $podNames[] =`ls -tr "pod*"`;
delete $podNames;

// Make the pseudopod joints.
select -clear;
```

If a joint happens to be selected when you use the joint tool or joint command, Maya will automatically create a bone and parent the second joint to the first, which is not what we want to happen. Clearing the selection before making a new joint ensures no bone is created and the joint remains unparented. The short name for the -clear flag is -cl, which we'll use from now on.

```
joint -p ($pod1Pos.x) ($pod1Pos.y) ($pod1Pos.z) -n pod1; select -cl;
joint -p ($pod2Pos.x) ($pod2Pos.y) ($pod2Pos.z) -n pod2; select -cl;
```

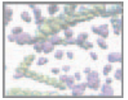
The use of brackets around $pod1Pos.x, etc. forces Maya to return the value of the attribute. Below, the skinCluster command is used with the edit and addInfluence flags to *add* the joints pod1 and pod2 to the existing skinCluster node. The -dropoffRate attribute flag determines how a joint's influence on the bound skin decreases, or drops off, with increased distance from the skin. The higher the value, the quicker the dropoff. The default value is 4.

```
// Add pod1 and pod2 to the existing skinCluster1.
skinCluster -edit -dropoffRate 4 -addInfluence pod1 $clusterNames[0];
skinCluster -edit -dropoffRate 4 -addInfluence pod2 $clusterNames[0];

// $tmpVect is the vector that translates each of the cell's joints.
vector $tmpVect = $ligandPos - $pod1Pos;
// $mag is the scalar distance the cell will travel.
float $mag = mag($tmpVect);

// pod1 step size; the protrusion distance per frame.
$pod1Steps = $mag/$pod1Speed;
$pod1Vect = $tmpVect/$pod1Steps;

// cellCenter step size; the traction distance per frame.
$centerSteps = $mag/$centerSpeed;
$centerVect = $tmpVect/$centerSteps;

// pod2 step size; the retraction distance per frame.
$pod2Steps = $mag/$pod2Speed;
$pod2Vect = $tmpVect/$pod2Steps;

print ("$pod1Steps = " + $pod1Steps + ", $centerSteps = ");
print ($centerSteps + ", $pod2Steps = " + $pod2Steps + "\n");
```

Next, the expression calculates the wait time between locomotive excursions. For starters, you'll make the average wait time value one half the crawl cycle time for the cell (*time* here refers to *frames*). You'll then use this average value, $waitAverge, to randomize the wait time, $wait according to the following formula[11]:

$wait = $waitAverage*log(1/$rnd)*$waitFactor;

Where $rnd is a random number between 0 and 1 and $waitFactor is a multiplier that was assigned the value of the cell's custom waitFactor attribute. By changing the value for $waitFactor in the Channel Box you can then easily scale the wait time up or down.




```
        /* In state zero, the cell waits before another excursion. Here
        you'll generate a random wait time of average value:$waitAverage
        frames. */
        $waitAverage = ($pod1Steps + $pod2Steps + $centerSteps)/2;
        float $rnd = rand(0,1);
        $wait = $waitAverage*log(1/$rnd)*$waitFactor;

        $stateZero = ceil($wait);
        // ceil (for "ceiling") rounds $wait up to the nearest integer.
        print ("$waitAverage = " + $waitAverage + ", stateZero = "
          + $stateZero + "\n");
```

The crawl cycle counter is then set to the sum of the pod1, cellCenter, pod2, and stateZero steps sizes.

```
        // Set the crawl cycle counter.
        $i = $pod1Steps + $pod2Steps + $centerSteps + $stateZero;
        print ("$i = " + $i + "\n");

    } // End if.
    // The crawl cycle is now set up.
```

Increment the crawl cycle

The following section of code is executed at every frame for which the crawl cycle counter, $i, is greater then zero. Its function is to move first pod1 through each of its steps, then cellCenter, followed by pod2. The time remaining in each stage is given by $i minus the time for the remaining stages. When the cell has finished moving—when the retraction stage is finished and pod2 has returned to its default position relative to cellCenter—the cell remains in state zero until $i has counted down to zero. At this point the cell is detached from its skeleton (the three joints), pod1 and pod2 are deleted, then the cell is rebound to cellCenter using the skinCluster command.

```
    /***** Increment the crawl cycle.*****/
    if (frame > 1){

        // Increment pod1 toward ligandPos.
        if ($i > ($pod2Steps + $centerSteps + $stateZero)) {

            // Move the pseudopod, pod1.
            $pod1Pos = $pod1Pos + $pod1Vect;
            setAttr pod1.translate ($pod1Pos.x) ($pod1Pos.y) ($pod1Pos.z);
        }
        else if ($i <= ($pod2Steps + $centerSteps + $stateZero) && $i >
          $pod2Steps + $stateZero) {

            // Move cellCenter.
            $centerPos = $centerPos + $centerVect;
            setAttr cellCenter.translate ($centerPos.x) ($centerPos.y)
              ($centerPos.z);
        }
        else if ($i <= ($pod2Steps = $stateZero) && $i > $stateZero) {

            // Move the tail, pod2.
            $pod2Pos = $pod2Pos + $pod2Vect;
            setAttr pod2.translate ($pod2Pos.x) ($pod2Pos.y)
              ($pod2Pos.z);
        }
```

```
        // Increment the counter.
        $i -= 1;
        print ("$i = " + $i + "\n");

        // If cell has completed a crawl step...
        if ($i == 0 && frame > 3) {

            // Unbind the cell skin.
            print "*************** DETACHING SKIN *****************\
              n\n";
            skinCluster -edit -unbind cellShape;

            // Delete the pseudopod joints
            $podNames = `ls -tr "pod*"`;
            delete $podNames;
```

When you detach the `cell` from its rigging, it snaps back to the world origin since its translate values never changed from 0,0,0 (only its CVs were moved by the joints). You must therefore move it to the location of `cellCenter` before rebinding it using the `skinCluster` command.

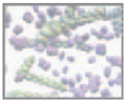
```
            // Move the cell to the location of cellCenter.
            setAttr cell.translate ($centerPos.x) ($centerPos.y)
              ($centerPos.z);

            // Rebind the cell to the center joint.
            skinCluster cellCenter cell;

        } // End if ($i == 0 && frame > 3)
    } // End if (frame > 1).

    // End cell crawl expression.
```

That's it for the coding. It's now time to enter the expression in Maya and get your cell crawling.

Methods: **Loading the script**

With your script now complete, turn it into an expression within the cell geometry scene file you created earlier in this chapter.

1. **Start Maya, open your cell geometry cell file, and set the current time indicator to frame 1.**
2. **Select and copy the expression entire script in your text editor.**
3. **In Maya, enter `ExpressionEditor` in the Command Line to launch the Expression editor, or select it from the menu Windows → Animation Editors → Expression Editor.**
4. **LMB + click in the Expression text field.**



5. **Press Ctrl + V to paste your expression.**

6. **Press the Create button at the bottom of the Expression Editor.**

If Maya accepts your script, the Command Line will display the following line:

```
// Result: expression1 //
```

You can rename the expression by entering text in the Expression Name field at the top of the Expression Editor.

If, however, Maya detects an error in your script, it may or may not create an expression depending on the type of error. In either case, you'll need to debug the script: open the Script Editor and look for error notices that point to specific lines in your script. If your text editor supports line numbers, you can then track and fix errors line by line. You can also compare your script directly with the complete script we've included on the CD-ROM:

16_Mobile_Cell/MEL/cellCrawl.txt

Results: Running the script

When you have successfully created the expression, you need only press Play to set your cell crawling. Each time you press Rewind the cell should return to the world origin, ready for another excursion. Try varying the cell's custom attribute values in the Channel Box to observe their effects. If you included the various print commands listed in the code above, you can get an ongoing report of what the cell crawl algorithm is doing behind the scenes: open the Script Editor and observe the data that displays as the simulation plays.

Data I/O

If you wish to try out the Data I/O methods described in *Chapter 13*, this crawling cell model makes a good test subject. You can export the cell's trajectory as a list of XYZ coordinates and analyze it using the statistics software of your choice. In the *Further reading* section we have listed resources concerning cell migration modeling many of which provide information on the statistics of cell migration.

Troubleshooting

We found this scene file to be unstable in Maya for Mac OS X; the application crashed every time we attempted to open the file. This is due to a problem with the way Maya evaluates the cell crawl expression when it opens the file. An easy workaround for this problem is to use a safety check that disables the expression unless a condition is met. For example, enclose the entire expression within a conditional statement, as follows:

```
global int $safety;
if ($safety == 1) {

   // ENTIRE EXPRESSION SCRIPT.

};
```

When you start Maya and/or open the scene in Maya for Mac OS X, the default value of safety is zero, therefore the expression won't evaluate and Maya won't crash. Once the scene file loads, Maya is stable, and you can enter the following line in the Command Line to *unlock* the expression before playing the animation:

```
global int $safety = 1;
```

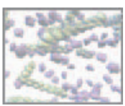
Another peculiarity in Maya for Mac OS X is the behavior of the `Evaluation` setting in the Expression Editor. We find it reverts to **On Demand** no matter how many times we set it to **Always**, which should be its default setting. If your Expression Editor is set to On Demand, the cell crawl expression will not evaluate properly and cell will not move. We found that setting Evaluation to Always once fixed the problem even though the Evaluation setting *indicator* in the Expression Editor reverted to **On Demand**.

Summary

With the 2-pod model up and running you are ready for explorations that will take you further into the science of cell locomotion. Real cells, for example, may have a number of pseudopods extended at any one time, from which a "winner" seems to emerge that points the direction for the cell's next migratory step. How could a many-pod version of your current model be created? On a planar surface, the leading pseudopod has a dynamic anatomy that includes not only extension by an intricate, delicate flattening and ruffling of the leading edge. Moreover, your cell model is a behavioral "container" that invites you to equip it with chemistry, such as the turnover of actin filaments and intracellular signal ligands, to drive the deformation biophysics. These are all exciting problems at the frontier of current research. In the next two chapters we turn to another amazing frontier, which is the world of cell behavior when locomotion is released from 2D wide open planes and the cells moved into crowded 3D jungles of tangled fibers—matrix worlds like those holding your body together!

References

1. Wolf K, Mazo I, Leung H, Engelke K, von Andrian UH, Deryugina EI, Strongin AY, Bröcker E-B, Friedl P: Compensation mechanism in tumor cell migration: Mesenchymal–amoeboid transition after blocking of pericellular proteolysis. *Journal of Cell Biology* 160: 267–277, 2003.
2. Pollard T: Regulation of actin filament assembly by arp2/3 complex and formins. *Annual Review of Biophysics and Biomolecular Structure* 36: 451–477, 2007.
3. Paluch E, Sykesa C, Prosta J, Bornens M: Dynamic modes of the cortical actomyosin gel during cell locomotion and division. *Trends in Cell Biology* 16: 5–10, 2006.
4. Condeelis J, Singer RH, Segall JE: The great escape: When cancer cells hijack the genes for chemotaxis and motility. *Annual Review of Cell and Developmental Biology* 21: 695–718, 2005.
5. Mitchison TJ, Cramer LP: Actin-based cell motility and cell locomotion. *Cell* 84: 371–379, 1996.
6. Berg HC: *Random Walks in Biology*, 2nd edn. University Press, Princeton NJ, 1993.



7. Rudnick J, Gaspari G: *Elements of the Random Walk: An Introduction for Advanced Students and Researchers*. Cambridge University Press, Cambridge UK, 2004.

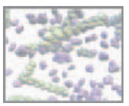
8. Maheshwari G, Lauffenburger DA: Deconstructing (and reconstructing) cell migration. *Microscopy Research and Technique* 43: 358–368, 1998.

9. Zygourakis K: Quantification and regulation of cell migration. *Tissue Engineering* 2: 1–16, 1996.

10. Othmer HG, Dunbar SR, Alt W: Models of dispersal in biological systems. *Journal of Mathematical Biology* 26: 263–298, 1988.

11. Gillespie DT: *Markov Processes: An introduction for Physical Scientists*. Academic Press, San Diego CA, 1992.

17 Growing an ECM scaffold



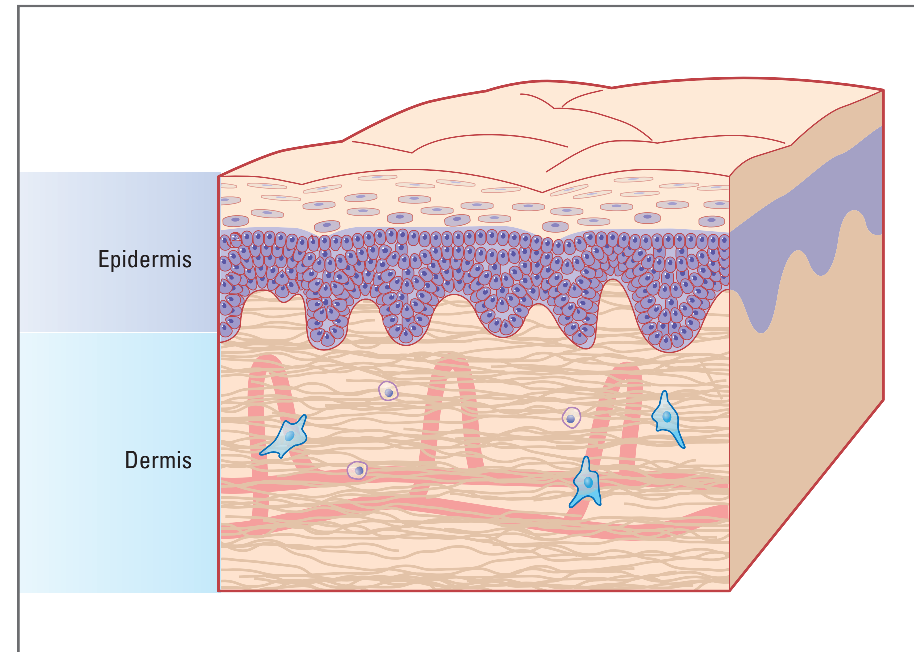


FIGURE 17.01

Although tissue structure and function vary widely throughout the body, all tissues are composed of cells and ECM. Connective tissues—like the dermal layer of skin shown here—are composed mostly of ECM, with relatively few cells. For other tissues—like the epidermis—the opposite is true: many cells, with relatively little ECM (by volume) connecting them.

Introduction

For all their mystery and beauty, the body's cells do not live or work in isolation. Each organ and tissue is an intricate society of cellular specialists, whose chemical and mechanical activities assure health. In the brain, for example, information is processed and transmitted by electrically linked cells called neurons, which in turn are cradled in nests of supporting cells (the glia) and fed oxygen and chemical energy by the blood cell's of the brain's circulation. In the heart, specialized nerve cell circuits time-out the rhythm of the cardiac beat, triggering the heart muscle cells to contract in wave-like patterns.

Tissue architecture thus varies widely throughout the body and is highly specialized for different functions; the microscopic composition and dynamic properties of heart muscle, we see, are decidedly different from that of bone, brain, kidney, skin, and so on. The approaches taken to modeling activity in these cellular societies can likewise vary from one tissue to the next. Nonetheless, there are characteristics—design parameters, if you will—that recur through bodily tissues and serve as powerful organizing themes. For instance, tissues generally are composed of cells and **extracellular matrix** (**ECM** or *matrix* for short). Look through a microscope at a tissue and you will see arrays of cells embedded within ECM, the long interweaving ECM fiber molecules (protein and carbohydrate polymers) making a pliant 3D scaffold to which the cells adhere.

While cells perform localized functions such as information processing (e.g. brain), protein synthesis and secretion (e.g. bone), and contraction (e.g. muscle), the ECM

provides support for these activities to work together. When modeling a tissue, one therefore is modeling cells and ECM in silico.

Problem overview

The dermis: A model tissue

In this chapter, you will build an idealized ECM model comprised of structural protein fibers like those found in connective tissue such as the dermis, a deep layer of the skin. Cell-matrix interactions have been deemed highly important in the progression and treatment of disease. Notable examples include tumor metastasis, generative disorders of the brain (including Alzheimer's disease), wound healing, and tissue regeneration. The fields of tissue engineering and regenerative medicine are concerned with problems in wound repair and tissue transplantation. Connective tissue makes an excellent 3D in silico model tissue because it has both geometric *and* physical properties that impact normal physiology and disease (Figure 17.0 2).

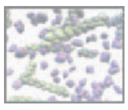
Your project will focus on the dermis, the connective tissue layer of skin that is rich in collagen and elastin proteins, and provides much of the skin's structural integrity. Figure 17.01 shows the dermis in relation to other components of the skin. The dermis is crucial to both health and disease. In the treatment of deep burns, diabetic ulcers, and large wounds requiring skin replacement, missing or damaged dermis is often damaged and must be surgically replaced with substitute scaffolds to promote the skin's natural regeneration. The success or failure of a tissue replacement depends on several factors including its ability to provide a suitable structural and mechanical environment for the patient's own cells (called host cells). Indeed, a critical issue in the surgical repair of wounds is the ability of host cells to effectively infiltrate—by cell migration—the substitute dermis, after which they begin synthesizing a new, autologous dermal ECM scaffold of collagen, elastin, and proteoglycans. The fibrous ECM molecules become the foundation for regenerated skin. It is now understood that fibroblast cells (the cell type largely responsible for dermal wound repair) are highly sensitive to the 3D architecture of dermal collagen fiber bundles, and tissue **morphometry** is therefore a focus of much work in regenerative medicine. Cells are also sensitive to mechanical traits such as ECM tension and compliance. If you wish to learn more about the role of the ECM in regulating cell behavior and gene expression, the chapter's references 1–3 will take you beyond the brief discussion we have time for here.

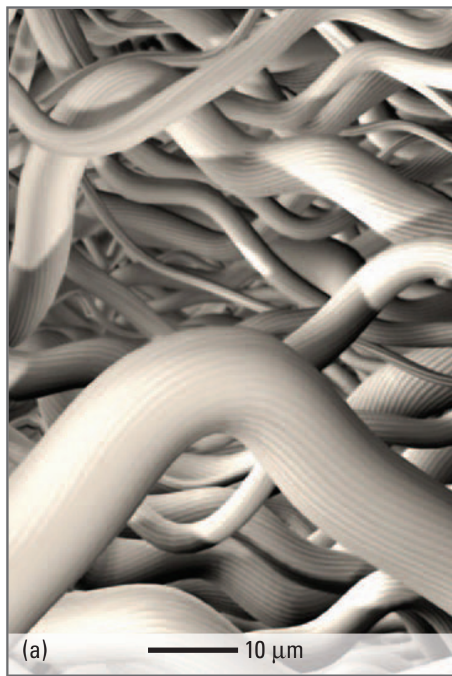
Fibers and bundles: There appears to be no consensus among authors regarding the definitions of **fiber** and **fiber bundle** in reference to collagen. The terms **fibril** and **fasciculus**, used in different ways by different authors, further confuse the issue. In dermis, collagen is arranged in bundles of various diameters and there appears to be no practical distinction between what constitutes a fiber and what does a bundle. For clarity, we use the term **fiber** to refer to a collection of individual collagen molecules that forms a structural unit in the dermis—be it a "fiber" or a "fiber bundle". For our purposes a **fiber** ranges from 1 to 50 μm in diameter.

In this project you will deal with the challenges of the scaffold geometry. The size and arrangement of the dermal scaffold bundles, including the negative space between them, presents cells with an array of structural information that can at once promote, inhibit, and guide their migratory behavior. It is with this in mind that you will embark on the project: modeling a 3D ECM scaffold that embodies general morphometric properties of the dermal collagen scaffold and can be used to study patterns of migration for simulated cells.

Parameters of the dermis

Figure 17.03 shows dermis samples at three levels of magnification. In the human body, the dermis varies in thickness from 0.3 mm on the eyelid to 3 mm on the back.





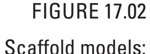


FIGURE 17.02

Scaffold models:

(a) A procedural model of the collagen component of dermis, similar to the one you'll build in this project. In this example, a procedural texture was used to give the ribbed appearance of many small collagen fibrils that make up the larger fibers.

A procedural model of **cancellous bone**, an ECM structure widely studied for its role in surgical bone graft implantation.

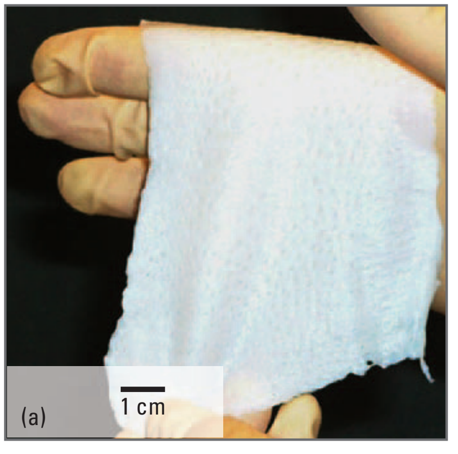
(b) Courtesy and copyright © 2006 Eddy Xuan.

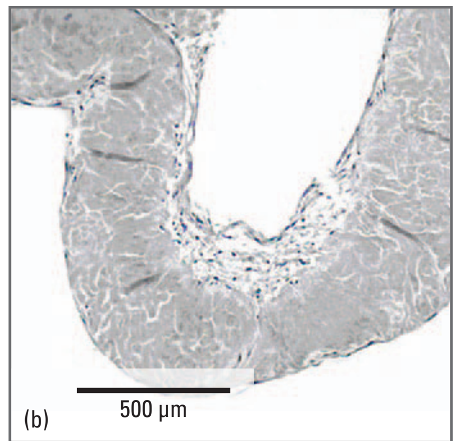
You will use 0.1 mm (100 µm)—a thickness one-fifth that of modern, surgically implanted dermal substitutes (Figures 17.03a and b). Starting with a relatively small scaffold will allow you to see results quickly. When the model is working to your satisfaction, you can easily change a few parameters and output larger scaffolds.

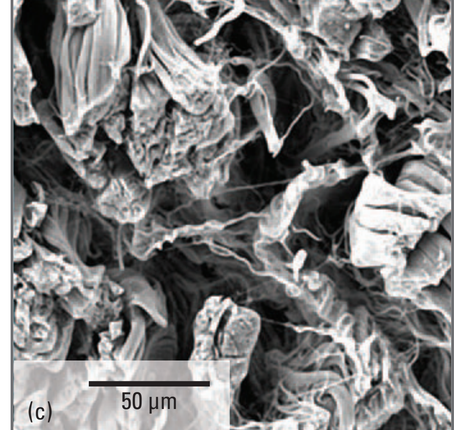
Dermal microstructure is characterized by discrete collagen fiber bundles (Figure 17.03c) whose average diameter, cross-sectional shape, and directional orientation vary in relation to their depth from the skin surface. Collagen fiber bundles are generally arranged parallel to the plane of the skin surface but their orientation about the other two major axes is random, apparently to manage tensile forces on the skin from multiple directions. This random orientation stands in stark contrast to the arrangement of collagen in another tissue, tendon, in which fibers are aligned for maximum tensile strength in (most often) one direction. Collagen bundles in the dermis cross-link with one another, forming complex meshworks and presenting cells with an array of opportunities for, and obstacles to, migration.

Five characteristics of this complex 3D pattern provide a material starting point. They are listed in Table 17.01. By treating them as variables when you construct the modeling algorithm, you can subsequently adjust them and test their downstream effect on the migration behavior of your simulated cell population (next chapter!). The macroscopic dimensions of the dermis are also parameters you'll incorporate into the model; unlike living skin which is continuous over the entire body, your Maya model will capture a small, cubic sample of tissue matrix. By building the dimensions into the model as variables, you can create scaffolds of different sizes and length/width/height ratios.

An advantage to creating the scaffold model in Maya geometry—NURBS curves and surfaces—is that later you can build in the physical matrix properties in the form







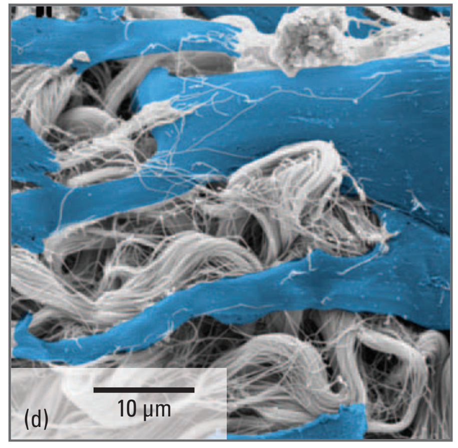


FIGURE 17.03

The collagen component of the dermis (Ref. 2).

(a) Photograph of a dermal graft similar to those used in surgical procedures. Here it's shown being held for the camera by a surgeon's gloved hands. The tissue was rendered white by chemical processing that removes cells debris that may cause an adverse immune response in the patient receiving the graft.

(b) Light micrograph of the dermal layer of the skin. This slide was made from a sample similar to that shown in (a).

(c) SEM of the cut edge of a dermis sample similar to that shown in (b). Note the appearance of distinct collagen fiber bundles.

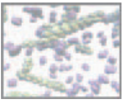
(d) High-magnification SEM showing fibroblast cells seeded on the surface of a dermal graft sample similar to that shown in (a). The fibroblasts have been artificially colored blue for easy identification.

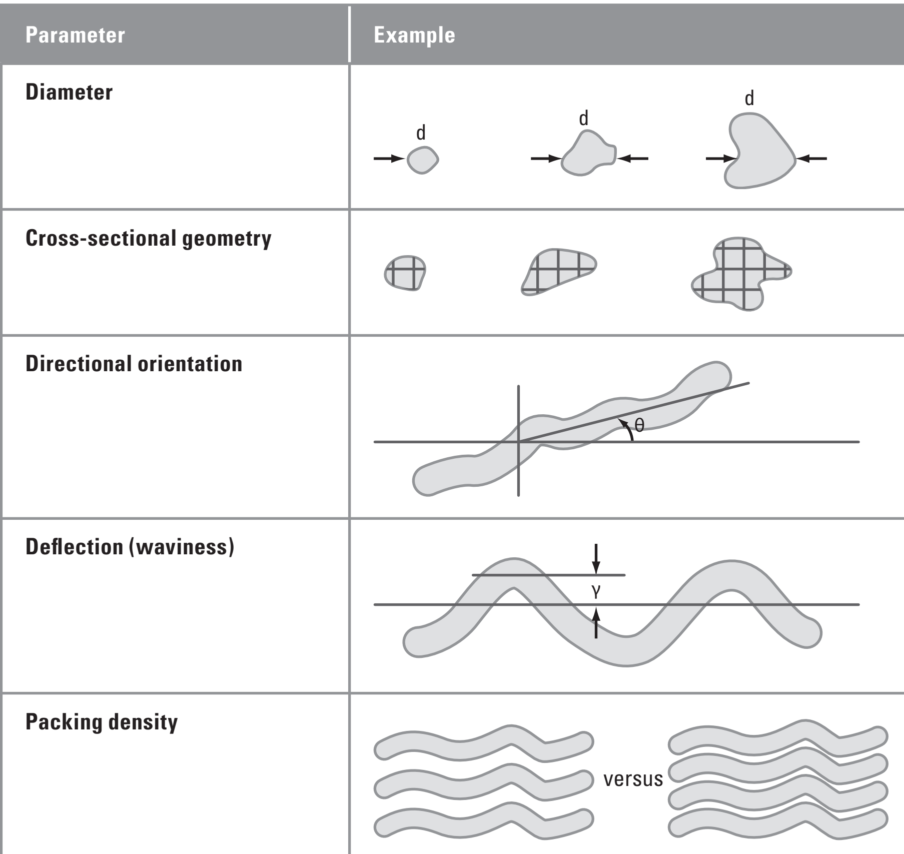
All images courtesy and copyright © 2006 Alexis Armour, MD, University of Toronto. Used with permission.

of deformations and custom attributes, and perhaps even dynamics, if you choose to develop this model further.

Model definition

The goal of this project is to make a geometric model that embodies key characteristics of a dermal collagen scaffold, can be generated in relevant clinical dimensions[2], and can be used subsequently as a substrate in an in silico model of cell migration. The model will involve dozens of individual fibers. We will explain shortly how we estimate the number of fibers in a given scaffold volume. Building these using standard Maya UI tools—that is, making and shaping individual wavy collagen fibers while minimizing their interpenetrations with one another—is a highly impractical task. You want automatic control, through Maya, over the scaffold's structure via parameters like fiber deflection (waviness) and packing density. Therefore, in addition to satisfying the scaffold design parameters, you are also faced with the challenge of getting such a complex tangle of geometry build on the fly in Maya.





Parameter	Example
Diameter	
Cross-sectional geometry	
Directional orientation	
Deflection (waviness)	
Packing density	

TABLE 17.01
Scaffold parameters.

Scaffold dimensions

The scaffold shape will be a rectangular prism 100 μm deep (or thick) × 100 μm wide × 150 μm long. The effective volume for the cell migration model in the next chapter is 100 μm cubed. You're making the scaffold longer than it is deep or wide to accommodate boundary conditions for the migrating cells; they tend to move more freely lengthwise along the fibers than width- or depth-wise between parallel-running fibers.

Fiber size, packing density, and shape

Table 17.02 gives you an idea of how fiber sizes are distributed throughout the dermis. Let's call this the **fiber size distribution**. A statistical histogram of the size distribution is shown in Figure 17.04. If the long axes of fibers are aligned on average with a major axis (the Z-axis in this model)—their meandering paths will provide the overall multidirectional orientation that characterizes dermal collagen—you can estimate the appropriate number of fibers for a given volume from the number that intersect the plane perpendicular to this major axis. We tested a number of different densities in Maya and arrived at a range of values that were both practical time-wise for Maya

	Fiber diameter bins				
	1	2	3	4	5
Diameter (μm)	**3 and under**	**6**	**9**	**12**	**18 and over**
Size distribution (%)	**30**	**29**	**18**	**8**	**15**

TABLE 17.02

Fiber sizes and size distribution from measurements of published SEM of dermal collagen.

From electron micrograph measurements made by the authors and reported in reference 3.

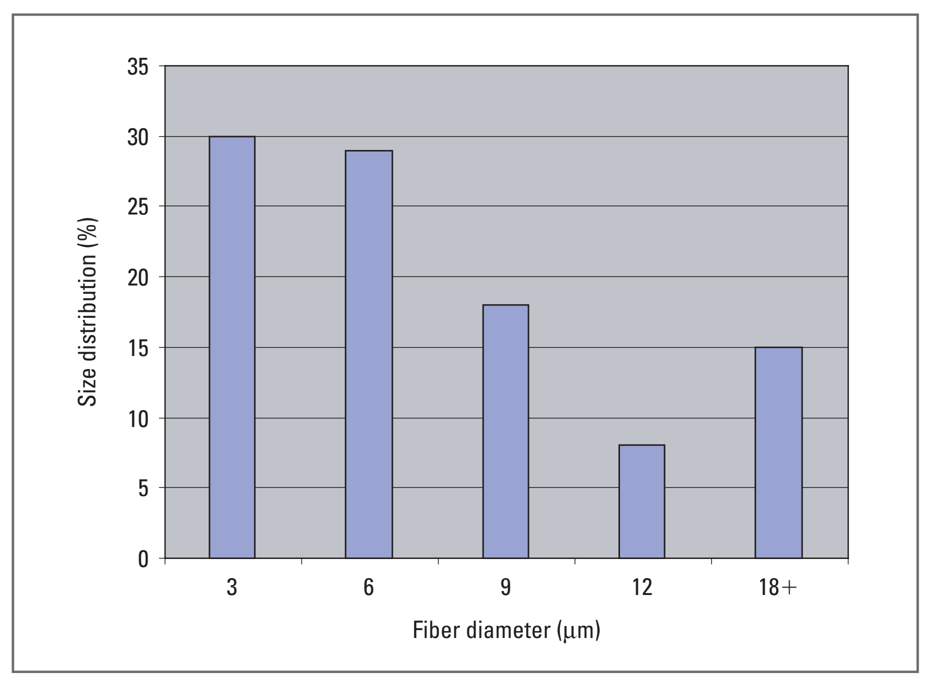


FIGURE 17.04

Size distribution of collagen fiber bundles measured from published scanning electron micrographs[3].

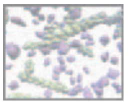
and functioned well in the cell migration project. In a $(100\ \mu m)^3$ cube these density values equate to 20, 40, and 60 fibers. In this project, you'll fill the scaffold with 40 fibers.

Forty fibers accord nicely with the range of 750–1500 fibers reported for real lab specimens[3].

The model fiber circular fiber cross-sections let us simplify the process of scaffold generation. As you saw back in *Chapter 05*, you can make fiber models in Maya easily, by extruding a cross-sectional shape along a spline curve (the fiber axis). With this approach, you may later choose to randomize the cross-sectional shape of the fibers to reflect the irregular shapes seen in cut views of dermal collagen.

Fiber orientation and intersections

By deflecting each fiber along its length, you can capture in the model the characteristic waviness and random orientation of dermal collagen seen in scanning electron micrograph (SEM) data. As well, while dermal fiber bundles do split apart and also form cross-links with one another, they do not interpenetrate. It follows that your Maya fibers must also avoid interpenetrations—a considerable challenge given their wavy paths and tight packing!



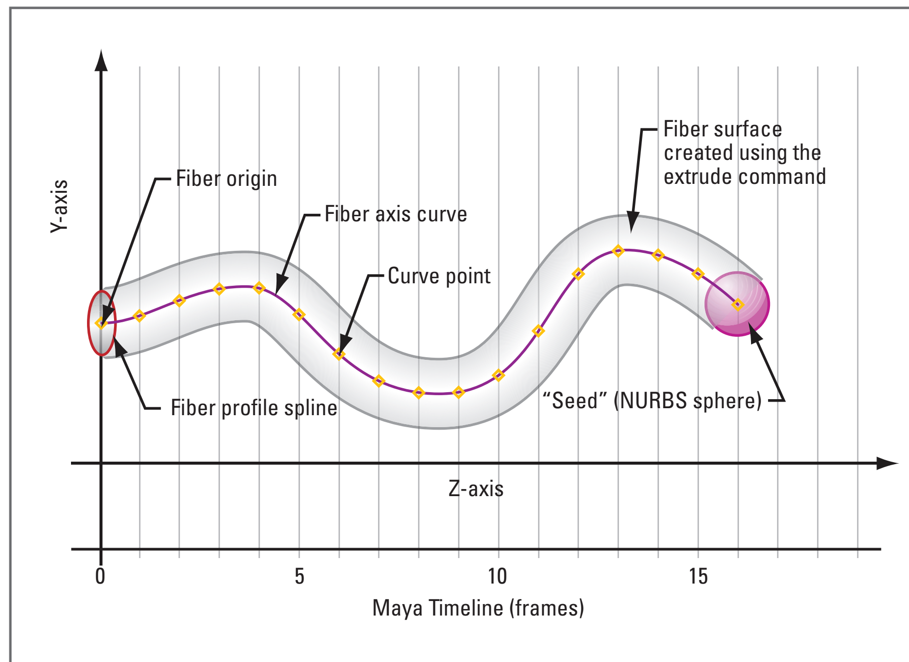


FIGURE 17.05

The elements of a single fiber model. Curve deflections are shown for the Y-axis only, but also occur along the X-axis.

Methods: **Algorithm design**

Now that we've covered the essential morphometric parameters of the model, it's time to design an algorithm to build ECM scaffolds procedurally—that is, using MEL and Maya's built-in functions to make the model under algorithmic control. Here then is the mission statement:

> **Take a number of input parameters (scaffold dimensions, fiber size distribution, etc.) and build a scaffold of randomly oriented cylindrical surfaces.**

Let's take a close look at the Maya elements that will make up the model and discuss a strategy for equipping the fiber orientation with the appropriate statistical properties. Figure 17.05 shows the different fiber model elements that are discussed below.

The fiber axis

In *Chapter 05* you created a fiber model—an extruded tube—by first making a spline curve for the fiber axis using the `CV Curve` tool, then making a circular profile spline, and finally extruding the circle along the axis using the `Extrude` tool. In this project, you will use the same approach but, rather than placing the axis curve points manually, you'll do so using the `curve` command from within an expression. The profile circle will of course have the diameter of the fiber (or tube) it creates.

NURBS spheres called seeds

Here we'll introduce an abstraction that helps in the design and visualization of the modeling process. You'll use an object—which happens to be a NURBS sphere,

but doesn't have to be—to represent the leading point of every fiber axis. Rules of motion—in the form of a random walk, collision avoidance, and adherence to scaffold boundaries—govern the motion of these objects. As each object moves, it lays down a spline curve in its path, which in the end is a fiber axis. We call these objects *seeds* in reference to their role in *growing* fibers. The diameter of each seed is the same as the diameter of the fiber it represents.

The next four sections outline the modeling strategy. As you read them, trace along the flowchart in Figure 17.11 to see how the different pieces of this in silico procedure relate to one another.

Modeling with the timeline

In the previous chapter, you used an animation expression to animate the crawling cell (remember: an animation expression is a script that Maya typically executes each time the frame number changes on the timeline). In this project you'll use an expression actually to build the 3D scaffold model. Called `moveSeeds`, this expression repositions each seed according a set of rules, which we'll discuss shortly, and adds a new CV point to the corresponding fiber axis curve (Figure 17.04). As `moveSeeds` builds the axes, the frame number is used as an incrementing variable, so that the model evolves as the scene plays back. Using an expression to build your model, instead of a MEL script that executes entirely at a single frame, allows you to watch the model *grow* before your eyes. If you wish, you can even make a rough playblast animation to keep a visual record of the scaffold fibers being created.

A rule-based design

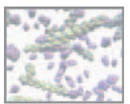
This project uses a rule-based design in order to move the seeds that determine the fiber paths. There are three rules, each parceled in a procedure called `rule1()`, `rule2()`, and `rule3()`, respectively. They are shown schematically in Figure 17.06. Each procedure is called from the `moveSeeds` expression at every frame during playback and for every seed in your scene. Each rule returns a vector: `$v1`, `$v2`, and `$v3`. These vectors are added together and then added to the current position of the seed in order to move it (Figure 17.07). By parceling rules of motion into separate procedures instead of putting them all in one you can easily turn off individual rules or add new ones in a modular fashion.

Rule-based strategies are common in agent-oriented programming (AOP) approaches to in silico biology.

`rule1()` provides the random element to the seed motion using a biased random walk algorithm—a variation on the random motility generated for the crawling cell in the previous chapter—which is largely responsible for the meandering nature of the fiber paths. `rule2()` uses collision avoidance to keep seeds (and by extension, the axes) from approaching one another too closely in order to minimize fiber interpenetrations. By issuing standard MEL commands that query Maya's internal geometry engine, you can check for collisions very concisely, without having to formulate large amounts of your own custom code. Finally, `rule3()` constrains the seeds within the prescribed scaffold boundaries, which we'll call the bounding box. Let's take a quick look at the methods used in each rule.

rule1(): The random walk

To generate the meandering fiber paths, you'll employ a basic random walk algorithm. It uses Maya's random number generator (the rand() MEL function) to select one of



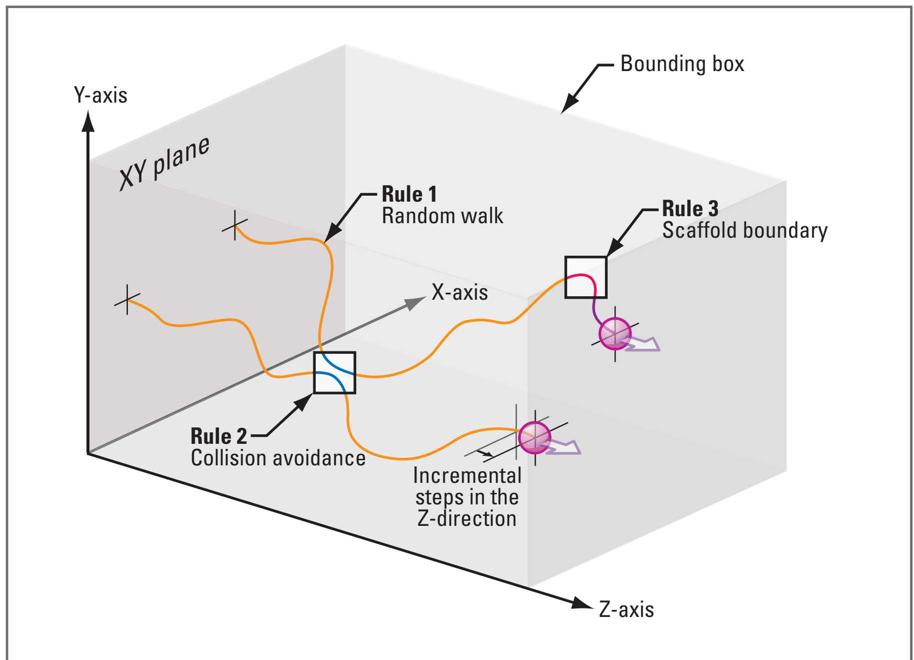


FIGURE 17.06

Three rules are responsible for the meandering fiber paths, collision avoidance, and keeping the fibers within the prescribed scaffold volume.

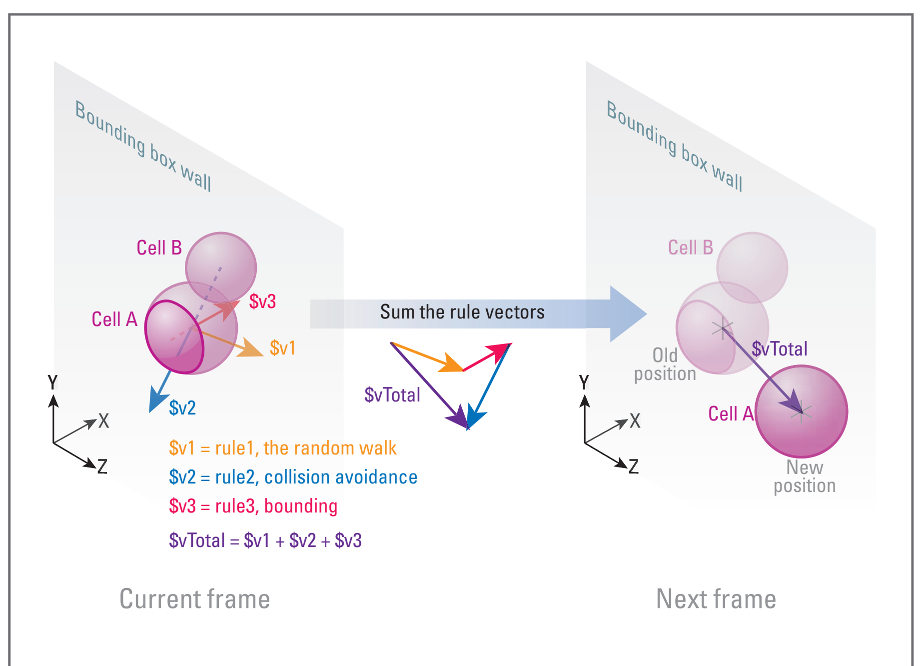


FIGURE 17.07

The vectors returned by rule1(), rule2(), and rule3() are added together to produce the stepwise displacement $vTotal for each seed.

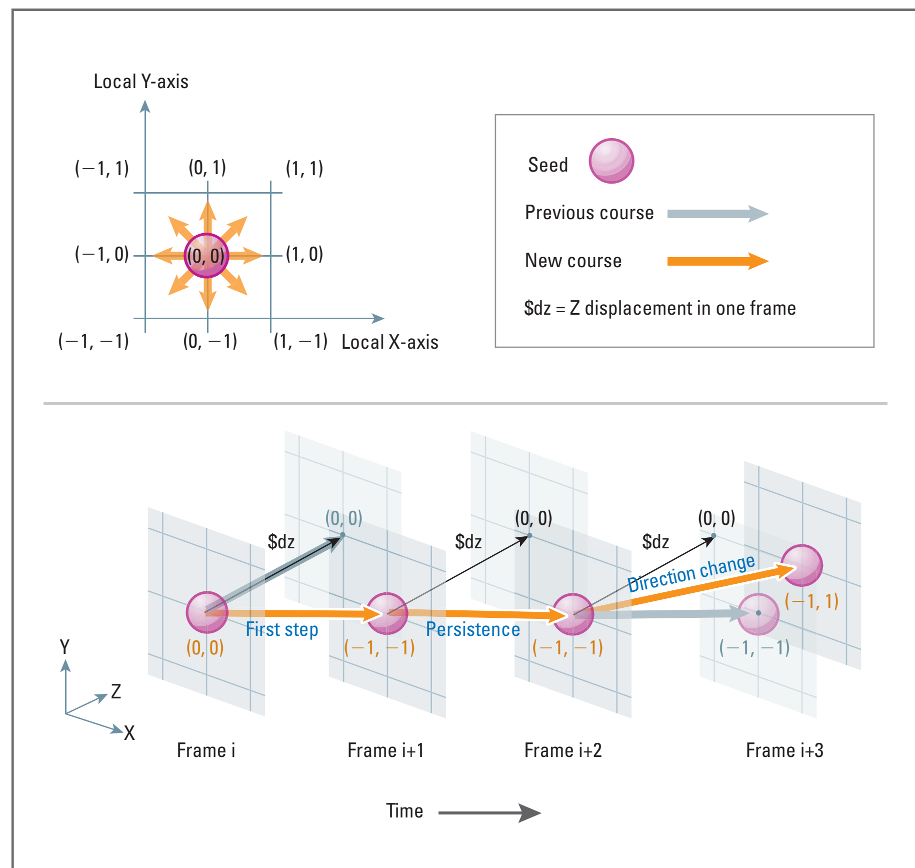


FIGURE 17.08

In this random walk algorithm, a seed selects one of nine possible vectors (including (0, 0)) in the XY plane relative to its local (versus global) position. dz is the seed's displacement in the Z-direction. If the seed is persisting, it moves in the same direction it did in the previous frame.

nine possible vectors (dx, dy) relative to a seed's current position (see Figure 17.08). We'll call the incremental Z-distance traveled by each seed, dz. Together, dx, dy, and dz make the 3D random walk vector (dx, dy, dz). At each time step (one frame of scene playback) the seed may either continue along its current course or change direction, depending on a parameter we call *persistence*. Persistence here means the same thing it did in the last chapter when it was applied to cell migration: the tendency to continue on the present course. When a seed ceases to persist, it selects a new direction and a new persistence value. It then persists in the new direction for the prescribed time until once again it selects a new direction and new persistence value. This process is shown schematically in Figure 17.08.

So that you can take a straightforward approach to collision avoidance, all seeds will advance uniformly in the Z-direction, resulting in a *biased* random walk.

Non-uniform movement in the Z-direction would certainly be desirable in some tissue design applications. This would be achieved by varying the incremental step in Z on a per-seed basis, and would require a modified approach to collision detection and avoidance. This is a good follow-up project!

rule2(): Collision avoidance

rule2() evaluates the separation between a given seed (seed A in Figure 17.09) and every other seed in the scene (represented by seeds B and C in Figure 17.09). If another seed lies within a critical distance of seed A then an avoidance vector is calculated and

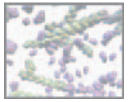


FIGURE 17.09

rule2() returns a vector, $avoid, that is the sum of all avoidance vectors between the current seed (seed A in the illustration) and all other seeds in the scene (B and C). The procedure then performs the same calculations on each of seed B and C. The net result is a tendency for all seeds to avoid one another, which in turn minimizes the intersections between fiber surfaces once the scaffold is built. The names of vectors and scalars in this illustration (e.g. $surfaceSep) are the ones you'll use when composing your MEL code. The vector math functions for calculating magnitude and unit vectors are denoted here with the MEL commands mag and unit, respectively.

added to a master avoidance vector called `$avoid`, which is the return value of `rule2()` and determines the "avoidance" component of seed A's motion.

The method used here is a simple approach to collision avoidance but is not flawless—the resulting scaffolds contain some interpenetrating fibers. Collision avoidance in crowded environments is by no means a trivial problem. More robust (and more complex) solutions exist and we encourage you to explore the topic further if your in silico modeling projects require absolute avoidance of collision between objects.

rule3(): Bounding box

`rule3()` checks each seed to see if it lies outside of the bounding box. If so, the procedure returns a vector called `$bound` to the `moveSeeds` expression which is used to nudge the

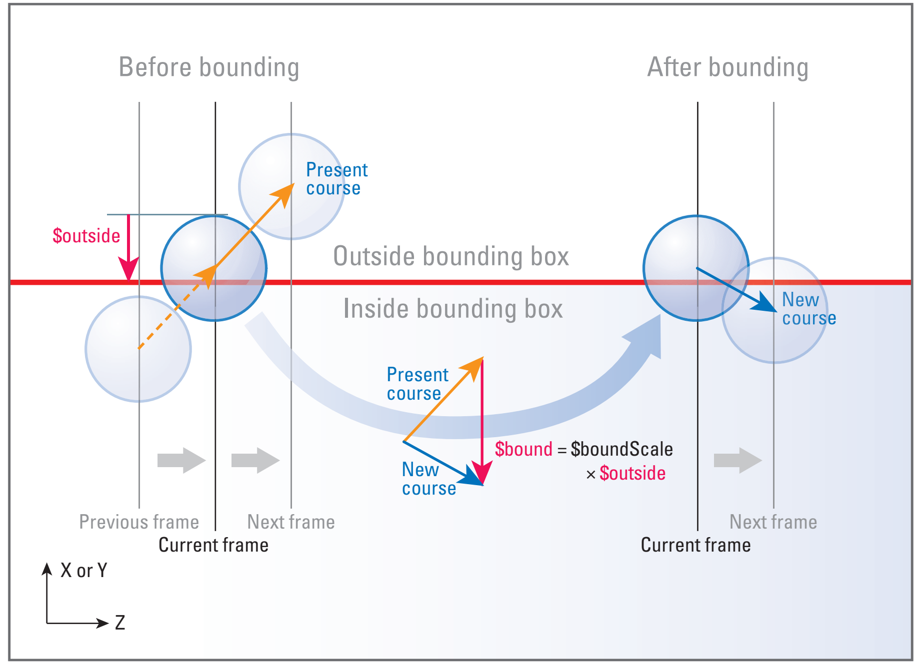


FIGURE 17.10

A vector called $bound is used to nudge errant seeds back into the bounding box of the scaffold. $bound varies exponentially with distance of the seed outside the box in order to smooth the motion.

seed in question back into the bounding box within the current frame. The strength of the "nudge" varies exponentially with the distance of the seed outside the box, in order to smooth the motion. Figure 17.10 shows this rule in schematic form.

Randomizing the fibers at the start

The first thing you'll do before building the expressions and the rule procedures is code a procedure called `makeSeeds()` that creates the seeds (one for every fiber) and positions them randomly in the XY plane at Z = 0. In the beginning, the seeds mark the starting points of the fiber axes. Their random placement will likely result in overlapping of seeds. Therefore, the `moveSeeds` expression will spend the first several frames of playback calling rules 1 and 2, to space the seeds out while keeping them within the bounding box. We dubbed this process *untangling* and found through trial and error that, for the packing densities you're interested in with this project, the seeds untangle by frame 20. Beginning at frame 20 `moveSeeds` calls all three rules to move the seeds and grow the scaffold.

Resetting the seeds

After building a whole or partial scaffold, you may want to scrap it and start again. You may also want to build a new model using different parameters—without having to create and untangle a new batch of seeds. To accommodate these situations, you'll write a short expression call `resetSeeds` that executes only at frame 1—that is, when you rewind the playhead (current time indicator) to the start of the playback range.

The fiber surfaces

The fiber axes of your scaffold are complete once the seeds have traveled the length you specify up front: 150 µm. At this point playback is halted from within the `moveSeeds` expression and the motion rule procedures are no longer called. The next step then is to make and position a profile spline—a circle of the appropriate diameter—at the first CV of every curve. The `extrude` command is then called to make the fiber surfaces.

To your migrating cells in the next chapter, each NURBS tube represents the adhesion contact surface of a collagen fiber bundle. Subsequently, any point on a fiber surface can be queried for its position in 3D worldspace (X,Y,Z coordinates) and therefore used to position a cell. In addition to the NURBS tube representing the fiber surface (which can be used to locate cells while providing a visual representation of this contact surface) you can also query Maya for the 3D worldspace position of a point on the axis curve and use it to position a cell. This saves a lot of coding. The bottom line is that within each axis curve and its fiber surface lies all the spatial information needed to position and move cells within the scaffold.

Algorithm summary

To make things crystal clear before building the algorithm, let's sum up the script elements (procedures and expressions) and the procedural modeling steps.

1. **makeSeeds()** procedure
   (a) Create NURBS spheres called seeds.
   (b) Position the seeds randomly in a 100 x 100 µm plane parallel to the XY plane at Z = 0.
2. **moveSeeds** expression
   (a) For frames 2 to 20, untangle the seeds using rule1() and keep them bound with rule3().
   (b) At frame 20, start a curve for every fiber axis, at the position of the corresponding seed.
   (c) After frame 20, call rule1(), rule2(), and rule3() to move the seeds as the scene plays.
   (d) Add a CV to each fiber axis at the position of the corresponding seed.
3. **rule1( )** procedure

   Calculate the random walk component of seed motion, biased in the positive Z-direction.
4. **rule2( )** procedure

   Calculate the collision avoidance component of seed motion.
5. **rule3( )** procedure

   Calculate the bounding box component of seed motion.
6. **resetSeeds** expression

   Reset the seeds to their untangled starting positions when the playhead is rewound to frame 1.

The flowchart in Figure 17.11 shows how the procedures and expressions link together. In the next section you will build them in the order we've listed above.

Start

frame = 1 ?

no

yes

resetSeeds expression
- Initialize global variables
- Return seeds to initial positions
- Delete existing fiber construction elements: curves, circles, and surfaces

makeSeeds procedure
user inputs:
- Cube size
- Fiber number
- % of total fibers allocated to each diameter bin

moveSeeds expression
- Call the rule procedures
- Update seed positions
- At frame 20, create a curve for every seed, placing its first point at the center of its seed
- After frame 20, append a point to each curve at the center of its seed

rule 1
Random walk in XY plane; uniform step in Z-direction

rule 2
Avoid collisions with other seeds

rule 3
Keep seeds within the scaffold bounding box

Has seed traveled 150 µm ?

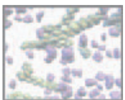
no

yes

End

FIGURE 17.11

Flowchart for the scaffold algorithm.



Methods: Encoding the algorithm

Unlike the previous project, in which the cellCrawl algorithm operated on a prebuilt geometry model, this project requires no initial file setup. Everything to grow the fiber scaffold model in silico is contained in the scripts you're about to build. We recommend composing them as you did the scripts in previous chapters: in an external text editor, with each element (procedure or expression) saved in a separate text file within your Maya Scripts directory on your hard drive. When your code's done, you can simply load the elements into a new Maya scene and make the scaffold. You may wish to have Maya open as you follow along with the material below so that you can test out certain commands or bits of code. You may also wish to test each script in its entirety before moving on to the next one. This will save you from debugging all five scripts at once at the end of this chapter.

If you want to begin growing scaffolds right away, you can open and play the ready-made Maya scene file entitled `scaffold.ma` which is located on the CD-ROM. This scene contains the resetSeeds and moveSeeds expressions and a group of seeds ready to go. To grow the scaffold, you'll need to source the three rule procedures which are also on the CD-ROM.

17_ECM_Scaffold/scenes/scaffold.ma (Maya scene file)

17_ECM_Scaffold/MEL/rule1.mel
/rule2.mel
/rule3.mel

The makeSeeds() procedure

To save space we've foregone the usual "author, date, etc." header information.

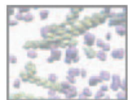
```
/* Description:
Run this procedure first when making a fiber scaffold. It creates
NURBS spheres called seeds, which are used in a subsequent
expression called moveSeeds that creates the fiber axes.

The procedure arguments are as follows:
$cubeSize            The length of one side of the scaffold cube.
$seedNum             The number of seeds (and therefore fibers) to
                     be made.
$three–$twentyfour   The percentages of each of the fiber diameters.
*/

global proc makeSeeds(float $cubeSize, int $seedNum, float $three,
  float $six, float $nine, float $twelve, float $eighteen) {

    /***** DECLARE THE VARIABLES. *****/

    /*
    $values[]        An array of initial values for the widget
                     attributes.
    $bins[]          An array of diameter bin sizes.
    $binSize         The number of seeds in the current bin.
```

```
$diameter[]      The six fiber diameters, one value for each bin.
$dia             The diameter of a given seed.
$radius          The radius of a given seed.
$x, $y, $z       Position the seeds randomly in the XY plane at
                 Z = 0.
*/
float $values[], $bins[], $binSize, $diameter[], $dia, $radius,
                 $x, $y, $z;

/*
$attributes[]    Custom attributes to be added to the widget.
$name            The name of the current seed.
*/
string $attributes[], $name;

/*
$arraySize       The size of $attributes[];
$i and $j        Counters.
*/
int $i, $j;
```

Next you'll initialize the variables, make a locator object called `widget`, and assign it custom attributes. These attributes will store parameters that affect scaffold design. They will be queried in the moveSeeds expression and the three rule procedures. Unlike a variable value which gets erased when Maya is closed, an attribute value gets written into the scene file. Therefore the parameter values will be ready and waiting. That way you won't have to enter them by hand each time you restart Maya and open an existing scaffold scene file. The initial attribute values are specified in `$values`. These are the numbers we recommend you start with. You can change them at any time by selecting the widget and entering new values in the Channel Box.

```
/***** INITIALIZE THE VARIABLES. *****/

$attributes = {"cubeSize", "maxSpan", "minSpan", "dx", "dy", "dz",
  "persistMax", "persistMin", "sizeBias", "spacingMin",
  "avoidScale", "boundScale"};
$values = {$cubeSize, 40, 20, 1, 1, 1, 10, 0, 0.05, 2, 2, 1};
$arraySize = size($attributes);
$bins = {$three, $six, $nine, $twelve, $eighteen};
$diameter = {3, 6, 9, 12, 18};

/***** MAIN BODY *****/

// Make a locator called widget to store scaffold design
  parameters.
spaceLocator -p 0 0 0 -name "widget";

for ($i = 0; $i < $arraySize; $i++) {

    // Add and set values for the custom attributes.
    addAttr -ln $attributes[$i] -dv 0 widget;
    string $tmpStr = "widget." + $attributes[$i];
    setAttr -e -keyable true $tmpStr;
    setAttr $tmpStr $values[$i];

}
```

The name **widget** is arbitrary. It doesn't refer to a specific Maya construct.



Make the seeds

Next you'll make two for loops, one nested inside the other, to create the seeds. The outer loop cycles through the seed size (i.e. fiber diameter) distribution variables, $three, $six, and so on. The inner loop makes and positions the seeds for each size. The rand() command is used to generate random X and Y position values that lie within the scaffold boundaries.

```
// Make the seeds. $i counts the array index in $diameter;
for ($i = 0; $i < 5; $i++) {

    $binSize = (float) $bins[$i]/100 * $seedNum;

    for ($j = 0; $j <= $binSize ; $j++) {

        $dia = $diameter[$i];
        $radius = $dia/2;

        // Set random start positions
        $x = rand((0 + $radius), ($cubeSize - $radius));
        $y = rand((0 + $radius), ($cubeSize - $radius));
        $z = 0;

        // Create a name for the current seed.
        $name = ("seed" + $dia + "_" + $j);
```

Create the first seed of each size (diameter) bin (see Table 17.02) using the sphere command. The radius is set to 0.5, giving a starting diameter of 1. The seed is then scaled to the appropriate fiber diameter (3, 9, etc.). Scale is a compound attribute, composed of scaleX, scaleY, and scaleZ, therefore, it must be set using the -type double3 flag with the setAttr command. Subsequent seeds of the same size can then be duplicated from the first. You'll ensure only one "first" seed is created, and the rest duplicated from it, by checking if the first exists with an "if not objExists" expression below (where "not" is expressed by the character ! in MEL). If the sphere does not exist, the expression returns 1 and the code in the curly brackets executes. On the other hand, if the sphere does exist, the expression returns 0, and the first seed is duplicated.

```
        // Make and position the sphere.
        if (!`objExists seed3_0`) {

            // Create a new NURBS sphere.
            sphere -r 0.5 -n $name;
```

With this next bit of code, you'll add a custom compound attribute called posInit to the seed's transform node and use it to store the seed's initial position. This attribute's values will be used in the resetSeeds expression to return the seeds to their starting points when the frame number is reset to 1.

The **parent** flag in the addAttr command indicates the attribute that is to be the new attribute's parent; in this case: **posInit**.

```
            // Add custom attributes.
            addAttr -longName posInit -attributeType double3
              $name;
            addAttr -ln posInitX -at double -parent posInit
              $name;
            addAttr -ln posInitY -at double -p posInit $name;
            addAttr -ln posInitZ -at double -p posInit $name;

            // Make the attributes keyable.
            setAttr -e -keyable true ($name + ".posInit");
            setAttr -e -keyable true ($name + ".posInitX");
            setAttr -e -keyable true ($name + ".posInitY");
```

```
            setAttr -e -keyable true ($name + ".posInitZ");
        }
        else duplicate -n $name seed3_0;

        // Scale the sphere to the proper diameter.
        setAttr ($name + ".scale") -type double3 $dia $dia $dia;

        // Set the custom vector attribute values.
        setAttr -type double3 ($name + ".posInit") $x $y $z;

        // Move the sphere to $x $y $z.
        move $x $y $z $name;
    }
}
```

Conclude the procedure

In this last bit of code, you'll group the seeds (to keep the Outliner window manageable) and then send a message to the Maya user (most likely yourself) stating that the procedure is complete. Next, you'll set the current time indicator to frame 1 in order to force the resetSeeds expression to evaluate and thereby initialize key global variables. Finally, close the procedure using a curly bracket. It is often helpful to print a sample procedure call to the Script Editor and Command Line. You can cut, paste, and execute the sample code after loading the procedure.

```
        // Group the seeds.
        group -n "groupSeeds" `ls -tr "seed*"`;

        // Print user feedback in the Script editor.
        print "\nThe seeds are ready to go!";

        // Set the current time to 1.
        currentTime 1;

    } // End makeSeeds procedure.

    // Print user instructions in the Script editor.
    print "Enter: makeSeeds(100, 40, 30, 29, 18, 8, 15)";
```

That's all for this procedure. You can enter and run it now if you like, or wait until you've composed the remaining scripts. The next script is the `resetSeeds` expression.

The resetSeeds expression

After creating a group of seeds, you may want to make several scaffolds with it, testing the effects of different random walk parameters. This expression initializes the global variables and snaps the seeds back to their starting positions whenever the current time changes to frame 1 from another frame or when you press the Edit button in the Expression Editor at frame 1. It is also in this expression that you will set the random walk parameters. The random walk proper will be coded in the `rule1()` procedure, and called from the animation expression named `moveSeeds`. You'll build `moveSeeds` in the next section.

For clarity we'll present this expression without the escape line break notation, `/n`, that would allow you to enter it using the `expression` command. As mentioned in *Chapter 12*, we find all but the simplest one-line expressions easier to manage when composed with line breaks, and then copied and pasted directly into the Expression Editor, without regard for the restrictions imposed by the `expression` command.

Start the script with a test of the frame number. Since this expression needs to execute only at frame 1, open with a conditional statement that prevents Maya from wasting time reading variable declarations at other frames. The `currentTime` command is used to set or, in this case, query the current time in the timeline.

Remember that global variables must be declared within each script that uses them.

```
/* Description:
This expression resets spheres called "seeds" to starting positions
and declares key variables. These variables are used in the rule
procedures to calculate the motion of seeds, and therefore the
paths of scaffold fibers. This expression executes only at frame 1.
*/
if (`currentTime -query` == 1)
{
    /***** DECLARE THE VARIABLES. *****/
    /*
    $seedPos[]        A string array of seed positions.
    $v1_old[]         This stores the motion rule vectors and is used
                      for persistence.
    */
    global vector $seedPos[], $v1_old[];
    /*
    $seedNames[]   A list of seed names.
    */
    global string $seedNames[];
    /*
    $cubeSize         The X and Y dimensions of the scaffold.
    $dx, $dy, $dz     Incremental displacements used in rule1().
    $length[]         Tracks the length of each fiber.
    */
    global float $cubeSize, $dx, $dy, $dz, $length[];
    /*
    $end              1 if seeds have reached the scaffold end,
                      0 if not.
    $seedCount        The number of seeds in the scene.
    $persistence[]    An array of persistence times for all seeds.
    $persistMin       The minimum persistence time which is stored
                      in the widget's persistMin attribute.
    $persistMax       The maximum persistence time.
    */
    global int $end, $seedCount, $persistence[], $persistMin,
      $persistMax;
    /*
    $trans            The translate attribute values for the current
                      seed.
    */
    vector $trans;
    /*
    $name   The name of the current seed.
    */
```

```
string $name;
int $i;
```

Initialize the variables

Here you'll clear the global array variables and set the random walk displacement variables, $dx, $dy, and $dz to good starting values (we found them by experimenting with the procedure). When you begin making scaffold models, try varying $dx, $dy, and $dz yourself, and study their effect on scaffold architecture.

```
/***** INITIALIZE THE VARIABLES. *****/

clear ($seedPos);
clear ($v1_old);
$seedNames = `ls -transforms "seed*"`;
$seedCount = size($seedNames);
clear ($persistence);
clear ($length);
$end = 0;
$cubeSize = `getAttr widget.cubeSize`;
$dx = `getAttr widget.dx`;
$dy = `getAttr widget.dy`;
$dz = `getAttr widget.dz`;
$persistMin = `getAttr widget.persistMin`;
$persistMax = `getAttr widget.persistMax`;
$i = 0;
// Set playback to loop only once.
playbackOptions -l "once";
```

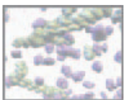
Reset seeds to their initial positions

This next bit of code resets every seed to its starting position at Z = 0 using the posInit attribute of each seed's transform node. This vector attribute is in the form << x, y, z >> and cannot be used directly in the setAttr command. An intermediate step is required: assign posInit to a temporary vector, then use the latter to set the attribute.

```
/***** MAIN BODY *****/

// Set seed positions equal to their initial positions.
for ($name in $seedNames)
{

    // Query the initial seed position and move it there.
    $trans = `getAttr ($name + ".posInit")`;
    setAttr ($name + ".translate") ($trans.x) ($trans.y)
      ($trans.z);
```

Now store the initial seed positions in the variable $seedPos[] for use in the moveSeeds expression.

```
    // Assign the position vector to the global variable
      $seedPos[].
    $seedPos[$i] = <<$trans.x, $trans.y, $trans.z>>;

    // At frame 1 $oldPosition equals $seedPos.
    $oldPosition[$i] = $seedPos[$i];

    // increment the counter.
    $i++;
```




```
        } // End seed name loop.

        // Provide user feedback.
        print "resetSeeds expression has finished.";
```

The next four lines are optional. They delete any existing fiber axis curves, profile splines (circles), and fiber surfaces. This is handy when you're testing scaffold parameters (those stored in the custom widget attributes) but can lead you to accidentally delete a scaffold you wanted to keep, just by returning the seen to frame 1 (which executes this expression). When you want to make scaffolds to keep, comment these lines out or delete them.

```
        // Delete fiber elements on rewind.
        if (`objExists groupFibers`) delete groupFibers;
        if (`objExists fiberAxis3_0`) delete `ls -tr "fiberAxis*"`;
        if (`objExists nurbsCircle1`) delete `ls -tr "nurbsCircle*"`;
        if (`objExists fiberSurface3_0`) delete `ls -tr "fiberSurface*"`;

        // Select the widget so the scaffold parameters appear in the
          channel box. select widget;

    } // End resetSeeds expression.
```

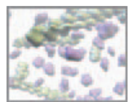
The moveSeeds expression

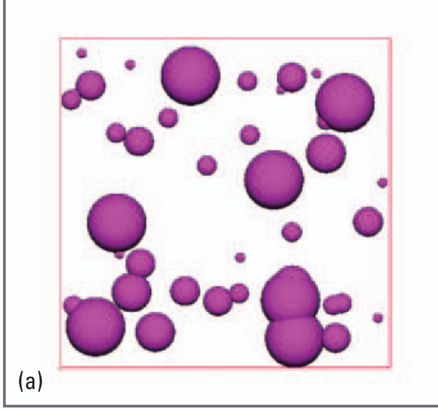
This expression calls the procedures, `rule1`, `rule2`, and `rule3`, in order to update the position of each seed. Each procedure requires as an argument the current seed number, which is stored in the variable `$i`. Each procedure then returns a vector called `$v1`, `$v2`, and `$v3`, respectively. These vectors are added to the current position vector, stored in `$seedPos[$i]`, which is in turn used to set the translate X, Y, and Z attributes for `seed[$i]`. Once the seed has been moved, a new curve point is added to the corresponding fiber axis curve.

Since this expression is intended to be read after, but not at, frame 1, let's start with a conditional statement similar to that used in `resetSeeds`.

```
    /* Description:
    This expression moves spheres called "seeds" one step in the Z-
    direction each time the frame increments, while building splines
    that follow the seed paths.

    The rules of motion are contained in three procedures:
    rule1() Returns $v1, a biased random walk vector.
    rule2() Returns $v2, a collision avoidance vector.
    rule3() Returns $v3, a bounding vector.
    */

    if (`currentTime -query` > 1)
    {

        /***** DECLARE THE VARIABLES. *****/

        global vector $seedPos[], $v1_old[];
        global float $length[], $dx, $dy, $dz, $cubeSize;
        int $i;

        /*
        $curveNames[]  A list of fiber axis curves.
```

```
*/
global string $curveNames[], $seedNames[];

/*
$frameCheck   Used to send the current frame number to the rule
              procedures. A global variable must be used since
              the Maya variable "frame" cannot be queried outside
              of its scope: this expression.
*/
global int $frameCheck, $seedCount, $end;

/*
$seed         The current seed.
$index[]      Stores the return strings from the tokenize command.
$curve        The current fiber axis (NURBS curve).
$circle       The current fiber profile (NURBS circle).
$surface      The current fiber surface (extruded NURBS tube).
$cpos         The closestPointOnSurface node.
$posi         The pointOnSurfaceInfo node.
$fiberGroup   A group containing $curve, $circle, and $surface.
*/
string $seed, $index[], $curve, $circle, $surface, $cpos, $posi;
string $fiberGroup;

/*
$v1           The return vectors from rule1() procedure.
$v2           The return vectors from rule2() procedure.
$v3           The return vectors from rule3() procedure.
$vTotal       The sum of $v1, $v2, and $v3.
$oldPos       The seed position prior to calling rules 1, 2,
              and 3.
$cStart       The axis curve's starting point.
*/
vector $v1, $v2, $v3, $vTotal, $oldPos, $cStart, $trans;

/*
$z            The Z-position of the first seed (seed0).
$maxSpan      The number of curve spans corresponding to the
              smallest diameter fiber. Spans are curve sections
              between edit points.
$minSpan      The number of curve spans corresponding to the
              largest diameter fiber.
$span         The number of spans for the current fiber curve.
$maxD         The largest seed diameter.
$minD         The smallest seed diameter.
*/
float $z, $maxSpan, $minSpan, $span, $maxD, $minD, $diameter;

/***** INITIALIZE THE VARIABLES. *****/

// Set the global variable $frameCheck for use in the
procedures.
$frameCheck = frame;
$maxD = 18;
$minD = 3;
```





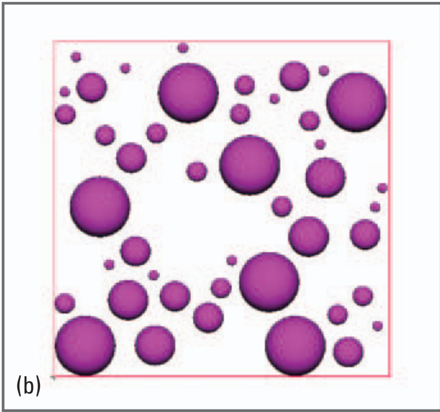


FIGURE 17.12

This front view shows the seeds at the start (a) and the end (b) of the first 19 frames of a modeling run. In the interim, rule2() and rule3() were called each frame to eliminate seed intersections and keep them within the bounding box (gray square).

Check if the seed journey is complete

The first thing the expression must do is check if the seeds have completed their journey. $end is used as a Boolean variable (although it's declared as an integer), with 0 equal to *false* and 1 equal to *true*. Also, since all seeds move uniformly in the Z-direction, you need to only query the translateZ value of the first seed to know whether all seeds have reached their destination and therefore whether to set $end equal to 1.

```
/***** MAIN BODY *****/

// Check if the seeds have reached the scaffold end.
$z = `getAttr ($seedNames[0] + ".tz")`;
if ($z >= $cubeSize * 1.5) $end = 1;

if ($end == 0) { // Seeds are still traveling
```

Main loop

This is a for loop that increments for every seed in the scene, calls the rule procedures, and builds the axis curves. Because $seedCount is a global variable and was initialized in resetSeeds, you needn't set its value in this expression. From frame 2 to 20, only rule2() and rule3() are called in order to space out the seeds while keeping them inside the bounding box (see Figure 17.12). At frame 20 the curves are started. After frame 20 all three rules are called.

```
// Loop for every seed.
for ($i = 0; $i < $seedCount; $i++) {

    // Get the seed position at the end of the previous
      frame.
    $oldPos = $seedPos[$i];

    /***** RULES OF MOTION *****/

    if (frame <= 20) {

        // Call the avoid and bounding procedures.
        $v1 = <<0, 0, 0>>;
        $v2 = `rule2($i)`;
        $v3 = `rule3($i)`;
    }
```

```
else { // frame > 20.

    // Call the random walk (rule1) as well.
    $v1 = `rule1($i)`;
    $v2 = `rule2($i)`;
    $v3 = `rule3($i)`;
}
```

Update the seed position

Now the rule vectors are added to the seed's current position to determine its new position. `$vTotal` is shown graphically in Figure 17.07. `$v1_old[]` stores components of the seed's current motion that you'll use in `rule1()` as the seed's persistence vector. This vector incorporates the full random walk component and half of each of the avoidance and bounding components of motion.

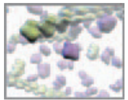
```
// Update the position array for seed $i.
$vTotal = $v1 + $v2 + $v3;
$seedPos[$i] += $vTotal;

/* Store the previous seed direction for persistence.
This vector is used in rule 1. */
$v1_old[$i] = $v1 + ($v2 + $v3)/2;

// Set the translate attribute for seed $i.
$seed = $seedNames[$i];
$trans = $seedPos[$i];
setAttr ($seed + ".translate") ($trans.x) ($trans.y)
  ($trans.z);
```

The degree to which the avoidance and bounding vectors contribute to persistence is yet another degree of control you have over scaffold shape. When determining how to calculate `$v1_old[]`—the random walk persistence vector—we found that including the full avoidance and bounding vectors caused seeds to intersect more than we'd like. However, excluding avoidance and bounding vectors from persistence altogether resulted in seed paths (or fiber axes) too angular and lacking the smooth curvature seen in natural ECM scaffolds. A happy medium was found using *half* of the avoidance and collision vectors. Incorporating these values into persistence achieves good curvature without overtly forcing seeds to intersect.

Build the fiber axis curves

At frame 20, after the seeds have untangled, a curve object is created for every seed. Each curve is given a unique name and a custom attribute called `diameter`. Both features will be used in the next project to locate cells on the fiber surfaces.

```
/***** MAKE OR UPDATE THE FIBER CURVES *****/

/*
Create the name of the fiber curve. Begin by tokenizing
the seed name at the split character "d". E.g. "seed3_7"
gets split into "seed" and "3_7";
*/
tokenize $seed "d" $index;
$curve = "fiberAxis" + $index[1];
$diameter = `getAttr ($seed + ".scaleX")`;


if (frame == 20) {

    // Make a fiber curve.
    curve -point ($trans.x) ($trans.y) ($trans.z) -n
      $curve;
```




```
            /* Add a custom diameter attribute and set it to
            match the diameter of the corresponding seed. */
            addAttr -ln diameter -at double -min 0 -max 48 -dv 0
              $curve;
            setAttr -e -keyable true ($curve + ".diameter");
            setAttr ($curve + ".diameter") $diameter;
        }
```

After frame 20, a CV is added to the curve at each frame using the append flag with the curve command.

```
        else if (frame > 20) {

            // Append an edit point to the curve.
            curve -append -p ($trans.x) ($trans.y) ($trans.z)
              $curve;

            // Increment length of the fiber.
            $length[$i] += mag($trans - $oldPos);

        }
    } // End loop for every seed.

} // End if ($end == 0).
```

Next, you'll rebuild the curve, reducing its spans to the number specified in the `maxSpan` and `minSpan` attributes of your control widget. Reducing the spans smoothed the fiber axis curve by eliminating CVs. However, if too few spans exist and the curve has frequent deflections, the resulting surface may not maintain a cylindrical shape along its length. Furthermore, the number of spans will be scaled according to fiber diameter. A small diameter fiber requires more CVs to maintain a consistent cross-section than does a large diameter fiber. The widget attributes maxSpan and minSpan store the values corresponding to the 3 and 18 μm fiber diameters, respectively. The intermediate span values will be calculated in the code below, assuming a linear relationship between spans and diameter.

The rebuildCurve command has a flag called `keepRange` flag which you'll set to 0 in order to reparameterize the curves to a range of 0–1. This allows you to query a point on a curve using the `pointOnCurve` command and a parameter value between 0 and 1; 0 corresponds to the start of the curve and 1 to the end.

```
/***** BUILD THE FIBER SURFACES *****/
if ($end == 1) {

    // Stop playback.
    play -state off;

    // Change $end so that this code will be called only once.
    $end = 2;

    // Loop for every curve.
    $curveNames = `ls -transforms "fiberAxis*"`;
    $i = 0;
    for ($curve in $curveNames) {

        $diameter = `getAttr ($seedNames[$i] + ".scaleX")`;

        // Rebuild the curve.
```

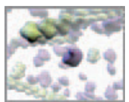
```
$maxSpan = `getAttr widget.maxSpan`;
$minSpan = `getAttr widget.minSpan`;
$span = (($minSpan - $maxSpan)*$diameter + ($maxD*$maxSpan
  -$minD*$minSpan))/($maxD - $minD);
rebuildCurve -keepRange 0 -spans $span $curve;

// Tokenize $curve to get its index number.
tokenize $curve "s" $index;

// Make the NURBS profile circle.
$circle = "nurbsCircle" + $index[1];
circle -center 0 0 0 -normal 0 0 1 -radius ($diameter/2)
  -n $circle;

// Query the curve's starting point.
$cStart = `pointOnCurve -parameter 0 -position $curve`;

// Move the circle to the start of the curve.
move -relative ($cStart.x) ($cStart.y) ($cStart.z) $circle;
```

Note that you're creating the NURBS circle at the Maya world origin (0 0 0) and then moving it to the starting point of the curve. Why not just create the circle at this second location in the first place? If you do, the circle Translate attributes will be 0 0 0 even though the object center is not at the world origin. Because of the way the extrude algorithm works, Maya will double transform the resultant surface, placing the fiber surface nowhere near the axis curve. Making the circle at the origin and then moving it with the move command to the axis curve origin avoids the double transformation, placing the surface at its proper location along the fiber axis.

There are three key flags to be aware of for the extrude command: extrudeType, fixedPath, and useProfileNormal. An extrudeType value of 2 specifies a *tube*. If true (a Boolean value), fixedPath positions the extruded surface relative to the axis curve as opposed to the profile curve (see Figure 17.13 for the difference between a true and false setting for this flag). If useProfileNormal is true then the surface will follow the profile normal direction as it extrudes along the path of the fiber curve, allowing the surface to accurately reflect the curvature of the path.

```
// Extrude the NURBS surface.
$surface = "fiberSurface" + $index[1];
extrude -extrudeType 2 -fixedPath 1 -useProfileNormal
  1 -n $surface $circle $curve;

// Add a custom length attribute to the surface.
addAttr -ln length -at double -dv 0 $surface;
setAttr -e -keyable true ($surface + ".length")
  $length[$i];

// Add a custom diameter attribute to the surface.
addAttr -ln diameter -at double -dv 0 $surface;
setAttr -e -keyable true ($surface + ".diameter")
  $diameter;

// Create closestPointOnSurface node and connect it to
  the surface.
$cpos = "cpos" + $index[1];
createNode closestPointOnSurface -n $cpos;
connectAttr -force ($surface + ".worldSpace[0]")
  ($cpos + ".inputSurface");

// Add a diameter attribute to cpos.
addAttr -ln "diameter" -at double -dv $diameter $cpos;
setAttr -e -keyable true ($cpos + ".diameter");
```

Note: In earlier versions of Maya, the -name flag in the extrude command didn't work and the resulting object was given a default name. If you're using a version of Maya earlier than 8, you may have to name each fiber surface immediately after it's created by using the rename command.



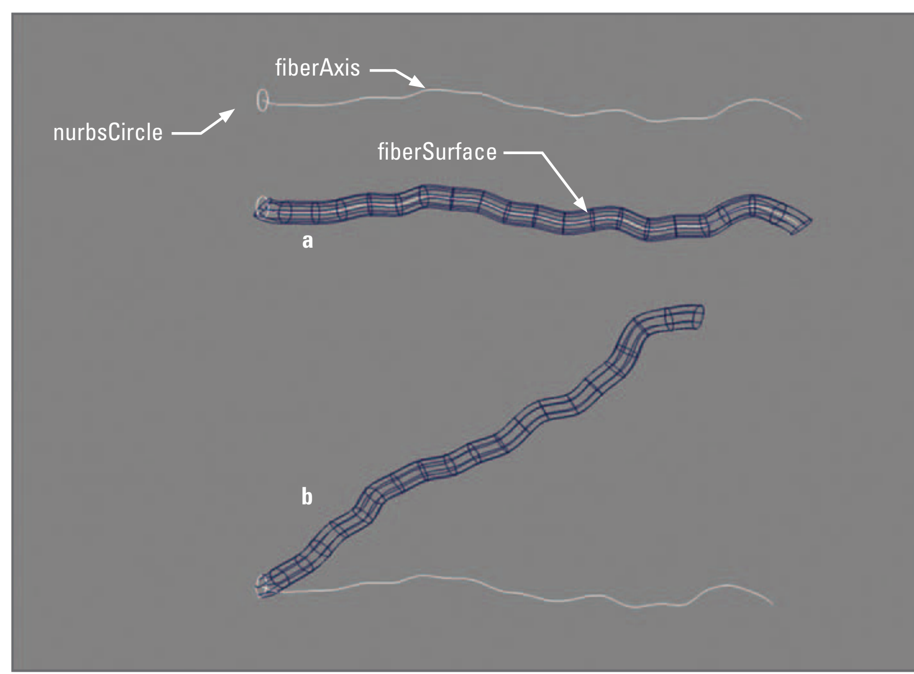
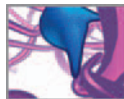


FIGURE 17.13

Setting the fixedPath flag to "1" in the extrude command positions the extruded surface relative to the fiber axis path (a) rather than the profile curve (b).

```
            // Create pointOnSurfaceInfo node and connect it to the
              surface.
            $posi = "posi" + $index[1];
            createNode pointOnSurfaceInfo -n $posi;
            connectAttr -force ($surface + ".worldSpace[0]")
              ($posi + ".inputSurface");

            // Group together each fiber's curve, circle, and
              surface nodes.
            $fiberGroup = "fiberGroup" + $index[1];
            group -n $fiberGroup $curve $circle $surface;

            // Increment the counter.
            $i++;

        } // End loop for every curve.

        // Group all of the fibers together.
        group -n "groupFibers" `ls -tr "fiberGroup*"`;

        // Select the widget so scaffold parameters appear in the
          channel box. select widget;

    } // End if ($end == 1).
} // End moveSeeds expression.
```

That's it for the `moveSeeds` expression. Next let's give it something to call: the rule procedures.



rule1(): The random walk

rule1() calculates the random walk we described starting on page 487. Let's start with the header information. Again, to save space, we'll include only the description.

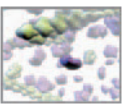
```
/* Description:
This procedure moves seeds randomly in the xy plane and positively
in the z-direction. To "run" is to continue in the same direction.
Between persistence runs, a seed picks a new direction. This
procedure is called from the moveSeeds expression.
*/
```

Note that this procedure is of type "vector" and therefore returns a vector value to moveSeeds. It requires a single argument, $i, which is the current seed number.

```
global proc vector rule1(int $i) {

    /***** DECLARE THE VARIABLES. *****/

    global vector $v1_old[];
    global string $seedNames[];
    global float $dx, $dy, $dz;
    global int $persistence[], $persistMin, $persistMax;

    /*
    $randomWalk     The return value of this procedure.
    */
    vector $randomWalk, $trans;

    /*
    $xDir           The X-component of the random walk.
    $yDir           The Y-component of the random walk.
    $diameter       The diameter of the current seed.
    $sizeBias       Used to scale the random walk steps according to
                    fiber (or seed) diameter.
    $base           Used to calculate $sizeBias.
    $exponent       Used with $base to calculate $sizeBias.
    */
    float $xDir, $yDir, $diameter, $sizeBias, $base, $exponent;
```

The persistence[] variable stores the time, in frames, to the next direction change in the random walk. It was set to 0 for every seed in the resetSeeds expression using the clear command. Before this procedure calculates new directions for the current seed, it must check to see if it's *persisting*.

```
    /***** MAIN BODY *****/

    // Has this seed ceased persisting?
    if ($persistence[$i] < 1){

        // Pick a new direction.
        $xDir = floor(rand(-1,2)) * $dx;
        $yDir = floor(rand(-1,2)) * $dy;

        // Pick a new $persistence.
        $persistence[$i] = rand($persistMin, $persistMax);

    }
```



The floor command returns the largest integer that is still less than the argument given to it. For example, floor(-0.09) returns -1 and floor(0.09) returns 0. When used with the rand() command it allows you to generate random integers. If persistence[] for the current seed is >1, the seed will continue on its present course using the $v1_old[] stepwise displacement vector from the previous frame. Then persistence[$i] is decremented by 1.

```
else {

    // The seed is persisting and takes the same step it did
      last time.
    $trans = $v1_old[$i];
    $xDir = $trans.x;
    $yDir = $trans.y;

    // Decrement $persistence By 1 frame.
    $persistence[$i] -= 1;
}
```

With less cross-sectional rigidity, smaller fibers will tend to show more directional deflection over a given distance than larger ones. The next segment of the code scales the random walk result for different fiber sizes, and therefore has a large impact on scaffold architecture. The $bias variable queries the widget sizeBias attribute, and is used as the exponent with the MEL pow() function. For a sizeBias value of 1, the random walk is scaled down for all fibers by a factor of 3/$diameter. This means that 3 µm fibers will get the full effect of the random walk vector calculated above, while 18 µm fibers will get 3/18 × the effect of the random walk. Higher sizeBias values result in smaller deflections for all fibers with a diameter >3. We recommend starting with a sizeBias value of 0.05 and then increasing it with each successive modeling run as you tune your scaffold design.

```
// Scale the random walk according to fiber diameter.
$diameter = `getAttr ($seedNames[$i] + ".scaleX")`;
$exponent = (`getAttr widget.sizeBias`);
$base = 3/$diameter;
$sizeBias = pow($base, $exponent);
$xDir = $xDir * $sizeBias;
$yDir = $yDir * $sizeBias;
```

The last step returns the result of this procedure to the expression that called it, moveSeeds.

```
    // Return the random walk vector to the moveSeeds expression.
    $randomWalk = <<$xDir, $yDir, $dz>>;
    return $randomWalk;

} // End procedure.
```

rule2(): Collision avoidance

Here you'll use a strategy similar described starting on page 489 and in Figure 17.08. Like rule1() this procedure takes the current seed, $i, as its argument and returns a vector to the moveSeeds expression.

```
/* Description:
This procedure moves seeds apart if they come within a critical
distance of one another. It is called from the moveSeeds expression.
*/

global proc vector rule2 (int $i) {

    /***** DECLARE THE VARIABLES. *****/

    global vector $seedPos[];
    global string $seedNames[];
    global int $seedCount;
    int $i, $j;

    /*
    $avoid          The vector returned from this procedure.
    $sepUnit        The unit vector (of magnitude 1) for $centerSep.
    */
    vector $avoid, $sepUnit;

    /*
    $CenterSep      The distance between the centers of seeds $i
                    and $j.
    $surfaceSep     The distance between the surfaces of seeds $i
                    and $j.
    $iDiameter      The diameter of seeds $i.
    $jDiameter      The diameter of seeds $j.
    $spacingMin     The minimum tolerated surface separation between
                    seeds $i and $j.
    $avoidScale     Scales the avoid vector.
    */
    float $CenterSep, $surfaceSep, $iDiameter, $jDiameter,
      $spacingMin;
    float $avoidScale;

    /***** INITIALIZE THE VARIABLES. *****/

    $avoid = <<0,0,0>>;
    $spacingMin = `getAttr widget.spacingMin`;
    $avoidScale = `getAttr widget.avoidScale`;
    $iDiameter = `getAttr ($seedNames[$i] + ".scaleX")`;
```

This procedure evaluates the separation, `$surfaceSep`, between the current seed and every other seed in the scene, one at a time. If `$surfaceSep` is less than a critical value, `$spacingMin`, then a vector is added to the return avoidance vector, `$avoid`. After all seeds have been considered, `$avoid` is the sum total of every step the current seed must take to avoid colliding with the others.

```
    /***** MAIN BODY *****/

    // Loop for every seed in the scene.
    for ($j = 0; $j < $seedCount; $j++) {

        // The current seed can't collide with itself. Therefore:
        if ($j != $i) {

            // How far apart are seed $i and $j centers?
            $centerSep = mag($seedPos[$i] - $seedPos[$j]);
```

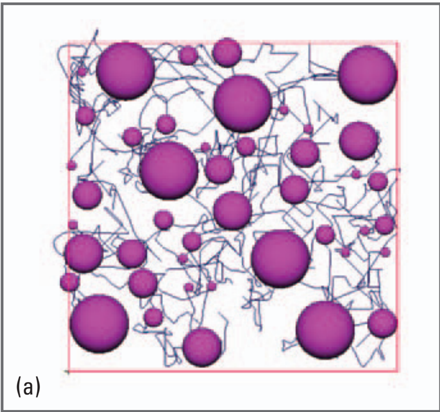
```
            // How far apart are their surfaces?
            $jDiameter = `getAttr ($seedNames[$j] + ".scaleX")`;
            $surfaceSep = $centerSep - ($iDiameter + $jDiameter)/2;

            if ($surfaceSep < $spacingMin) {

                // The seeds are too close to one another.
                $sepUnit = unit ($seedPos[$i] - $seedPos[$j]);
                $avoid += ($avoidScale * $sepUnit);
            }
        }
    }

    // Zero the Z-component of $avoid.
    $avoid = << $avoid.x, $avoid.y, 0 >>;
    // Return the collision avoidance vector to the moveSeeds
      expression.
    return $avoid;

} // End procedure.
```

That's it for `rule2`! You have just one more short procedure to complete and you'll be ready to build your first scaffold.

rule3(): Bounding

Like `rule1()` and `rule2()`, this procedure takes the current seed, `$i`, as its argument and returns a vector to the `moveSeeds` expression.

```
/* Description:
This procedure constrains seeds to the scaffold bounding box. The
bounding box depth and width dimensions are stored in the cubeSize
attribute of the widget.
*/

global proc vector rule3 (int $i) {

    /***** DECLARE THE VARIABLES. *****/

    global vector $seedPos[];
    global float $cubeSize;
    global string $seedNames[];
    int $i;

    /*
    $bound          The return vector for this procedure.
    */
    vector $bound, $trans;

    /*
    $radiusSeed radius.
    $outside        The distance from the seed surface to a
                    corresponding side of the bounding box.
    $boundScale     A multiplier to scale the X
```



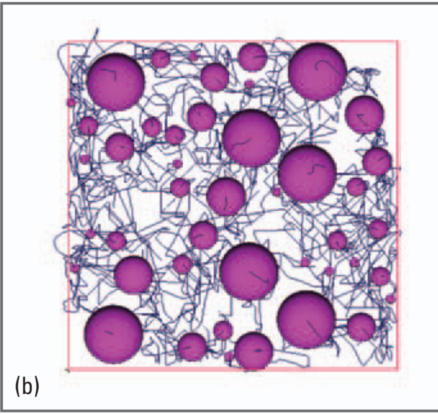


FIGURE 17.14

This Front view shows the seeds and fiber axes (a) half-way through and (b) at the end of a modeling run. This view demonstrates how rule2() and rule3() keep the seeds (and therefore the fiber axes) largely free of intersections and within the bounding box (red square).

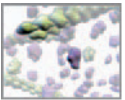
```
$pushX          The X-distance by which the seed will be pushed
                back inside the bounding box.
$pushY          The Y-distance by which the seed will be pushed
                back inside the bounding box.
*/
float $pushX, $pushY, $outside, $radius, $boundScale;

/***** INITIALIZE THE VARIABLES. *****/

$boundScale = `getAttr widget.boundScale`;
$trans = $seedPos[$i];
$radius = `getAttr ($seedNames[$i] + ".scaleX")`/2;
```

Determine if the seed is outside of the bounding box in the X-direction.

```
/***** MAIN BODY *****/

// Check bounding in the X-direction.
if (($trans.x - $radius) < 0) {

    // The seed is outside the bounding box in the negative
      X-direction.
    $outside = 0 - ($trans.x - $radius);
    $pushX = $boundScale*sqrt($outside);
}
```

Using the square root of `$outside` gently pushes the seed back toward the bounding box (note the smooth curvatures of fiber paths that leave and then re-enter the bounding box in Figure 17.14). A linear or exponential push (exponent > 1) tends to launch the seed back into the bounding box, causing large loops in fiber paths and collisions between seeds.

```
else if (($trans.x + $radius) > $cubeSize) {

    // The seed is outside the bounding box in the positive
      X-direction.
    $outside = ($trans.x + $radius) - $cubeSize;
    $pushX = -$boundScale*sqrt($outside);
}
```




```
        else { // The seed is in the box.
            $pushX = 0;
        }
```

Determine if the seed is outside of the bounding box in the Y-direction.

```
        // Check bounding in the Y-direction.
        if (($trans.y - $radius) < 0) {

            // The seed is below the bounding box.
            $outside = 0 - ($trans.y - $radius);
            $pushY = $boundScale*sqrt($outside);
        }
        else if (($trans.y + $radius) > $cubeSize) {

            // The seed is above the bounding box.
            $outside = ($trans.y + $radius) - $cubeSize;
            $pushY = -$boundScale*sqrt($outside);

        }
        else { // The seed is in the box.
            $pushY = 0;
        }

        // Return the bounding vector to the moveSeeds expression.
        $bound = <<$pushX, $pushY, 0>>;
        return $bound;

    } // End procedure.
```

This concludes the three rule procedures. With the rule-based program architecture, you can add additional rules of motion if you wish by encoding them in procedures and plugging them in to the main `moveSeeds` expression. Now let's load the different script elements and build a scaffold!

Methods: Grow your scaffold!

Prepare your Maya scene

Start Maya or, it's already running, start a new scene. Create a new Maya Project directory to use for the scaffold model and for the cell migration model you'll build in the next chapter:

1. **From the main menu bar, choose File → Project → New.**
2. **Enter cellMigrationProject in the Name field.**
3. **For Location, browse to your Maya Projects directory or another location on your hard drive where you'd like to save this project.**
4. **Enter scenes and images in the appropriate text fields and then press the Accept button.**

Source the script elements

If you haven't done so already, save your `makeSeeds` and `rule` procedure files in your Maya Scripts directory. Make sure the file names match the respective procedures.

For instance, the procedure called `rule1` should be saved in a file called `rule1.mel`. You may have tested your procedures by entering them in the Script Editor as you created them. Likewise for expressions in the Expression Editor. If you haven't yet tested your scripts, there's a chance they may contain errors. If this is the case, we recommend sourcing one script at a time fixing the bugs as they're flagged by Maya. You can check your scripts against our working copies which you'll find on the CD-ROM:

17_ECM_Scaffold/MEL/makeSeeds.mel
/moveSeeds.txt
/resetSeeds.txt
/rule1.mel
/rule2.mel
/rule3.mel

Now source the procedures:

1. **Refresh the search path contents. In the Script editor, enter:**

```
rehash;
```

2. **Source the script files. In the Script editor, enter:**

```
source "makeSeeds.mel";
source "rule1.mel";
source "rule2.mel";
source "rule3.mel";
```

Make the seeds

Your first step in building the scaffold is to make those little pathfinders, the seeds. Begin with the parameters outlined earlier in the chapter:

- **100 µm thick × 100 µm wide × 150 µm long.**
- **40 seeds (for 40 fibers).**
- **a size distribution of 30%, 29%, 18%, 8%, and 15% for the 3, 6, 9, 12, and 18 µm diameter fibers, respectively.**

To call the procedure:

Enter the following in the Command Line or Script Editor:

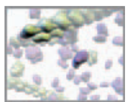
makeSeeds(100, 40, 30, 29, 18, 8, 15)

The speed of execution will vary depending on computing power and memory, but should take no more than a few seconds in any case. When the procedure is done, you'll see in the Command Line, the message you sent yourself:

```
The seeds are ready to go!
```

Create the expressions

Here you'll create the `resetSeeds` and `moveSeeds` expressions. If you didn't build the expression scripts earlier in the chapter, copy them from the CD-ROM to your Maya Scripts directory.



Reminder: To Maya typographers quotation marks (“ ”) are unknown characters and will generate errors when used in MEL scripts and animation expression.

1. **Open resetSeeds.txt (either the file you created or the one on the CD-ROM) in the text editor of your choice.**
2. **Ensure that the text editor is set to *not* use typographer's quotation marks.**
3. **Select and copy the entire script to the clipboard.**
4. **In Maya, enter ExpressionEditor in the Command Line to launch the Expression Editor, or select it from the menu Windows → Animation Editors → Expression Editor.**
5. **Choose Select Filter → By Expression Name.**
6. **Press the New Expression button.**
7. **LMB + click in the Expression text field.**
8. **Press Ctrl + V to paste your expression into the text field.**
9. **Press the Create button at the bottom of the Expression Editor.**
10. **In the Expression Name field, replace the default name with resetSeeds and press Enter.**
11. **Repeat steps 6 through 10 for the moveSeeds expression, but name it moveSeeds in the Expression Name field.**
12. **Press Ctrl + S to save your scene with the expressions in it.**

Prepare the scene

Adjust the view

If you move around your scene a little, you may see some of the seeds disappearing. This is due to the default clipping plane settings (`Near Clip Plane: 0.10; Far Clip Plane: 1000.0`) of the `persp` camera; the array of seeds is large enough that it intersects the clipping planes as you move the camera back and forth. To remedy this, set the Far Clip Plane to a sufficiently large value (say, 20,000). While you're at it, you may as well set the Far Clip Planes for all cameras once to save having to do it individually as needed.

Enter the following four code lines in the Script editor:

```
setAttr perspShape.farClipPlane 20000;
setAttr topShape.farClipPlane 20000;
setAttr frontShape.farClipPlane 20000;
setAttr sideShape.farClipPlane 20000;
```

Preload resetSeeds

Thanks to the effort you put into the expressions and procedures, all you do to build a scaffold model is press Play. However, the first time you do this, it's important to ensure the procedure `resetSeeds` has evaluated in order to load the key global variables.

1. **In the Expression Editor → Selections → Expressions field, select resetSeeds and press the Edit button.**

Parameter	Scaffold			
	Figure 17.15a	Figure 17.15b	Figure 17.15c	Figure 17.15d
cubeSize	100	100	100	100
maxSpan	40	40	80	60
minSpan	20	20	40	40
dx	1	3	1	1
dy	1	3	1	1
dz	1	1	0.5	1
persistMax	10	10	10	30
persistMin	0	0	0	20
sizeBias	0.05	0.05	0.05	0.05
spacingMin	2	2	2	2
avoidScale	2	2	2	2
boundScale	1	1	1	1

TABLE 17.03

These are the parameter values we used to generate the scaffolds shown in Figure 17.15. The parameters most responsible for the differences between scaffolds are highlighted in the Table.

Set the Playback range

A 150 µm long scaffold will take 150 ÷ $dz + 20 = 170 frames to complete its self-build. Give yourself a little extra room and set the Playback range from 1 to 180.

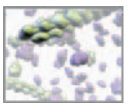
Inspect the scaffold parameters

Take a moment to look at the custom attributes of the widget object you made in the `makeSeeds()` procedure. Relative to the seeds, the widget may be hard to see in the scene, so use the Outliner to select it. In the Channel Box you'll see the custom attributes listed under the `transform (widget)` node, with the values you set using the `makeSeeds()`. By adjusting these parameters in the Channel Box at the start of each model making run, you can vary the outcome of each scaffold. We've explored these values in our own research on ECM and cell migration, but much remains to be seen about how different combinations of values affect the model. The parameters shown in Table 17.03 were used to generate the scaffolds shown in Figure 17.15.

Press Play!

All that's left to do is ...

1. **Adjust your perspective view to get a comfortable view of the scene.**
2. **Press the Play button in the Playback controls.**
3. **Sit back, relax, and enjoy the show.**



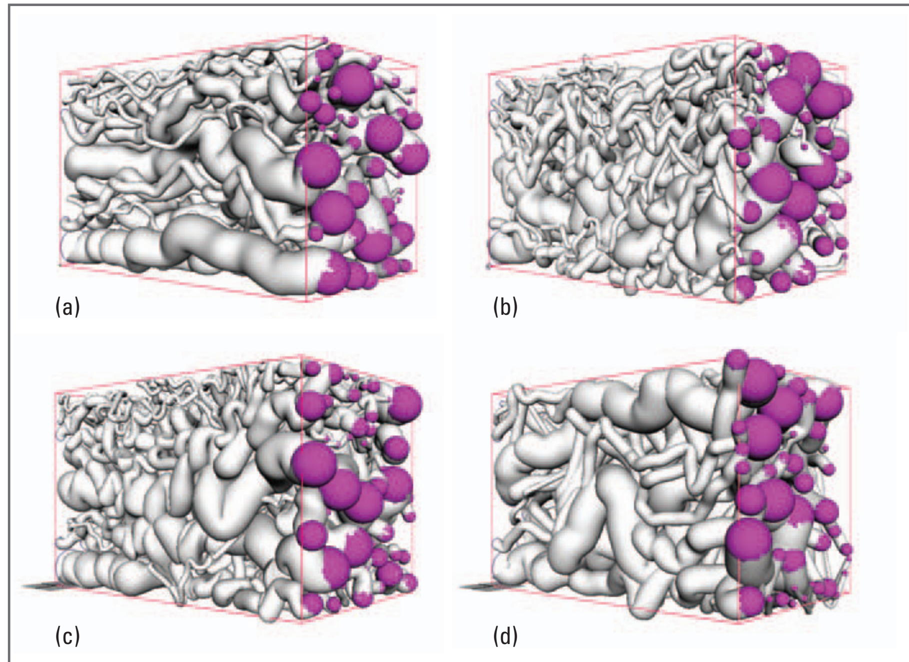


FIGURE 17.15

Four scaffold we built by varying key model parameters (see Table 17.03). The box dimensions are $100 \times 100 \times 150$ µm.

Once your model is complete you have two options for how to proceed:

A. **Save the file at the current frame to preserve your scaffold model.** You may wish to delete both the resetSeeds and makeSeeds expressions in the Expression Editor to prevent yourself (or someone else) from accidentally deleting your model by rewinding the scene to frame 1.

   Note that you can also select and export the fiber axes and surfaces to a new file—using Export Selected, located under the main File menu—and then reset your scene (see below).

B. **Press the rewind button to reset the scene.** If you included the set of "delete" lines at the end of resetSeeds, then your fiber axis curves, surfaces, and profile splines will be eliminated from the scene, leaving only the seeds you began with.

If you chose option B, you can simply hit Play again to make another model with the current set of seeds. If you want to vary the seed number, scaffold size, and or fiber size distribution, then you simply select and delete the group of seeds and the `widget`, and then run `makeSeeds()` again with a new set of parameters. You won't have to recreate the expressions since they already exist in the scene, although you'll want to press Edit in the Expression Editor to evaluate resetSeeds for the new set of seeds.

Results: Parameter effects

Figure 17.15 shows a sampling of scaffolds that we built, varying the design parameters. Worth noting is the effect that the displacement variables `$dx`, `$dy`, and `$dz` have on collision avoidance (`rule2`). Small values (1–2 units) allow the `rule2()` to work with

reasonable effectiveness. As the values were increased—upwards of 3 or more units—the resolution for collision detection became coarser and fiber paths intersected (Figure 17.15b). Figure 17.15c illustrates the effect of dz—the smaller its value, relative to dx and dy, the wavier the fibers. Finally, higher persistence values result in greater fiber deflection (Figure 17.15d). For each modeling run we tuned the `maxSpan` and `minSpan` values to achieve fibers whose diameters remained reasonably constant along their lengths.

If time permits, play around with the various settings of the model. You made this process easy by listing the model parameters as attributes of the widget object.

Playblast your scene

Depending on the processing speed and RAM of your computer, watching a scaffold evolve from many seeds can be a bit like watching paint dry (although slightly more interesting) because it happens so slowly. If you want to capture a dynamic view of your scaffolds as they evolve, you can set up your camera and then make a playblast rendering while you make plans for your next in silico project.

Save your scene

You will use your scaffold model as the substrate for a migrating cell population in the next chapter. When you are satisfied with the model, delete the `resetSeeds` and `moveSeeds` expressions, then save your scene as a Maya ASCII file under the name `scaffold.ma`.

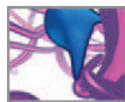
Summary

The ECM of living tissues are intricate 3D tapestries in which many kinds of fibrous molecules intertwine and cross react. The methods we've introduced to you here will get you started with Maya as a window on this little seen, poorly explored world. Alone, however, even the most exacting ECM is like a stadium without the baseball team and the spectators—an empty arena awaiting action. Let's fill it.

References

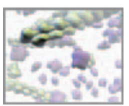
1. Nelson CM, Bissell MJ: Of extracellular matrix, scaffolds, and signaling: Tissue architecture regulates development, homeostasis, and cancer. *Annual Review of Cell and Developmental Biology* 22: 287–309, 2006.
2. Armour AD, Fish JS, Woodhouse KA, Semple JL: A comparison of human and porcine acellularized dermis: Interactions with human fibroblasts in vitro. *Plastic and Reconstructive Surgery* 117(3): 845–856, 2006.
3. Sharpe J, Lumsden CJ, Semple JL, Woolridge N: *Fibroblast behavior in human dermal substitutes: A computer simulation model, I—3d collagen matrix model structure and visualization*, International Tissue Engineering Society Conference, Orlando, 2003.

This page intentionally left blank



18 Scaffold invasions

Modeling 3D populations of mobile cells



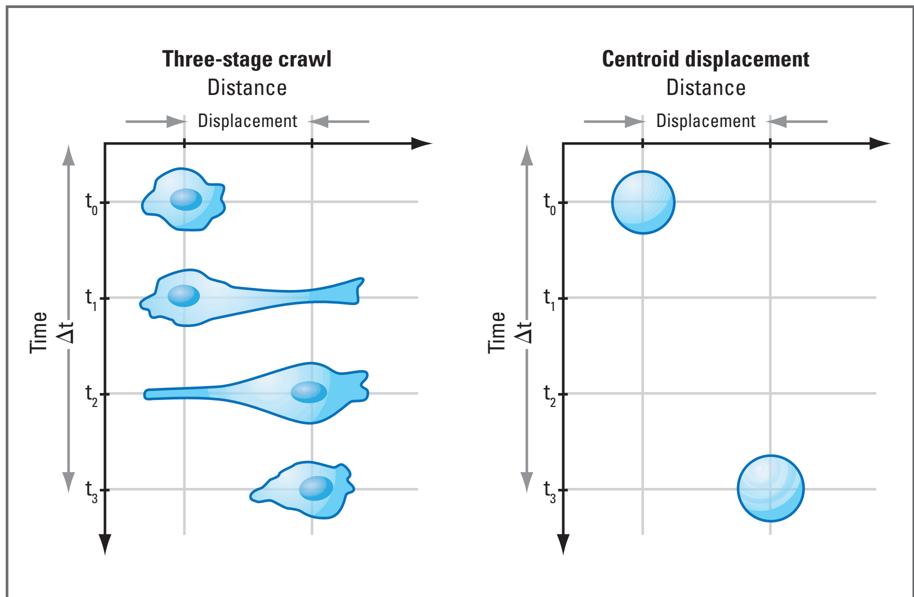


FIGURE 18.01

In this project you will build a cell migration algorithm that treats the three primary stages of locomotion as a collective event: the stepwise displacement of the cell center.

Introduction

In *Chapter 16* you modeled a single cell as an agent that behaves according to certain rules of motion. Then in *Chapter 17* you modeled agents called "seeds" whose trajectories became longitudinal axes for the extracellular matrix (**ECM**) collagen fiber bundles that composed your tissue scaffold. Like the single cell, the seeds moved in response to a set of rules. In the scaffold modeling MEL script, you parceled the motion rules into discrete software procedures called `rule1()`, `rule2()`, and `rule3()`. These rules were in turn "plugged in" to the main software component of the script—the `moveSeeds` expression—making for a modular software algorithm that can be modified by adding or removing rules.

In this project you will build on your progress modeling individual agents and on the cell biology concepts presented in the previous two chapters. The objective is to make a simulation involving a population of cells that migrate throughout the Maya tissue scaffold you built in *Chapter 17*. In *Chapter 16* you treated explicitly the stages of cell locomotion—protrusion, traction, and retraction. In the current project you will treat them collectively, and from one level up on the ladder of organizational hierarchy, by modeling the stepwise displacement of the cell center, a process that encompasses the individual crawling stages in a single event (Figure 18.01).

When your MEL script for this project is finished and working, you will have a tool for rapidly deploying a group of mobile cells in a scaffold model. The rules of motion depend on various parameters which you can change—as you did with the scaffold script—in order to observe their effects on migration behavior. Moreover, the rules themselves can be modified or removed altogether and new ones constructed and

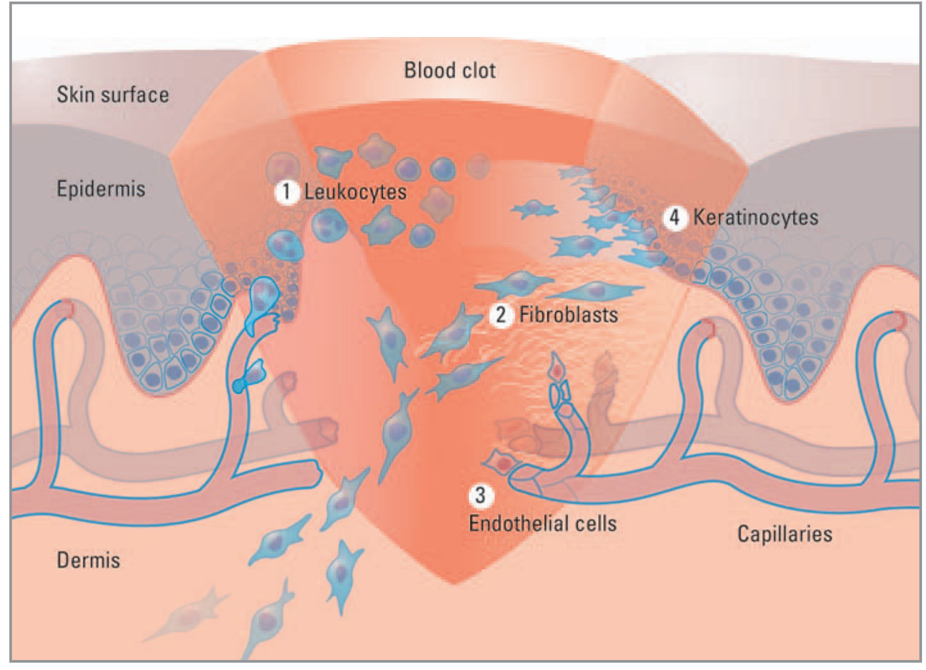


FIGURE 18.02

Collective cell migration is an essential feature of wound healing in the skin. Wound healing begins with an inflammatory response that includes leukocytes (white blood cells). In the skin's dermal layer fibroblasts migrate into a temporary scaffold of fibrin (the blood clot) then degrade the fibrin and synthesize and remodel a new scaffold of collagen. Endothelial cells migrate through the dermis, building new capillaries to provide a blood supply. In the epidermal layer, keratinocytes migrate as a confluent sheet to seal the wound.

then added to the algorithm to test different hypotheses about the behavior of cell groups.

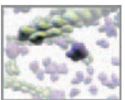
Problem overview

Cell migration as an emergent behavior

The change in state over time of a migrating cell population is a property that emerges from the minutia of cell mobility events beginning at the molecular level with signaling events and cytoskeleton turnover. The *state* could be any quantifiable property of the cell population such as the pattern of dispersion—the spatial arrangement of migrated cells. In *Chapter 16* you saw the single cell changing state—position and translocation speed—as a result of mobility events at the level of physical cell processes: protrusion, traction, and retraction. The cell's position and speed *emerged* from the rules governing the physical processes.

Mobile cell populations

Animal cells that are specialized for active mobility—and therefore lend themselves especially well to 3D agent-based models—include patrolling immune cells like lymphocytes and wound-healers like fibroblasts, keratinocytes, and endothelial cells (Figure 18.02). When cancer strikes, otherwise sessile cells can become specialized movers as well and aggressively invade adjacent tissues, disrupting normal bodily functions. Through intercellular signaling and by physically altering their ECM, many



individual cells can act as a coordinated front to accomplish a physiological goal. The overall movement of a group of cells is a behavior that emerges from the minutia of locomotion events on the level of individual cells.

Cells, scaffolds, and regenerative medicine

To researchers and practitioners in regenerative medicine, the ways in which cells infiltrate and interact with tissue scaffolds can be of great importance. Regenerative medicine aims to replace, or assist in the regeneration of, cells and tissues when the body is unable to do so itself. Such interventions cover a broad range of subspecialties and tissue types. Once such specialty involves the application of skin grafts to replace lost or damaged dermis in cases of severe burns and other situations involving deep trauma or disease of the skin. Currently, the gold standard for a skin replacement is skin itself—usually derived from a cadaveric donor and treated chemically to remove cellular tissue and soluble proteins that could elicit an immune response in the graft recipient, or host. The remaining tissue is largely composed of collagen fiber bundles which facilitate cell infiltration and mitigate scar formation. It is currently thought that the relative success of cadaveric skin grafts over alternatives is due to the positive response of host fibroblast cells to the natural architecture of the collagen scaffold. A key study that supports this view demonstrated a high degree of scaffold infiltration in cadaver grafts when compared with a potential alternative[1]. However, this success comes with a price—quite literally—since donor skin is rare and therefore costly. For less expensive and more plentiful alternatives such as synthetic or animal-derived tissues to be clinically viable—that is to facilitate normal wound healing—it must elicit a positive response from host cells and promote normal wound healing behavior. While there are numerous factors involved in such a response, the 3D tissue architecture itself has been identified as a key one, and the relationship between this architecture and cell migration flagged as an important area of research in regenerative medicine.

Emergence in a cell migration model

We can frame the infiltration of a tissue scaffold by cells—be they healing fibroblasts or invading cancer cells—as a population-level response to individual cell-scaffold and cell-cell interaction events. These include the cell crawling events associated with haptotaxis and chemotaxis that we looked at in *Chapter 16*. In our agent-based approach we treat each event as the outcome of a *rule*—like the rules you encoded in procedures in the previous chapter. The model evaluates the current state of each cell, applies the rules of migration, then updates the cell states (Figure 18.03)—a process that occurs at each time increment. As time progresses, the rules are applied, and the cell states changed, a pattern of migration for the entire population will emerge.

Parameters of cell migration

With an individual-based model you can study cell migration behaviors, or parameters, that emerge at different levels of detail. By modeling the stepwise translocation of each cell body in this project, you can evaluate parameters to quantify single cell locomotion (such as the mean squared displacement, mean speed, and directional persistence) and derive coefficients that characterize migration for

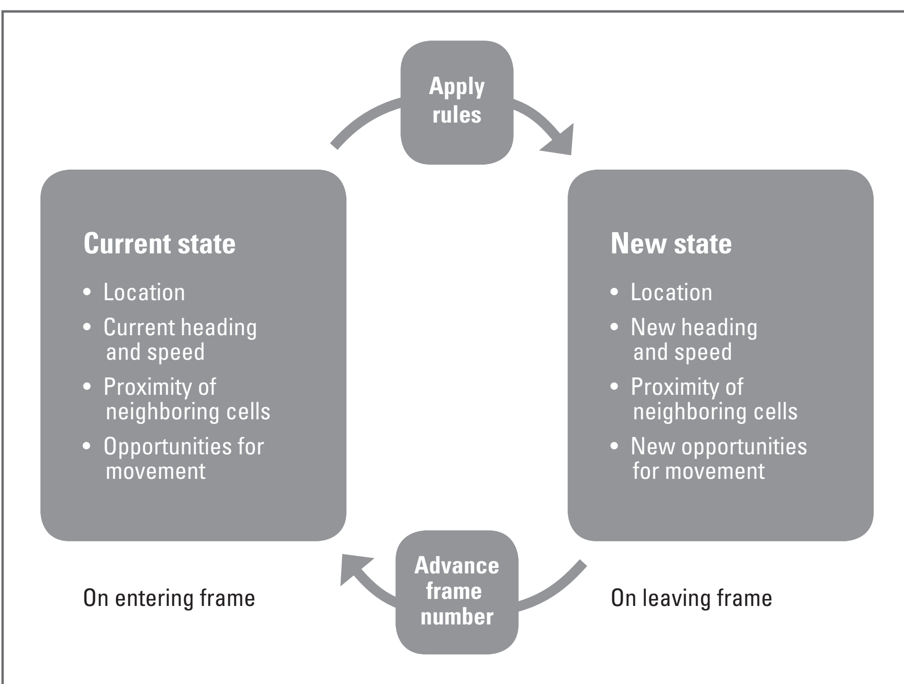


FIGURE 18.03

Outline for a cell population migration model.

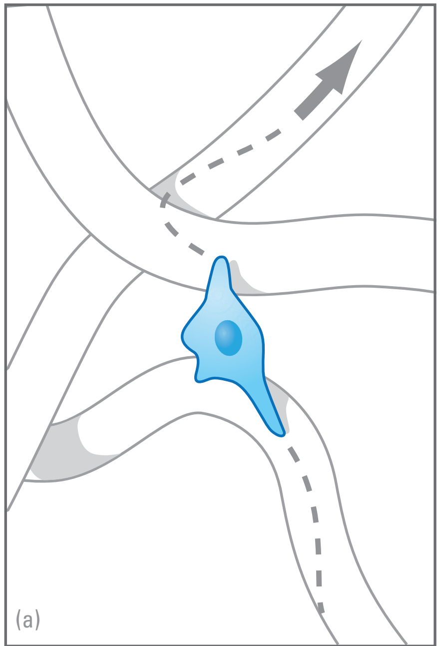
whole populations: random motility coefficient, chemotaxis/haptotaxis index, and chemokenesis/haptokenesis coefficient.

Cell migration in scaffolds

We can see that a cell makes chemical bonds called *adhesions* with its substrate in order to generate the traction force needed to move. Much attention in cell migration research has been given to adhesion molecules and ways in which they're regulated and, in turn, regulate cell locomotion events. For a moment, let's take for granted that the mechanism for adhesion is present—the cell has its components (integrins) and so does the ECM (**ligands**)—and consider other variables in the cell-ECM relationship that impact locomotion.

Many variables are at play in the complex relationship between cells and a scaffold environment—all of which contribute in some way to the emergent behavior of the cell population. In this project you will focus on a manageable piece of puzzle: the process of haptotaxis through which scaffold fibers serve as a substrate for cell crawling and a conduit for movement throughout the scaffold (Figure 18.04).

We'll deliberately omit some of the refinements such as the deformability of the moving cells, the way they can work with other cells to digest and remodel the matrix, and the arrival and departure of nutrients and waste products via surrounding blood vessels. This will let you focus on the first core problem of 3D cell group motion in the ECM, which you can then refine and extend as your interests develop. With 3D computational cell and tissue modeling still in its infancy, exciting opportunities abound for addressing factors like perfusion, cell deformation, and scaffold remodeling.



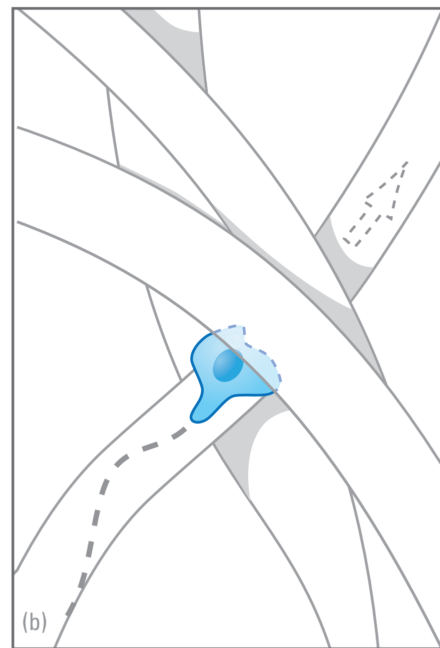


FIGURE 18.04

A migrating cell's progress from point A to point B can be both facilitated (a) and impeded (b) by the organization of ECM structures. This illustration depicts collagen fiber bundles as the conduits and obstacles to cell movement.

Cell migration nomenclature

Let's look briefly at the two key cell migration parameters that you'll model in this project.

Haptotaxis

The cell movement facilitated by contact with the substratum can be random in nature, directed, or both. A non-homogeneous substratum like the scaffold model you made in the previous chapter is bound to introduce bias into the motion of "randomly" moving cells by virtue of the fact that cells can travel only where the scaffold fibers go.

Chemotaxis

Chemotactic signals may originate from cells within the study population, from an external source such as the extracellular concentration of an ion like calcium in the tissue, or from another population of cells such as the immune cells involved in the inflammatory response to injury. Whatever the source—and it may in fact be a combination of influences—a spatial gradient of molecules that elicit a motility response in a cell biases the direction of its movement.

Cell-cell signaling

Chemically or physically mediated communication between cells is essential to many physiological and disease processes. Notable examples include the immune response, endocrine (hormone) function, and the growth and spread of cancer. Chemical

signals are a way for one cell to receive information from another. Going back to our Maya Hypergraph/biology network analogy from *Chapter 04* (page 82), you could think of a cell as a DG node and the signal as a connection between that node and another (the other cell). What the cell does in response to receiving the signal (often via membrane-bound receptor molecules) is analogous to a Maya node processing its input to calculate a result.

Furthermore, signaling between cells needn't be long range—like the passing of a hormone from an endocrine cell to its target far away in another part of the body. The fact that epithelial and mucosal cells (which line external and internal surfaces of the body) replicate and migrate to close defects in otherwise continuous cell sheets suggests that cells have a physical and chemical awareness of contact between each other. This ability to sense other cells at close range may help explain why solitary cells like fibroblasts or patrolling lymphocytes don't, under normal physiological conditions, bunch up or overlap one another. In contrast, they generally avoid contact with one another.

Now that we've established a vocabulary for discussing living cells engaged in migration—random and directed haptotaxis; biased movement due to chemotactic gradients; and contact avoidance—let's apply it in a definition of the model you're about to build.

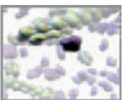
Model definition

This project realizes the scenario we described above: mobile cells that infiltrate an ECM scaffold via haptotaxis and chemotaxis. In your model, a population of cells will begin on the bottom surface of the scaffold you modeled in the previous chapter and migrate upwards through the scaffold. Each cell is then driven by rules for cell-matrix interaction (which incorporate both chemical and haptic cell responses) and cell-cell signaling. To develop the model you will build upon programming structures introduced in earlier chapters along with a novel technique for attaching the moving cells to the complex, 3D surfaces of the fiber scaffold.

Cell behavior

In order to simulate haptotaxis, you require a method to calculate a random walk (or crawl) over the surface of the fiber to which it's presently attached. We will refer to this method as the *cell crawling algorithm*. Biasing the random walk will introduce directed haptotaxis—in addition to that which we noted earlier is inherent in a non-homogeneous substrate. This bias can be built into the algorithm by manipulating the probabilities that govern the cell's choice of direction.

Each cell must also have the ability to sense contact with other fibers and then to decide whether or not to detach from its current fiber and transfer to another. We will call this property *transferring*. By giving cells a preference for transferring to fibers that allow them to move through the scaffold in a given direction—again by manipulating the probabilities that govern the cell's choices—you will effect a gradient of chemoattractant in the model; the preferred direction is the vector of chemotaxis and the probability for choosing that direction over any other is the magnitude of the chemoattractant. In your model, you will begin with a chemotaxis vector of <0,1,0> and a magnitude of 0.9 out of 1, thus setting up strong bias for cells to transfer to fibers above them (i.e., in the positive Y-direction) in the scaffold.



Cell-cell signaling

To prevent clustering, overlapping and interpenetration of cells, you will endow them with contact sensitivity and a mechanism for avoiding contact. This is the kind of close range signaling we mentioned above. Once your model is built you may wish to incorporate longer range signaling such as the ability for cells to broadcast their progress to others—a mechanism used by invading cancer cells.

The cell geometry

Your cell crawling algorithm will determine the direction and distance traveled in each time interval in a way that bundles the minutia of locomotion—molecular events, membrane protrusion, traction, and retraction—as a translocation step of the cell center. At the same time, each cell in your model requires a space-filling manifestation in the scene. This represents its boundaries both for the purpose of making it visible to you and for the detection of contacts between the cell and surrounding matrix fibers and other cells. A sphere primitive fulfills these requirements in an elegantly simple Maya object: the sphere has a center (its transform node) to displace each time the cell moves; it has a uniform contact radius represented by its surface; and a sphere can readily be discerned visually from the intertwining mass of the scaffold. Once you have this up and running, you can build on your progress to extend the simulation to include cells of more complex, changing shape.

We'll use the term **seeding density** to refer to the number of cells per unit area of the scaffold surface at the start of your simulation run. With cells that use close range signaling to communicate, it can be interesting to vary the seeding density and watch its effect on population behavior. You'll begin with 10 cells, a high density relative to clinical studies of cell-scaffold interaction in dermal regeneration[1] but necessary to get interesting cell-cell contact interaction effects in your small test scaffold.

The sutbstrate

To control the movement of your cells on a substrate it must be composed of objects whose surfaces can be surveyed for positions in Maya world space. You fulfilled this requirement in *Chapter 17* by constructing a scaffold out of parametric surfaces (NURBS); parameter values can be used to determine the world space position of any point on a NURBS surface, along with the corresponding surface normal which comes in handy when attaching a cell to the surface.

Using a relatively small, medium density scaffold—namely the one you made in the previous chapter—will allow you get reasonably quick simulation results on a computer of similar processing and memory attributes as our benchmark system (described on page xxi in the *Preface*) we used to develop and test this project. Figure 18.05 shows a scaffold like the one you made in *Chapter 17*, with its dimensions and *active volume* indicated. The active volume is the region of the scaffold in which this model's cells migrate. They are restricted to the active volume by the **boundary conditions** described below. In theory the size (length, width, height, and fiber packing density) of scaffold you use and, by extension, the number of cells in the simulation is limited only by the processing power and memory capacity of your computer, and by time constraints (i.e. how many minutes, hours, or days you can dedicate to a single simulation). The parameter values we've suggested will give you interesting simulation results quickly (less than 1 hour simulation time on our benchmark system).

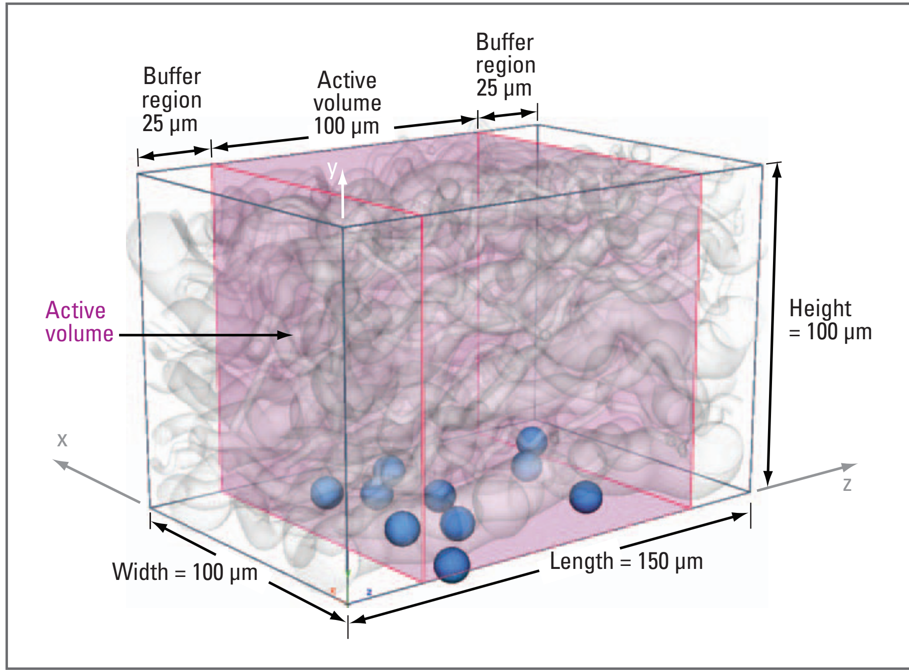


FIGURE 18.05

The model scaffold that will serve as the substrate for cell migration in this project. Cell locations and their contact radii are represented by the blue spheres. The active volume is the region in which cells can move and is limited along the Z-axis by the buffer regions.

Boundary conditions

Ideally your scaffold would be long and wide enough, relative to the cells, to present them with an essentially infinite sheet of tissue to migrate in. This would emulate the experience of microscopic cells in a surgical tissue graft measuring even a few centimeters across. For the reasons mentioned above, your model will be of more modest proportions. When dealing with a finite substrate, your model will require boundary conditions to prevent cells from running to the ends of fibers and jeopardizing the simulation when they can no longer move in a certain direction. Boundary conditions for this model can be handled in several ways. One straightforward method, and a good place to start, is to limit migration to a given region along the length (longitudinal axis) of the fibers. When a cell steps outside of this region (the Z-dimension of the active volume; see Figure 18.05), its random walk is biased to *push* it back inside. Natural boundaries exist for the depth of the scaffold, since cells begin their journeys on the bottom surface (and can't go any lower) and complete it upon reaching the top. Similarly, cells that reach either side of the scaffold width-wise will have nowhere to go but up, down, or back toward the center of the scaffold.

Spatial and temporal scales

You defined the spatial scale for this project back when you built your scaffold model: 1 Maya unit = 1 µm. Temporal scale is a little trickier. It equates one time increment (a frame) in Maya simulation time to a given number of seconds, minutes, or hours of living cell time. By basing the incremental displacements of the cell random walk on data from living cells, you can calculate what a time step is in Maya relative to a time step in vivo or in vitro (see Figure 18.06).

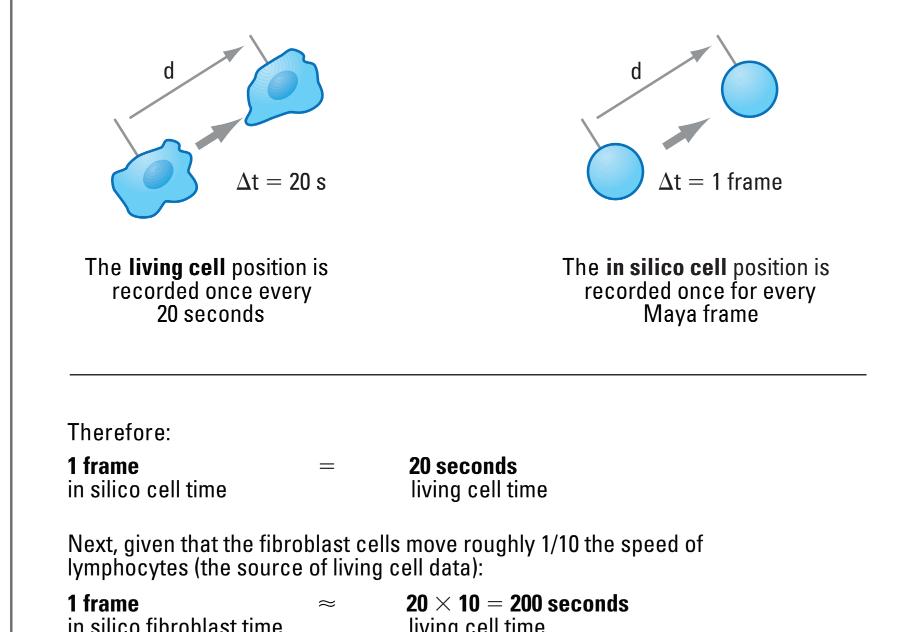


FIGURE 18.06

This diagram shows how we estimated the equivalence of a simulation time increment in Maya (1 frame) to the experimental 20 second increment in an in vitro cell migration study[2]. In the absence of appropriate data for fibroblast cells, we used an available dataset for lymphocytes. The lymphocyte migration time step was scaled to approximate an equivalence for the slower-moving fibroblasts.

Methods: Model design

The previous section outlined the different elements of this project and what it must do. Here we'll describe methods to make those elements and behaviors come to life in Maya. Let's start with the cell crawling algorithm which generates the random haptotaxis, which we'll call *haptotaxis 1*.

Haptotaxis 1: The random walk

To calculate the frame by frame displacement of a cell, you'll adapt a mathematical description of locomotion that was developed in a seminal study of migrating lymphocytes[2] on a 2D surface. Lymphocytes are highly motile white blood cells that play an important role in immunity. The authors of the influential study, Peter Noble and Martin Levine (N & L from here on) discovered that, in the absence of a chemoattractant, the motion of their lymphocytes closely resembled an unbiased random walk. They fit this observed behavior to a mathematical model called a **5-state Markov process** (named for Russian mathematician Andrey Andreyevich Markov; 1856–1922). A Markov process is a stochastic process for which the probability of future states for an item depends only on its present state and not its history[3]. In the case of a migrating cell, the 5 states are 5 possible choices for the cell's next position (Figure 18.07) which are based solely on its current position. Each state has a probability associated with it and varying those probabilities biases the random walk. N & L also recorded persistence times for each of the 5 states. You may recall that persistence is a cell migration term applied to the time duration between significant changes in direction.

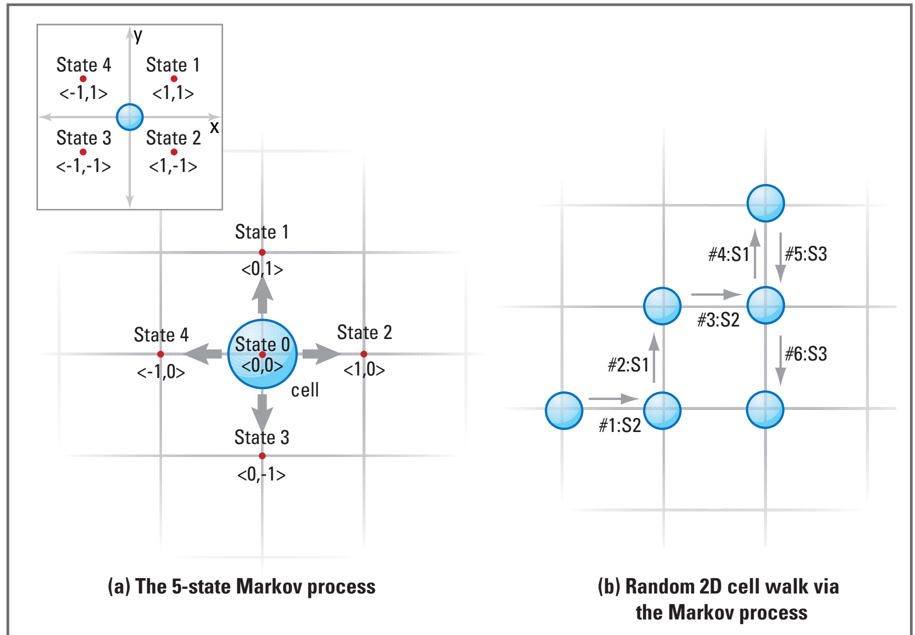


FIGURE 18.07

(a) The 5 states of the Markov process used by Noble and Levine (Ref. 3) to characterize the migration of lymphocyte cells. The original N & L Markov states (inset) have been rotated 90 degrees CCW to align with UV axes on NURBS surfaces in Maya.

(b) The 5 states are manifest as the displacement of the cell center.

State change probabilities

In their study, N & L recorded the displacement of cell center points at regular 20 second time intervals and then determined the probabilities associated with moving from any of the 5 states to any other state. A moving cell would either continue on its present course, that is, remain in its current state, or change its course, that is, change to a new state. These probabilities are presented in Table 18.01, in a form we call the **state change probability matrix**. The data show a distinct trend: if a cell in state 0 decides to change course, it has an approximately equal (25%) chance of choosing any of the other 4 states. However, if the cell is in one of states 1 through 4 and decides to change course, there is roughly an 80% chance that it will choose state 0 and between a 6% and 7% chance of choosing any of the remaining 3 states. The way that the Markov process describes the migration behavior, when a cell changes state, the cell must choose a state *other* than the one it's currently in.

Biasing the random walk

By adjusting the probability matrix you can favor motion in one direction over another—the modeling equivalent of a chemoattractant. For example, biasing motion for states one and three versus two and four in Figure 18.07 will affect the time that a cell spends moving in a vertical rather than a horizontal direction. In this project model, you'll bias the cell walks in terms of longitudinal versus circumferential motion on the cylindrical fibers.

Persistence time

The persistence, P, for a single mobile cell tells us how long a cell will persist in its current state (or continue on its present course). N & L measured P in intervals

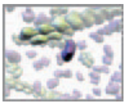


TABLE 18.01

The state change probability matrix for the 5-state Markov process.

Source: Reproduced from Noble and Levine[2].

State	0	1	2	3	4
0	0	0.800 ± 0.046	0.837 ± 0.054	0.841 ± 0.052	0.801 ± 0.042
1	0.254 ± 0.032	0	0.057 ± 0.017	0.046 ± 0.013	0.060 ± 0.037
2	0.223 ± 0.024	0.101 ± 0.021	0	0.057 ± 0.017	0.056 ± 0.015
3	0.266 ± 0.038	0.051 ± 0.016	0.064 ± 0.017	0	0.081 ± 0.015
4	0.254 ± 0.029	0.046 ± 0.024	0.039 ± 0.019	0.055 ± 0.036	0

20 seconds long. Their data show a mean persistence time of roughly one interval (or 20 seconds) for states one through four, and two intervals (or 40 seconds) for state 0—or standing still. Their statistical analysis found an exponential distribution of the recorded persistence times. To use the N & L persistence time data in your model, you'll therefore need an algorithm that simulates random draws from an exponential probability distribution. Here we'll use the formulation developed by the research physicist Daniel Gillespie as part of a more comprehensive theory of stochastic chemical reactions[4]. The full "Gillespie algorithm", though developed in the mid-1970s, has in the past few years come into its own as an essential tool of computational biologists. In the case of cell migration persistence time, the events would be changes in state—a start, a stop, or a change in direction. Gillespie's notation for the waiting time to each of those events is as follows:

In addition to Daniel Gillespie's seminal paper on stochastic chemical reactions[4], we also encourage you to see his book Markov Processes: *An Introduction for Physical Scientists* which we've listed in the *Further reading* section.

tau = (1/a)ln(1/r1)

where **tau** is the waiting time to the next event, measured in appropriate time increments such as 20 second frames.

1/a is the mean value of the exponential distribution of waiting times.

r1 is a random number drawn from the uniform distribution in the unit interval (i.e. between 0 and 1). You'll use Maya's rand() function to generate this.

ln(1/r1) is the natural logarithm of **1/r1.**

For this project we'll call **tau** the persistence time spent in the current state and 1/a the mean persistence time for that state.

Figure 18.08 shows our Markov process flowchart for the cell migration described by N & L. This will form the basis for your random walk algorithm.

Adapting for cell type

Since Nobel and Levine studied lymphocytes, you'll need to make a few adjustments with regards to the cell type in this project, that is: scale the lymphocyte time interval and mean persistence times to values more suitable to the slower-moving fibroblast cells in your model. Given that lymphocytes move roughly an order of magnitude

Start

Is the cell persisting?

yes

no

Draw a **random number**

Compare the random number to the probability matrix to determine the **new state** of the cell.

increment the time (frame += 1)

The **new state** becomes the **current state**.

Calculate the **new persistence time** (tau)

Move the cell to its **new position**

no

Simulation finished?

yes

End

FIGURE 18.08

Flowchart for a Markov process describing cell migration.

faster than fibroblasts, you can begin by multiplying both the time interval and mean persistence times by 10 as shown in Table 18.02. While these conversions are a gross simplification, they give you a starting point from which to incorporate living cell data into your in silico model. Moreover, although the mean speeds of the two cell types can be compared—allowing you to estimate the speed of one cell type based on the speed of another—there exists (at the time this book went to press) no analysis of state change probabilities and waiting times for fibroblasts comparable to the extremely thorough one published by N & L for lymphocytes at the level of single cell behavior. In the absence of such quantitative data you can reasonably make the modeler's assumption that a fibroblast would fit a Markov approximation of motion as well as a Noble and Levine lymphocyte does, that is, until it's proven otherwise.

Model-data time equivalence

The resolution of N & L's model is 20 seconds, meaning that no data exists in their compilation for time increments shorter than 20 seconds. It makes sense therefore to equate one frame (the unit time interval in Maya) with the 20 second unit time interval from N & L. The far right column in Table 18.02 lists the Maya frame equivalents you'll use in this project.

Leaving flatland: Moving from planes to fibers

Despite the gap between the worlds of 2D and 3D cell research, the movement of cells in a complex 3D environment can be thought of in some projects as building on 2D behavior. That is to say that the interactions between cell and substrate are essentially surface interactions (Figure 18.09). The fact that the substrate twists and turns throughout three dimensions only means that the emergent cell trajectory will be truly 3D. The degree to which we can treat 3D scaffolds as 2D surfaces depends of course on the tissue being modeled. An in vitro tissue preparation of fine collagen fibrils—which forms the substrate for many cell migration studies—for example, would not be amenable to such treatment. For projects involving scaffold structures such as dermis and cancellous bone that are large relative to the cells however, you can effectively treat the 3D environment as a construct built from 2D surfaces.

Such a treatment of course allows you to readily adapt the mathematics of 2D cell migration, such as the Noble and Levine methods described above, to a 3D scaffold model composed of surfaces. There still remains the interesting question of how to take 2D position data—like that generated by the random walk algorithm—and map it onto the 3D model surfaces, as shown in Figure 18.09. After all, we're not talking about simple planar motion like that generated by the single cell model in *Chapter 16*.

The solution to this problem lies in querying the 2D surface coordinates of the fiber object. We've mentioned the term *parameter* several times now with respect to NURBS curves and surfaces. Parameters are values in UV coordinate space that lie on a NURBS surface. For a Maya extrude node, which is the fiber surface geometry node, the U-axis corresponds the circumferential direction and the V-axis to the longitudinal direction. *Normalized* parameters have floating point values between 0 and 1, meaning you can query any point on a surface using a vector whose components span the range 0–1. For example, the parameter vector $\ll 0, 0.5 \gg$ is a point on the fiber surface seam ($u = 0$) and half-way ($v = 0.5$) along the length of the fiber (Figure 18.10).

	Cell type		
	Lymphocyte	Fibroblast	
Units	seconds	Seconds	Maya frames
Time interval	20	200	1
Persistence (state 0)	40	400	2
Persistence (states 1–4)	20	200	1

TABLE 18.02

Extrapolating data from a study of lymphocyte locomotion (Ref. 3) to slower-moving fibroblast cells, and the equivalent times in Maya frames.

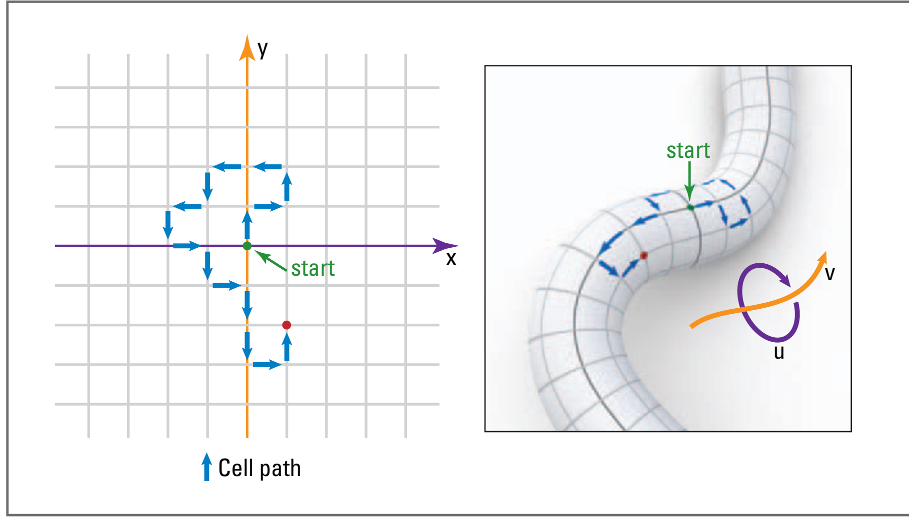
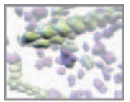


FIGURE 18.09

When fiber bundles are large relative to cells, fibers can be treated as 2D surfaces that meander through 3D space. A displacement generated by a 2D random walk algorithm can therefore be mapped onto a 3D surface as shown here. Every step in the XY plane has a corresponding step in the UV space of the fiber's surface.

With the help of two key nodes, you can find the world space location of a parameter vector on a given fiber surface. In contrast to a procedure you might code yourself, starting only from data on cell positions, sizes, and fiber axes and diameters, the MEL code you'll write to produce this essential information will be extremely concise. These nodes are the `closestPointOnSurface` (cpos for short) node and the `pointOnSurfaceInfo` (posi for short) node that were created and connected to each of the extrude (`fiberSurface`) nodes in the previous project. cpos and posi are represented schematically in Figure 18.10. Figure 18.11a and 11b shows them represented in the Attribute Editor. cpos takes as input connections the name of a surface in your scene and an XYZ point in world space (Figure 18.10). It returns the UV parameter values and the corresponding world space XYZ position lying on the input surface that is closest in distance to the input point. posi also takes a surface name as an input connection, along with U and V parameter values. It returns the world space position, the **unit surface normal** vector, and the tangent vectors of the input surface that correspond to the input UVs.



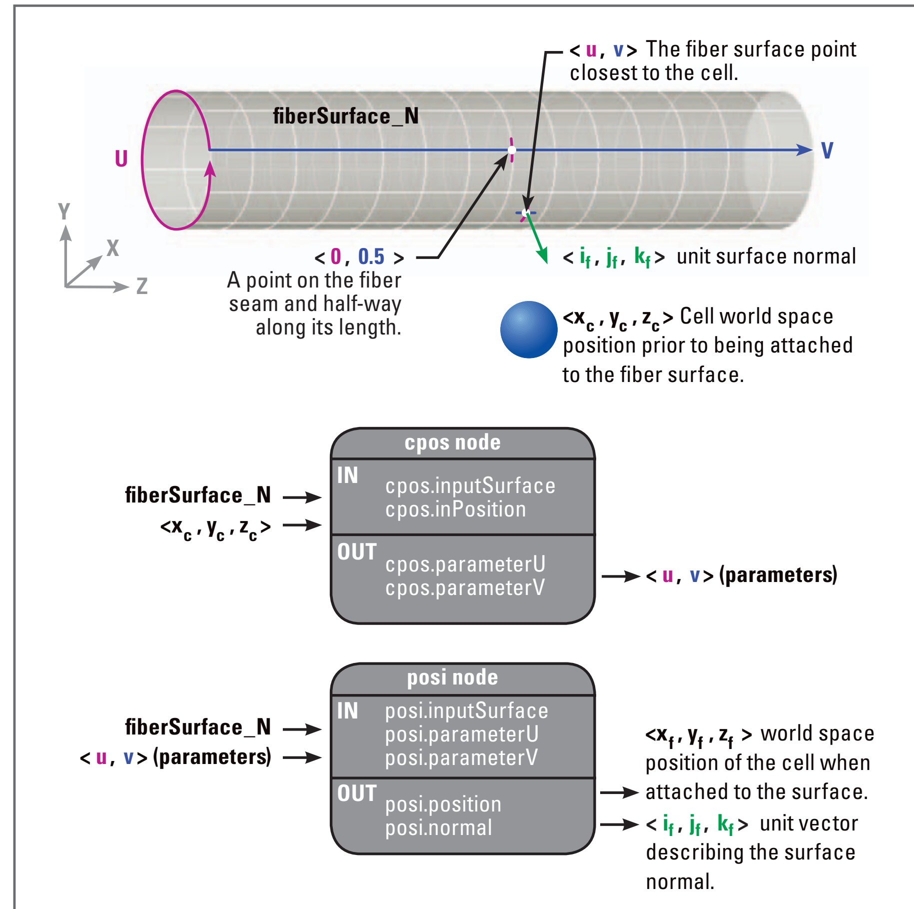


FIGURE 18.10

Parameters describe a point on the surface of a Maya object like the cylinder (fiberSurface) pictured here. The closest point on the fiberSurface to the cell's position is determined using the closestPointOnSurface (cpos) node. The world space (XYZ) position of the surface point is then given by the pointOnSurfaceInfo (posi) node and used to "attach" the cell to the surface.

By describing the location of a cell in terms of the UV coordinates of the fiber to which it's *attached*, you can query its world space position using the posi node and use that position to set the cell's translate attributes so that it's effectively *attached* to the fiber (Figure 18.10). The surface normal vector at the point of attachment is used to determine the cell's position perpendicular to the fiber surface. When the cell changes position as a result of the random walk algorithm (Markov process) and contact avoidance, the movement will be expressed as a change in the UV coordinates. Again, using the posi node, you will set the cell's translate attributes so that the cell moves to the appropriate world space position. In the next section, you'll see how to detect cell-fiber contacts and execute fiber transfers using the cpos node.

Haptotaxis 2: **Transferring between fibers**

The cpos nodes in your scaffold model can be queried after each time increment to determine if a cell is in contact with any of the fibers, that is other than the fiber

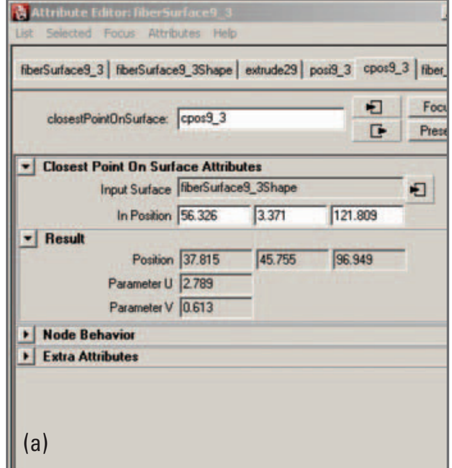


FIGURE 18.11

The Attribute Editor displays input and return attributes of

(a) the closesPointOnSurface node and

(b) the pointOnSurfaceInfo node.

to which the cell is currently attached. The cell's current translate X, Y, and Z values provide the input position for a given cpos node and the node returns the world space position of a point, the *return point*, on the corresponding fiber that is closer to the input than all other points on the fiber. If the distance (a scalar value) between the input position and the return point is less than the cell's contact radius, then the cell is in contact with the fiber surface. The decision to transfer is determined probabalistically by using Maya's `rand()` function (Figure 18.12). If the cell is in contact with more than one *new* fiber, then the contact points on these fibers are compared with one another based on the probability of choosing a contact point that lies in one direction relative to the cell, say positive Y, over a contact point in another direction. Once a cell has evaluated a potential transfer, it waits for a period of time (several frames at least) before checking again for a transfer. This prevents the cell from bouncing back and forth between two or more fibers with which it's in contact.

While we're discussing surface coordinates as *two*-element vectors (i.e. <u, v>), it's important to note that vector notation in Maya is always in the form of a *three*-element vector (i.e. <x, y, z>). If you enter two components for a vector, Maya will add a Z-element of zero value and return a three-element vector, as in the following example:

```
vector $tmpVect = <<1, 2>>;
// Result: <<1, 2, 0>> //
```

To avoid confusion and possible errors when building the script for this project, UV vectors are written as <u, v, 0> in the text.

Chemotaxis: A directional bias

Setting the directional probabilities for the inter-fiber transfer rules also lets you simulate the effects of a chemoattractant gradient. For example, suppose you want to simulate a gradient of chemoattractants that increases in strength in the positive X direction. When evaluating contacts with other fibers, you could then set a high probability (say 0.9 out of 1) so your cell will transfer when in contact with a point on

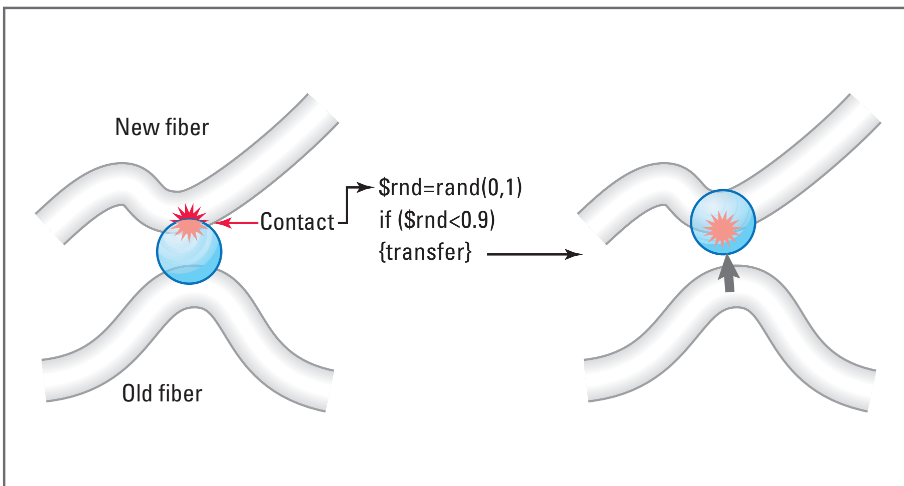


FIGURE 18.12

Regulating the probability of transferring between fibers based on direction is one way to model the effect of a chemoattractant gradient. In this example, the cell has a 90% probability of transferring to a fiber above it. The outcome is determined by drawing a random number ($rnd) and comparing it to the transfer probability.

a fiber that has an X-value greater than the cell's `translateX` value. Conversely a probability of 0.1 for transferring in the opposite direction makes it very unlikely that cells will move to fibers in the negative X-direction. In this project, you'll simulate a chemoattractant gradient to influence migration in the positive Y-direction, in order to promote infiltration of the scaffold by the cells which begin on its bottom (or inferior) surface.

Another way to effect chemoattraction in your model is to set the Markov state probabilities so that with each random walk step the cells tend in the direction of increasing chemoattractant concentration. We have not included such an effect in this project in order to keep the mathematics simple and the code concise. Nonetheless, we encourage you to explore the effects of biasing the Markov process one you have the basic model functioning.

Cell-cell signaling: **Contact avoidance**

For contact avoidance between cells you'll re-employ the method used to keep your fiber-axis seeds from colliding in the scaffold modeling algorithm. After every time increment, the world space position of each cell will be compared with that of every other cell in the scene. If the separation distance for a given pair of cells is less than a threshold value, an avoidance vector will be calculated and added to the UV fiber surface position of the first cell, effectively moving it away from its neighbor. To keep the calculations simple, the avoidance vector will effect a change in only the V-component of the cell's position on its fiber (Figure 18.13). You can extend this method later, if you wish, to include the U-component as well.

Before a model cell's MEL rules evaluate opportunities to transfer between fibers, the results of the other migration events—random walk, collision avoidance, and boundary conditions—are summed in a vector called `$deltaUVs`, which represent both the

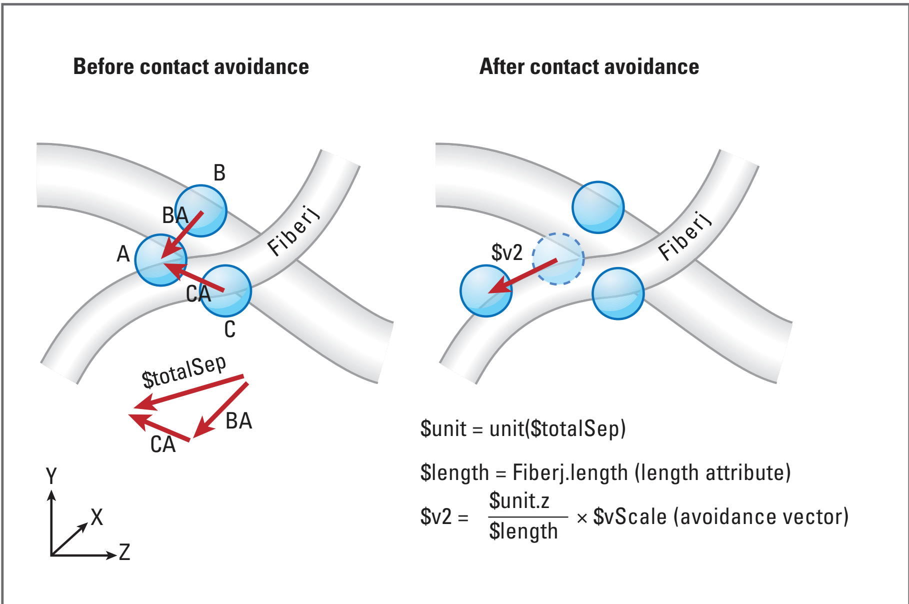


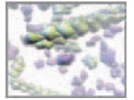
FIGURE 18.13

Cells avoid contact with one another by moving apart in the fiber surface V direction when they're within a threshold distance of one another.

cell's current heading and its speed (displacement over one time step, or frame). The contribution of the random walk algorithm to $deltaUVs is a unit vector, having a magnitude of 1 µm in this initial model no matter what its direction (refer back to Figure 18.07. So far we've been concerned only with which direction a cell chooses, but not the actual magnitude of its displacement per unit time. Likewise, the N & L process is concerned with direction and persistence, but not distance (explicitly). In your model, a 1 µm step in 200 seconds (see Table 18.02) equates to an instantaneous cell speed of 0.3 µm/min, which is well within the observed range of speeds for fibroblast cells. However, given that your cell will not be in constant motion due to periods in state 0, its mean speed over the course of a simulation run is bound to be much slower than 0.3 µm/min. Multiplying the random walk unit vector ($deltaUVs) by a scaling factor ($vScale) lets you set the incremental displacement, which in effect is the instantaneous speed of the cells. By using a custom attribute on the control widget (which was used to make the scaffold geometry) to set $vScale, you can easily scale the incremental displacement of the Markov process in the Channel Box—even while the simulation runs. For starters you'll set $vScale to a constant value but you can easily randomize it later on so that a cell moves at different speeds throughout its journey. Determining the mean speed of a cell is a matter for statistical analysis of its entire trajectory over the course of the simulation.

Summing the parts

Figure 18.14 shows how the different migration events work together to translocate a cell in one time increment. We've discussed methods for handling these events. Next you'll organize those methods logically into expressions and procedures, and



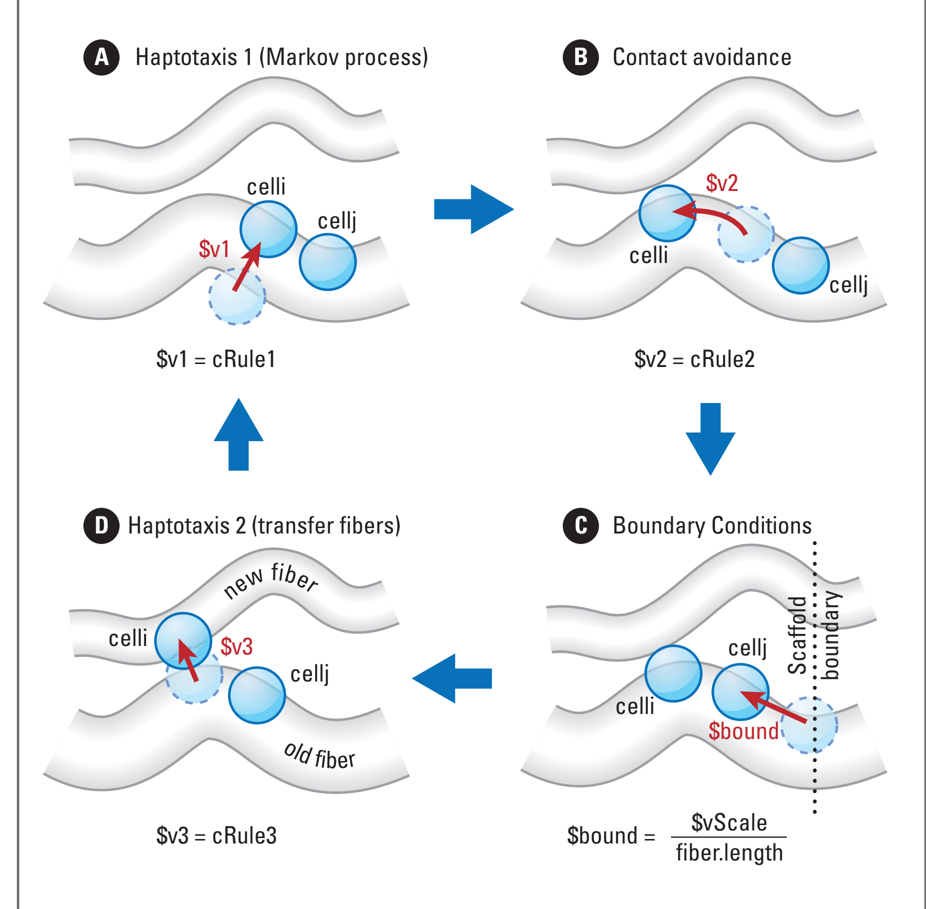


FIGURE 18.14

The translocation of cell i in one time step is the result of up to four locomotion events (labeled A, B, C, and D) in sequence. When the time step increments, the four events are evaluated again, and so on. The effect of boundary conditions (C) is shown here for cell j, a neighbor of cell i.

then encode them for Maya. Ultimately, you're aiming for a cell migration simulation that is entirely created and driven by MEL scripts. You fulfilled the first requirement by building the scaffold model entirely using MEL. Now it's time to encode the migration model.

Methods: Encoding the algorithm

Since the state of each cell is to be updated at regular time increments, the update process—reading the current state of a cell and its surroundings, evaluating the rules, and then updating the cell state—is well suited to the expression structure in Maya. You saw a similar process at work in the previous chapter, where an expression named moveSeeds updated the position of one fiber seed at a time. As well, the modular design, in which rules are parceled out as procedures to be called when needed by the main expression, makes for an extensible model that you can easily modify with additional procedures. Figure 18.15 illustrates how you'll implement the simulation model using expressions and procedures. If you wish to start using the

Start

frame = 1 ?

no

yes

resetCells expression

- return cells to their initial positions
- initialize global variables

createCells procedure
user inputs:
- cell count
- cell radius
- length of boundary zone (buffer)

moveCells expression

- call **rule 1**
- call **rule 2**
- update the cell's position
- if the cell is has strayed out of the scaffold's active area, nudge it back inside
- call **rule 3**
- update the cell's position
- if recording the simulation for playback, keyframe the cell's position

rule 1
random walk
(5 state Markov process)

rule 2
avoid contact with other cells

rule 3
transfer from one fiber to another

end of playback range?

no

yes

End

FIGURE 18.15

Flowchart for the cell migration model.

ready-made model, you can skip forward to the section entitled *Load the script files* on page 568. On the CD-ROM we've included a scene file containing the scaffold and cells along with the `resetCells` and `moveCells` expressions. You'll need only to source the three-rule procedure files to begin the simulation.

18_Cell_Migration/scenes/cellMigration.ma

18_Cell_Migration/MEL /cRule1.mel
/cRule2.mel
/cRule3.mel

As in previous chapters, we recommend that you compose your MEL scripts in a text editor (other than Maya's Script Editor), saving them periodically as you follow along with the instructions below. When you want to test bits of code in Maya, just copy and paste them from your text editor into Maya's Script Editor (for procedures) or Expression Editor (for expressions). You will save each procedure and expression in a separate text file (e.g. `makeCells.mel`) within your Maya Scripts directory.

The makeCells() procedure

This procedure makes the cell objects and positions them randomly on the bottom surface of the scaffold. You may want to open your scaffold scene file in Maya in order to test and debug this procedure as you build it. A ready-made scene file is available on the CD-ROM:

18_Cell_Migration/scenes/scaffold.ma

Some header information will be helpful when you refer back to this expression at a later date and to help others understand your code.

A commented description of each variable will be given only the first time it appears here in the text.

```
/* Description:
This procedure makes nurbs spheres called cells and initializes
their positions within a scene having a prebuilt NURBS fiber
scaffold for the cells to migrate in.

The procedure arguments are as follows:

$cellCount     The number of cells to be made.
$cellRadius    The radius, in micrometers, of each cell.
$buffer        The length, in Z, of the buffer zones used to
               contain the cells within the active area.
*/

global proc makeCells(float $cellCount, float $cellRadius, float
  $buffer) {
```

Next, declare and initialize the main variables. The global variables are ones that will be used by different expressions and procedures in the simulation.

```
/*****DECLARE THE VARIABLES*****/

/*
$cubeSize  The length, in Z, of the active scaffold volume.
*/
global float $cubeSize;
```

```
/*
$cposNames[]   An array of closestPointOnSurface node names.
$posiNames[]   An array of pointOnSurfaceInfo node names.
*/
global string $cposNames[], $posiNames[];

/*
$x, $y, $z     Vector components used to position the cell.
$dist          Used in detecting the closest fiber to the cell.
$shortestDist  Same as above.
$u, $v         Fiber surface parameter values.
$offsetDist    The distance each cell center is offset from its
               fiber.
$offset        The percentage of cell radius by which the cell
               center is offset from its fiber surface.

*/
float $x, $y, $z, $dist, $shortestDist, $u, $v, $offsetDist,
  $offset;

/*
$pos           A cell's current position.
$cpos          The closestPointOnSurface position attribute
               value.
$unitNorm      The unit surface normal vector.
$offsetVect    A vector used to offset the cell from the fiber
               surface.
*/
vector $pos, $cpos, $unitNorm, $offsetVect;

/*
$cellName      Used to name each cell.
*/
string $cellName;

/*
$fiberCount    The number of fibers in the scene.
$i, $j, and $k Counters.
*/
int $fiberCount, $i, $j, $k;

/***** INITIALIZE THE VARIABLES *****/

$cubeSize = `getAttr widget.cubeSize`;
$buffer = 25;
$cposNames =`ls -long "cpos*"`;
$fiberCount = size($cposNames);
$posiNames = `ls -l "posi*"`;
$offset = `getAttr widget.offset`; // A percentage of the cell
  radius.
```

As a precaution, delete the expressions from the scaffold model if they're still in your scaffold scene. If they're left in the scene, they will interfere with the migration simulation.

```
/***** MAIN BODY *****/
// Delete resetSeeds and moveSeeds if they exist.
if (`objExists "resetSeeds"`) delete resetSeeds;
if (`objExists "moveSeeds"`) delete moveSeeds;
```

Make the cells

Next you'll make a for loop to create the cells. In the case of the first cell you'll use the sphere command with its default settings except for the radius. Here's the first part of the for loop.

```
// Create the cells.
for ($i=0; $i<$cellCount; $i++) {

    if ($i == 0){

        $cellName = "cell_" + $i;

        // Create a nurbs sphere using default settings except
          for radius.
        sphere -r $cellRadius -n $cellName;
```

Add three custom attributes to the cell to store its initial position, the index number of its initial fiber (which is called `surfaceInit` and will be determined subsequently), and the index of its current fiber, called `closestSurface`. At the start of the simulation (frame 1) `closestSurface` and `surfaceInit` are equal and will remain so until the cell transfers to another fiber. Remember that when adding a vector attribute to a Maya node, you must explicitly add the vector elements (X, Y, and Z) using the parent flag. The attributes `surfaceInit` and `closestSurface` are used to store integers and are therefore created with the `-attributeType` flag with a value of `long`, which indicates a 32-bit integer. Using the surface index number will let you query any of the related nodes: `fiberSurface`; `closestPointOnSuface`; `pointOnSurfaceInfo`; or `fiberAxis`.

```
        // Add a custom attribute to store the cell's initial
          position.
        addAttr -longName posInit -attributeType double3
          $cellName;
        addAttr -ln posInitX -at double -parent posInit $cellName;
        addAttr -ln posInitY -at double -p posInit $cellName;
        addAttr -ln posInitZ -at double -p posInit $cellName;

        // Make posInit keyable.
        setAttr -e -keyable true ($cellName + ".posInit");
        setAttr -e -keyable true ($cellName + ".posInitX");
        setAttr -e -keyable true ($cellName + ".posInitY");
        setAttr -e -keyable true ($cellName + ".posInitZ");

        // Add a custom attribute called surfaceInit.
        addAttr -longName surfaceInit -attributeType long
          $cellName;
        setAttr -e -keyable true ($cellName + ".surfaceInit");

        // Add a custom attribute called closestSurface.
        addAttr -longName closestSurface -at long $cellName;
        setAttr -e -keyable true ($cellName + ".closestSurface");

    } // End if.
```

For subsequent cells, you'll use the duplicate command, so that only one sphere shape node is added to your scene.

```
    else {
                $cellName = "cell_" + $i;
                duplicate cell_0;
        }
```

Next you'll position the cell in a plane on the bottom surface of the scaffold using the move command. Refer back to Figure 18.05 for the size and location of the plane.

```
// Position the cell.
$x = rand(0, $cubeSize);
$y = 0; // At the scaffold base.
$z = rand($buffer, $buffer + $cubeSize);
move $x $y $z $cellName;
$pos = <<$x, $y, $z>>;
```

Find the closest fiber

The following code finds the closest point on all of the fibers to the cell's position. This is the first instance where you'll employ the closestPointOnSurface nodes. A for loop is used to narrow down candidates for closest point. Each time through the loop the distance ($dist) from the cell to the current fiber's closest point (stored in $cpos) is compared to the value of $shortestDist. If $dist is less than $shortestDist, then $shortestDist is assigned the value of $dist and the loop increments. Each time $shortestDist is updated, $k stores the index number of the corresponding fiber. At the end of the loop, $k will be the number assigned to the cell attributes surfaceInit and closestSurface.

Reminder:

cpos is our shorthand notation for a **closestPointOnSurface** node. Given an input XYZ position, cpos returns the point in 3D space closest to the input that also lies on the surface in question.

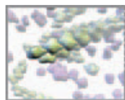
posi is our shorthand notation for a **pointOnSurfaceInfo** node. It takes the point returned by cpos and itself returns information (surface normal vector, and so on) about that point.

```
/*
Find the nearest fiber to the cell's position. Set
$shortestDist high enough to include all of fibers in the
first round of testing.
*/
$dist = 0;
$shortestDist = 10000;
$k = 0;

for ($j = 0; $j < $fiberCount; $j++) {

    // Get the closestPointOnSurface position for fiber $j.
    setAttr ($cposNames[$j] + ".inPosition") -type double3
      $x $y $z;
    $cpos =`getAttr ($cposNames[$j] + ".position")`;
    $dist = mag($cpos - $pos);

    // Is $dist the new $shortestDist?
    if ($dist < $shortestDist) {

        // Fiber $j is currently the closest surface.
        $closestSurface[$i] = $cposNames[$j];
        $shortestDist = $dist;

        // Store the closest fiber index in $k.
        $k = $j;
    }
} // End for loop.
// $k is the index for the closest fiber to cell $i.
```

Position the cell on the fiber

Next, the cpos and posi nodes will be used together to find the normal vector for the surface at the closest point to the cell. The cell will be positioned at the closest point and then offset from the fiber using the normal vector and the $offset value.

```
// Query the u and v surface coordinates from the cpos node.
$u =`getAttr ($cposNames[$k] + ".parameterU")`;
$v =`getAttr ($cposNames[$k] + ".parameterV")`;
```



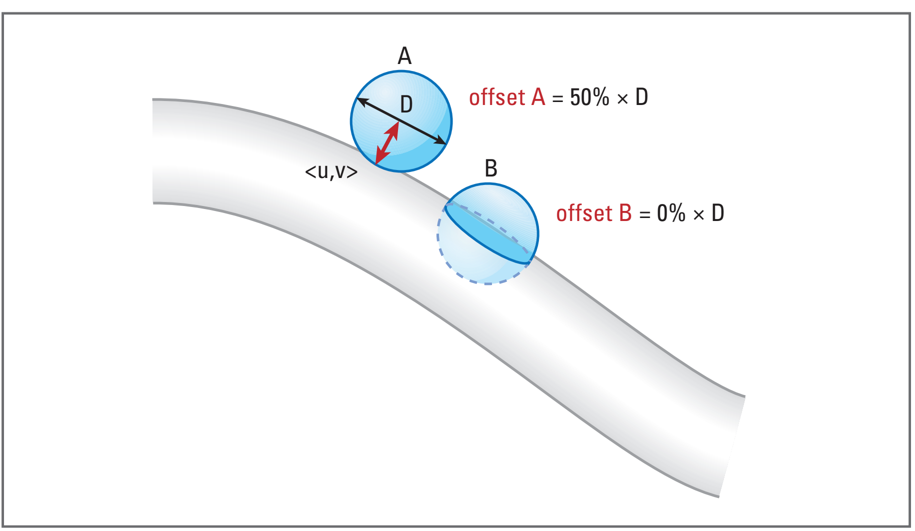


FIGURE 18.16

Offsetting the cell center from the fiber surface approximates cell spreading. The more a cell spreads on the surface, the shorter its reach perpendicular to the surface and therefore, the less likely it is to contact other fibers by chance. In this example, cell A has an offset equal to 50% of its diameter. Cell B is fully spread on the fiber, with an offset of 0.

```
// Input the parameters to the pointOnSurfaceInfo node.
setAttr ($posiNames[$k] + ".parameterU") $u;
setAttr ($posiNames[$k] + ".parameterV") $v;

// Query the surface normal vector.
$unitNorm = `getAttr ($posiNames[$k] + ".normal")`;

// Calculate the magnitude of the offset.
$offsetDist = $offset/100 * $cellRadius;
```

A minimum value for offset, 0%, would see the cell center positioned *on* the fiber surface. A maximum value of 100% would place the cell center at a distance equal to its radius away from the fiber surface (Figure 18.16). `$offset` takes its value from the offset attribute of the control widget, which you'll create later in this chapter.

```
// Calculate the offset vector.
$offsetVect = $unitNorm * $offsetDist;

// Calculate the vector to position the cell.
$cpos = `getAttr ($cposNames[$k] + ".position")`;
$pos = $cpos + $offsetVect;
```

Set the custom attributes

Here you'll set the cell's custom attributes which you'll query during the simulation. Vector attributes must be set with the `-type double3` flag.

```
// Set the cell's attributes.
setAttr ($cellName + ".translate") -type double3 ($pos.x)
  ($pos.y) ($pos.z);
setAttr ($cellName + ".posInit") -type double3 ($pos.x)
  ($pos.y) ($pos.z);

setAttr ($cellName + ".surfaceInit") $k;
setAttr ($cellName + ".closestSurface") $k;
```

```
    } // End the cell loop.

} // End procedure

// Print command line instructions for the user.
print "Call the procedure: makeCells(10, 5, 25)";
```

Save your file

Save your procedure as a file in your Maya Scripts directory.

1. **Query Maya's search path for the Scripts directory using the internalVar command. Enter the following in the Script Editor:**

   internalVar -userScriptDir

   In Windows, it will return a path such as:

   /Users/yourName/Library/Preferences/Autodesk/maya/8.0/scripts/

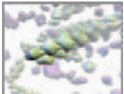
2. **Save your file under the name makeCells.mel.**

The resetCells expression

You made an expression similar to this one in each of the previous two chapters. Its purpose here is to declare and initialize global variables, return cells to their starting positions, and to reset the cells' `closestSurface` attributes. Once again, we'll recommend building the expression as it appears here rather than as one long string following the `expression` MEL command. If you prefer to do the latter, remember to escape all quotation marks and line breaks with the backward slash character, " \". Because this expression is to execute only at frame 1, it starts with a conditional time check. The `currentTime` command returns the same value as would a query of the global variable `frame`.

```
/* Description:
This is an animation expression that initializes global variables
and resets cells to their starting positions.
*/

if (`currentTime -query` == 1) {

    /***** DECLARE THE VARIABLES *****/

    /*
    $sProb[5][5]    The state change probability matrix for the
                    5-state Markov process.
    */
    global matrix $sProb[5][5];

    /*
    $cellPos[]      An array of current cell positions.
    $UVs[]          The current surface parameter position (UV) of
                    every cell.
    $deltaUVs[]     The last step taken by each cell in UV surface
                    coordinates.
    */
    global vector, $cellPos[], $UVs[], $deltaUVs[];
```

Unlike an array, whose length (the number of elements it contains) can be set to different values, a matrix must be declared with an explicit length and width which cannot change once declared.




```
/*
$cellNames[]        A list of cell names.
$surfaceNames[]     A list of scaffold fibers.
$cpos Names[]       A list of closestPointOnSurface node names.
$posiNames[]        A list of pointOnSurfaceInfo node names.
*/
global string $cellNames[], $surfaceNames[], $cposNames[],
  $posiNames[];

/*
$cDiameter          Cell diameter.
$probUp             The probability that a cell will transfer to
                    a fiber above it.
$minBound           The lower Z-coordinate value of the migration
                    boundary.
$maxBound           The upper Z-coordinate value of the migration
                    boundary.
*/
global float $cDiameter, $probUp, $minBound, $maxBound;

/*
$cellCount          The number of cells in the scene.
$crntState[]        The current Markov state for every cell.
$persist[]          The persistence time in the current state.
$transWait[]        The frame numbers at which corresponding
                    cells will once again checking for a possible
                    fiber transfer.
*/
global int $cellCount, $crntState[], $persist[], $transWait[];

/*
$cellPosName        The current closestPointOnSurface node name.
*/
string $cellPosName;

/*
$pos                Used to store single vectors from $cellPos[].
*/
vector $pos;

/*
$closest            The number of the fiber to which the current
                    cell is attached.
$i and $k           Counters.
*/
int $closest, $i, $k;
float $u, $v;

/***** INITIALIZE THE VARIABLES *****/

clear ($cellPos);
clear ($crntState);
clear ($persist);
clear ($transWait);
```

State	0	1	2	3	4
0	0	0.7	0.7	0.7	0.7
1	0.25	0	0.1	0.1	0.1
2	0.25	0.1	0	0.1	0.1
3	0.25	0.1	0.1	0	0.1
4	0.25	0.1	0.1	0.1	0

TABLE 18.03

The state change probability matrix, $sProb, for the 5 state Markov process.

```
clear ($UVs);
clear ($deltaUVs);
$cellNames = `ls -transforms "cell*"`;
$cellCount = `size $cellNames`;
$surfaceNames = `ls -transforms "fiberSurface*"`;
$cposNames = `ls -long "cpos*"`;
$posiNames = `ls -long "posi*"`;
$minBound = 25;
$maxBound = $minBound + `getAttr widget.cubeSize`;
$cDiameter = `getAttr widget.cDiameter`;
```

The state change probability matrix

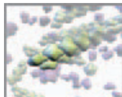
The state probability matrix regulates the directional choices in each cell's random walk. You'll initialize this matrix (represented by the variable $sProb) with values similar to those recorded by Noble and Levine[2] but with less weighting on the 0 state. Less time spent in state 0 means more time spent migrating, which allows you to see more movement in the model. Naturally the cells will travel farther in the scaffold over the course of a simulation if they spent less time standing still! Nonetheless, waiting time (time spent in state 0) is a natural component of cell migration and therefore one we chose to include in this version of the model. Table 18.03 shows the $sProb values in tabular form so that you can see how the columns and rows relate to the matrix notation below.

```
// Initialize the markov process state change probability matrix.
$sProb = <<0, 0.25, 0.25, 0.25, 0.25; 0.7, 0, 0.10, 0.10, 0.10;
  0.7, 0.10, 0, 0.10, 0.10; 0.7, 0.10, 0.10, 0, 0.10; 0.7, 0.10,
  0.10, 0.10, 0>>;
```

The chemoattractant

The probability value stored in $probUp is the analog of a chemoattractant gradient along the world space Y-axis. A $probUp value of 0 equates to no attraction. A value of 1 equates to a strong attraction. $probUp will be used in cRule 3 to evaluate fiber transferring.

```
// Set the chemotactic bias.
$probUp = 0.9;
```



Reset the cell's position and update its attributes

The for loop will cycle through the following set of instructions for each of the cells. $cCount stores the total number of cells in you scene.

```
/***** MAIN BODY *****/
for ($i=0 ; $i < $cellCount; $i++) {

    // Set the cell to its initial position using its posInit
      attribute.
    $pos =`getAttr ($cellNames[$i] + ".posInit")`;
    setAttr ($cellNames[$i] + ".translate") -type double3 ($pos.x)
      ($pos.y) ($pos.z);

    // Set keys to record cell position.
    setKeyframe -at "translateX" $cellNames[$i];
    setKeyframe -at "translateY" $cellNames[$i];
    setKeyframe -at "translateZ" $cellNames[$i];

    // Get the index number of the cell's initial surface.
    $closest =`getAttr ($cellNames[$i] + ".surfaceInit")`;
    $cellPosName = $cposNames[$closest];

    // Get the initial surface uv values.
    setAttr ($cellPosName + ".inPosition") -type double3 ($pos.x)
      ($pos.y) ($pos.z);
    $u =`getAttr ($cellPosName + ".parameterU")`;
    $v =`getAttr ($cellPosName + ".parameterV")`;

    // Initialize the global arrays.
    $cellPos[$i] = $pos;
    $UVs[$i] = <<$u, $v, 0>>;

    // Set the cell's closestSurface attribute.
    setAttr ($cellNames[$i] + ".closestSurface") $closest;
}

// Select the widget so the model parameters appear in the
  channel box.
select widget;

} // End expression.
```

Save your file

In its present form this expression cannot be sourced like a MEL script. Instead, you will copy and paste it into Maya's Expression Editor. For this reason we recommend not using the .mel file extension in order to avoid confusion with files that qualify as stand-alone MEL scripts. Save resetCells in a plain text file called resetCells.txt within your Maya Scripts directory.

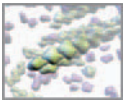
The moveCells() expression

This expression is the workhorse of the model. It calls the migration rule procedures and updates the state of each cell—UV and world space positions and the current fiber surface to which the cell is attached—each time the Maya scene enters a new frame.

```
/* Description:
This is an animation expression that updates cell positions using
vectors $v1, $v2, and $v3. The vectors are derived from the rule
procedures cRule1, cRule2, and cRule3.
*/

// Execute this expression only if the frame is greater than 1.
if (`currentTime -query` > 1) {

    /***** DECLARE THE VARIABLES *****/

    global vector $cellPos[], $UVs[], $deltaUVs[];
    global string $cellNames[];
    global float $cDiameter, $minBound, $maxBound;

    /*
    $frameCheckUsed to send the current frame number to the rule
      procedures.
    */
    global int $frameCheck, $cellCount, $transWait[];

    /*
    $surfacePos    The point of cell attachment on a fiber surface.
    $unitNorm      The unit vector of the surface normal at the
                   point of attachment.
    $v1            The vector returned from the random walk
                   procedure, rule1().
    $v2            The vector returned from the contact avoidance
                   procedure, rule2().
    $uvVect        The UV coordinates of the current cell.
    $newPos        The cell's new position after all three rules
                   and boundary conditions have been accounted for.

    */
    vector $surfacePos, $unitNorm, $v1, $v2, $uvVect, $newPos;

    /*
    $posiName      The current pointOnSurfaceInfo node.
    $surfaceName   The current fiber surface.
    $v3[]          The array array returned by the fiber transfer
                   procedure, rule3().
    */
    string $posiName, $surfaceName, $v3[];

    /*
    $vScale        Scales the cell's random walk step size.
    $bound         U parameter element of the vector used to nudge
                   cells back into the active region of the scaffold.
    */
    float $vScale, $bound, $cRadius, $u, $v, $offset, $offsetDist;
    int $closest, $i;
```

Remember that a global variable must be declared within each expression or procedure that uses it.

Multiplying the cell migration unit vector `$deltaUVs` by the scaling factor `$vScale` lets you set the incremental displacement, which in effect is the instantaneous speed of the cells.




```
/***** INITIALIZE THE VARIABLES *****/
$vScale = `getAttr widget.vScale`;
$offset = `getAttr widget.offset`;
$cRadius = $cDiameter/2;
$frameCheck = frame;
$buffer = 25;
$bound = 0;
```

Begin the main loop

The following code increments once for every cell in the population.

```
/***** MAIN BODY *****/
for ($i=0; $i < $cellCount; $i++) {
    // Get the pointOnSurfaceInfo (posi) and extrude (surface)
      node names.
    $closest = `getAttr ($cellNames[$i] + ".closestSurface")`;
      // e.g. "12".
    $posiName = $posiNames[$closest]; // e.g. "posi6_12".
    $surfaceName = $surfaceNames[$closest]; // e.g.
      fiberSurface6_12.
```

Invoke the first two rules of behavior

Continuing on in the main program loop, you'll now call rules 1 and 2 to calculate the results of the Markov process and contact avoidance, respectively.

```
    /***** RULE 1 *****/
    $v1 = (cRule1($i, $vScale, $surfaceName));
    // cRule1 returns the uv coordinates of the random walk step.
    /***** RULE 2 *****/
    $v2 = (cRule2($i, $vScale, $surfaceName));
    /* $cRule2 returns the V-component of the unit contact
    avoidance vector, multiplied by scaling factor $vScale. */
```

Next, you'll update the vector array `$deltaUVs[]` with the cell displacement as a result of rules 1 and 2. Let contact avoidance override the random walk. In other words, if the current cell is too close to another cell, its avoidance step (along the fiber's long axis) will override the V-component of its random walk. This prevents the cell from stepping away from its neighbor, only to step back toward it with a random walk step. Likewise, the cell will not take a double step if avoidance and the random walk are in the same direction.

Reminder: U and V surface components are represented by .x and .y components, respectively, in vector notation. There is no third component in UV space, so the .z vector component is given a default value of 0.

```
    // Update the $deltaUVs array with the results of rules 1
      and 2.
    if ($v2.y != 0) {
        /* The V-component of avoidance is non-zero, therefore
        the cell displacement will consist of the U-component of
        $v1 and the V-component of $v2. Because surface vectors
        are 2D, the third (Z) component in $uvVect will be 0. */
        $uvVect = <<$v1.x, $v2.y, 0>>;
        $deltaUVs[$i] = <<$v1.x, $v2.y, 0>>;
    }
    else {
        // There is no collision avoidance vector ($v2 = <<0,
          0, 0>>).
        $deltaUVs[$i] = $v1;
    }
```

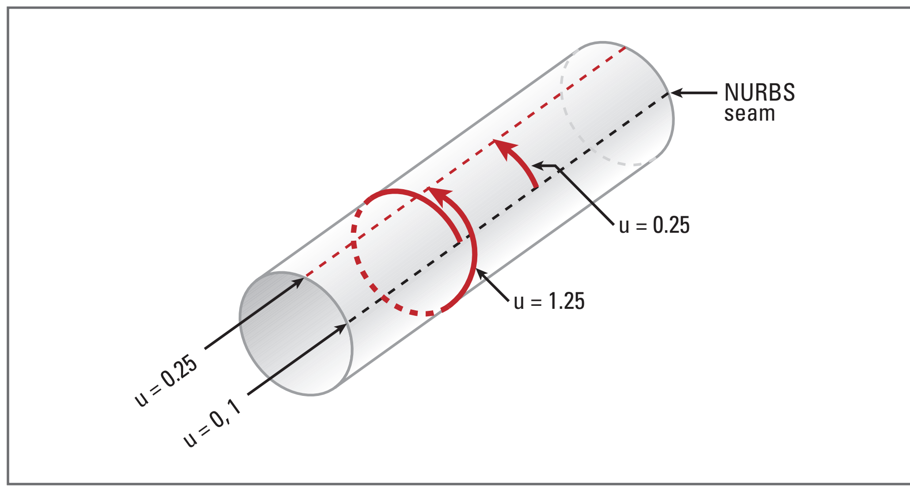


FIGURE 18.17

The seam of a NURBS object is both the origin and terminus of the U-axis. When a U-value is less than 0 or greater than 1, it must be normalized to a value that corresponds to an equivalent circumferential position on the fiber. In this example the U-value is 1.25. Its normalized equivalent value is 0.25.

```
// Add $deltaUVs[$i] to the cell's current position on the
  fiber surface.
$UVs[$i] += $deltaUVs[$i];

// Query the UV position for cell $i.
$uvVect = $UVs[$i];
```

Because fiber parameter values are normalized (spanning zero to one) you must adjust U-values that would otherwise move the cell past the fiber U-origin. For example, a U-value of 1.25 must be adjusted to 0.25 (Figure 18.17). In normalized parameter space, values above 1 and below 0 are taken to be 1 and 0, respectively.

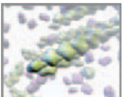
```
// Correct for circumferencial movement past the U-origin.
$u = $uvVect.x;
if ($u > 1) $u = $u - 1;
else if ($u < 0) $u = $u + 1;
```

Evaluate the boundary conditions

Now find out where the cell's new surface position lies in world space in order to test if the cell has stepped out of the imaginary bounding box shown in Figure 18.05.

```
// Get the worldspace coordinates of the attachment point.
setAttr ($posiName + ".parameterU") ($uvVect.x);
setAttr ($posiName + ".parameterV") ($uvVect.y);
$surfacePos = `getAttr ($posiName + ".position")`;
// This is the temporary new position of the cell.

/***** BOUNDARY CONDITIONS *****/

if ($surfacePos.z < $minBound || $surfacePos.z > $maxBound) {

    // Get the length of the cell's fiber.
    $length = `getAttr ($surfaceName + ".length")`;

    /* Calculate the step size in uv parameter space for a
    step back in the migration boundary. $bound will be double
    the random walk step size in order to not only halt the
    cell's progress outside of the box, but reverse it.*/
    $bound = 1.5 * $vScale / $length;
```

The || operator stands for the logical "or".




```
        // For a negative step.
        if ($surfacePos.z > $maxBound) $bound = -$bound;
        // Else, $bound will remain positive.

        // Update the $v component of $deltaUVs[$i].
        $deltaUVs[$i] += << 0, $bound, 0 >>;
        // The u value (X-component of the vector) remains
          unchanged.

        // Update $UVs[$i] and store it in a vector.
        $UVs[$i] += << 0, $bound, 0 >>;
        $uvVect = $UVs[$i];

        // Get the worldspace coordinates the attachment point.
        setAttr ($posiName + ".parameterU") ($uvVect.x);
        setAttr ($posiName + ".parameterV") ($uvVect.y);
        $surfacePos =`getAttr ($posiName + ".position")`;
    }
```

The temporary world space position

Before calling `cRule3` for fiber transferring, we'll update the world space cell position stored in `$cellPos[$i]` without actually moving the cell. You will move the cell after evaluating fiber transfers.

```
// Get the surface normal vector using posi.
$unitNorm =`getAttr ($posiName + ".normal")`;

// Determine the cell/fiber offset.
float $tmpFloat = $offset/100 * $cRadius;
$offsetVect = $unitNorm * $tmpFloat;

// Set the new cell position, offset from the new surface
  position.
$cellPos[$i] = $surfacePos + $offsetVect;
// This is the new position of the cell unless it tranfers
  fibers.
```

Invoke the third rule of behavior

Now check to see the cell has an opportunity to transfer from its current fiber to a new one, given its new world space position, `$cellPos[$i]`. The condition for calling the `cRule3` procedure is that the cell's `$transWait` time has expired and it is therefore free to check for potential transfers. After evaluating `cRule3`, a new `$transWait[$i]` is set within the procedure.

```
/***** RULE 3 *****
Call the fiber transfer procedure, rule 3, only if the
cell has waited long enough since it last checked for a
transfer.
*/
if (frame > $transWait[$i]) {

    $v3 = (cRule3($i));
    /*
    cRule3 returns the array:
    {"yes or no", "closestSurface", "<<$x, $y, $z>>",
      "<<$u, $y, 0>>"}.
    If no transition occurs, cRule3 will return "no" and
      null values for $closestSurface and the two vectors.
```

```
        */
        if ($v3[0] == "yes") { // A transfer has occurred.

            // Update variables with the new information from
              rule3.

            // The fiber's number:
            $closest = (int) $v3[1];

            // Update the cell's closestSurface attribute.
            setAttr ($cellNames[$i] + ".closestSurface") $closest;

            // The cell attachment position in world space:
            $surfacePos = $v3[2];
            // The cell attachment position in uv space:
            $UVs[$i] = $v3[3];

            // Get the new posi node.
            $posiName = $posiNames[$closest];

            // Get the surface normal vector using posi.
            $uvVect = $UVs[$i];
            setAttr ($posiName + ".parameterU") ($uvVect.x);
            setAttr ($posiName + ".parameterV") ($uvVect.y);
            $unitNorm = `getAttr ($posiName + ".normal")`;

            // Determine the cell/fiber offset.
            $offsetDist = $offset/100 * $cRadius;
            $offsetVect = $unitNorm * $offsetDist;

            // Set the new cell position.
            $cellPos[$i] = $v3[2] + $offsetVect;

        } // End if ($v3[0] == "yes").

    } // End if (frame > $transWait[$i]).
```

Move the cell into position

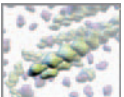
Until now, the expression has dealt with position data stored in attributes and arrays, but has not physically moved the cell within the scaffold—i.e. Maya's scene graph has not changed. Use the familiar `setAttr` command to set the cell's transform attributes and thereby place it where it belongs in world space.

```
// Set the cell's attributes.
$newPos = $cellPos[$i];
setAttr ($cellNames[$i] + ".translate") -type double3
  ($newPos.x) ($newPos.y) ($newPos.z);
```

Keyframe the cell's position

The following lines set keyframes for the cell's translate attributes. The code is commented out (using the "//" notation) for now. When your model is functioning to your satisfaction and you wish to make a record of a simulation, remove the comments.

```
/* Set keys to record cell position. Remove the forward
   slashes to make these commands active. */
// setKeyframe -at translateX $cellNames[$i];
```




```
        // setKeyframe -at translateY $cellNames[$i];
        // setKeyframe -at translateZ $cellNames[$i];
    } // End the main loop.
} // End the expression.
```

Save your file

Save moveCells in a plain text file called moveCells.txt within your Maya Scripts directory. Next, you'll compose the three-rule procedures.

cRule1(): The random walk

This procedure returns a vector in UV space—the result of the 5-state Markov process—and calculates the persistence time until the next state change. The state change probabilities are defined by the $sProb matrix in the expression called resetCells. This procedure takes three arguments: the current cell index, $i; the float $vScale, which is used to scale the magnitude of the random walk step size; and the name of the surface, $surfaceName, to which cell $i is currently attached.

```
/* Description:
This procedure calculates a random walk based on a 5-state
Markov process and corresponding in-state waiting times.

The procedure arguments are as follows:
$i                  The index number of the current cell.
$vScale             Scales the cell's random walk step size.
$surfaceName        The surface to which the current cell is
                    attached.
*/

global proc vector cRule1 (int $i, float $vScale, string
  $surfaceName) {

    /***** DECLARE THE VARIABLES *****/

    global matrix $sProb[5][5];
    global vector $UVs[], $deltaUVs[];
    global int $crntState[], $frameCheck, $persist[];

    /*
    $states[]       The unit vector equivalents of the 5 Markov
                    states.
    */
    vector $states[], $surfacePos, $v1;

    /*
    $length         The length of the fiber, $surfaceName.
    $pi             The ratio of the circumference of a circle to
                    its diameter.
    $fDiameter      The diameter of the fiber $surfaceName.
    $u and $v       The cell's stepwise displacements in UV
                    parameter space.
    $rnd            A number returned by Maya's rand() function.
    $pAvg           The average persistence time in frames.
    $tau            The new persistence time calculated using the
                    Gillespie equation.
    */
    float $length, $pi, $fDiameter, $u, $v, $rnd, $pAvg, $tau;
```

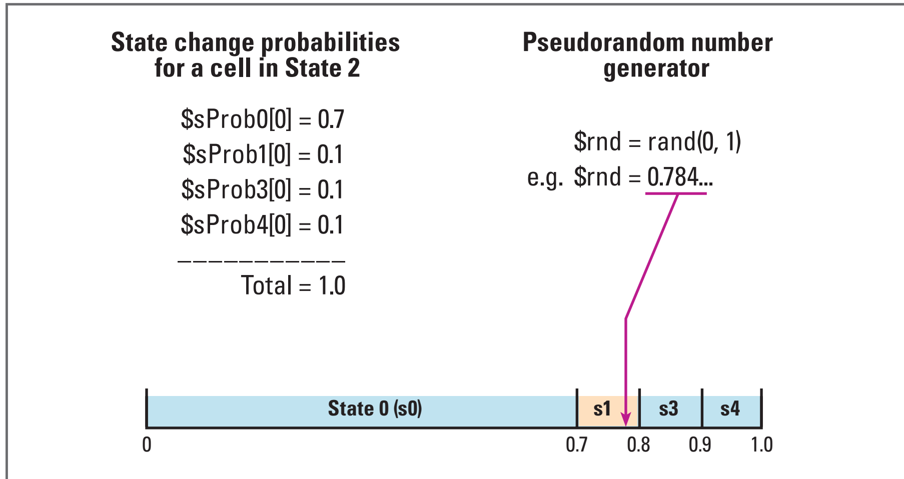


FIGURE 18.18

The unit interval is broken into segments that reflect the state change probabilities (shown here for a current cell waking state of 2). The segment into which the pseudorandom number, $rnd, falls determines which new state the cell chooses—state 1 in this example.

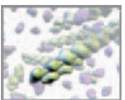
```
/***** INITIALIZE THE VARIABLES *****/

$states = {<<0,0,0>>, <<0,1,0>>, <<1,0,0>>, <<0,-1,0>>,
  <<-1,0,0>>};
$length = `getAttr ($surfaceName + ".length")`;
$fDiameter = `getAttr ($surfaceName + ".diameter")`;
$pi  = 3.14159;
```

State change probabilities

In this next section, you'll simulate a random draw from a probability distribution and use it to determine the new state of the cell. Because the probabilities vary depending on which state the cell is currently in, a separate conditional (`if`) statement will handle the evaluation for each of the five states. Enable the probability as follows: draw a pseudorandom number on the unit interval (i.e. between 0 and 1) using MEL's `rand()` function and compare it with the probability for each state (Figure 18.18). To conserve space, we have listed the conditional statements for states 0, 1, and 2 below. By applying the logic outlined in Figure 18.18, you can write the code for the remaining three states.

```
/***** MAIN BODY *****/

/* Determine the new state of the cell by comparing the
random number, $rnd with the state change probability matrix
$sProb[5][5] that was initialized in moveseeds. */

if ($persist[$i] <= $frameCheck) { // time to pick a new state.

    // Pick a pseudorandom number on the unit interval (0 to 1).
      $rnd = rand(0,1);

    // STATE 0.
    if ($crntState[$i] == 0) {

        // Pick any of states 1 through 4.

        if (($rnd > 0) && ($rnd <= $sProb[0][1])) {
            // Pick state 1.
            $crntState[$i] = 1; $v1 = $states[1];
```

Because the algorithm of rand() and functions like it do not perfectly reproduce all the properties of randomness, the numbers they generate often are called **pseudorandom numbers** instead of random numbers.




```
    }
    else if (($rnd > $sProb[0][1]) && ($rnd <= ($sProb[1][0] +
      $sProb[0][2]))) {
        // Pick state 2.
        $crntState[$i] = 2; $v1 = $states[2];
    }
    else if (($rnd > ($sProb[0][1] + $sProb[0][2])) && ($rnd <=
      (1 - $sProb[0][4]))) {
        // Pick state 3.
        $crntState[$i] = 3; $v1 = $states[3];
    }
    else if (($rnd > (1 - $sProb[0][4])) && ($rnd <= 1)) {
        // Pick state 4.
        $crntState[$i] = 4; $v1 = $states[4];
    }
}

// STATE 1.
else if ($crntState[$i] == 1) {

    // Pick one of states 0, 2, 3, OR 4.

    if (($rnd > 0) && ($rnd <= $sProb[1][0])) {
        // Pick state 0.
        $crntState[$i] = 0; $v1 = $states[0];
    }
    else if (($rnd > $sProb[1][0]) && ($rnd <= ($sProb[1][0] +
      $sProb[1][2]))) {
        // Pick state 2.
        $crntState[$i] = 2; $v1 = $states[2];
    }
    else if (($rnd > ($sProb[1][0] + $sProb[1][2])) && ($rnd <=
      (1 - $sProb[1][4]))) {
        // Pick state 3.
        $crntState[$i] = 3; $v1 = $states[3];
    }
    else if (($rnd > (1 - $sProb[1][4])) && ($rnd <= 1)) {
        // Pick state 4.
        $crntState[$i] = 4; $v1 = $states[4];
    }
}

// STATE 2.
else if ($crntState[$i] ==  2) {

    // Pick one of states 0, 1, 3, OR 4.

    if (($rnd > 0) && ($rnd <= $sProb[2][0])) {
        // Pick state 0.
        $crntState[$i] = 0; $v1 = $states[0];
    }
    else if (($rnd > $sProb[2][0]) && ($rnd <= ($sProb[2][0] +
      $sProb[2][1]))) {
        // Pick state 1.
        $crntState[$i] = 1; $v1 = $states[1];
    }
```

```
        else if (($rnd > ($sProb[2][0] + $sProb[2][1])) && ($rnd <=
          (1 - $sProb[2][4]))) {
            // Pick state 3.
            $crntState[$i] = 3; $v1 = $states[3];
        }
        else if (($rnd > (1 - $sProb[2][4])) && ($rnd <= 1)) {
            // Pick state 4.
            $crntState[$i] = 4; $v1 = $states[4];
        }
    }
```

Remember to complete the code for current states 3 and 4! All states are represented in finished file on the CD-ROM:

18_Cell_Migration/MEL/cRule1.mel

Persistence time

Next you'll calculate the time, `$persist[$i]`, that the cell will persist in its new state, via Daniel Gillespie's formulation described on page 530. Run many times, the formula will return values, `$tau`, distributed exponentially about the average persistence time, `$pAvg` for each cell state. You will store `$pAvg` values within attributes of the control widget later in the chapter.

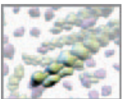
```
    // Calculate the persistence time.
    $rnd = rand(0,1);

    if ($crntState[$i] == 0) { // State 0.

        $pAvg = `getAttr widget.persistState0`;
        $tau = $pAvg*log(1/$rnd);
    }
    else if ($crntState[$i] == 1 || $crntState[$i] == 3) { // State
      1 or 3
        $pAvg = `getAttr widget.persistState13`;
        $tau = ($pAvg)*log(1/$rnd);
    }
    else { // State 2 or 4.
        $pAvg = `getAttr widget.persistState24`;
        $tau = ($pAvg)*log(1/$rnd);
    }
```

The new `$persist[$i]` is determined by adding `$tau` to the current frame number which is stored in the global variable `$frameCheck`. The ceil function return the next highest integer to the value of `$tau`. `$persist[]` is a global variable; the value assigned to it here is available the next time cell this procedure is called from the moveCells expression.

```
    $persist[$i] = $frameCheck + ceil($tau);
```

The return value

The state vector `$v1` must be converted from a unit vector (i.e. of magnitude 1) to a vector in fiber parameter space. The parameter U-component is calculated by dividing



the unit vector U-component ($v1.x) by the fiber circumference ($pi × $fDiameter). The parameter V-component is calculated by dividing the unit vector V-component ($v1.y) by the fiber length ($length).

```
            // Calulate the random walk vector.

            // Convert the return vector to parameter space.
            $u = (1 / ($pi * $fDiameter)) * $v1.x;
            $v = (1 / $length) * $v1.y;

            // Scale $v1 by the magnitude $vScale.
            $v1 = $vScale * << $u, $v, 0 >>;

        } // End if ($persist[$i] <= $frameCheck).
```

If the cell did not transfer fibers, it will persist on its current heading which is stored in $deltaUVs

```
        else $v1 = $deltaUVs[$i]; // The cell continues in its previous
          state.

        // Return the random walk vector to the moveCells expression.
        return $v1;

    } // End cRule1 procedure.
```

Save your file

Save cRule1() in a file called cRule1.mel within your Maya Scripts directory.

cRule2(): Contact avoidance

This procedure embodies the cell-cell signaling component of the model. It returns a vector that is used in the main moveCells expression to nudge the current cell away from neighbors that are too close according to a threshold value you'll set below.

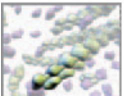
```
    /* Description:
    This procedure moves the current cell $i away from a neighboring
    cell if the two are within a threshold distance of one another.
    The procedure arguments are the same as those used in cRule1().
    */

    global proc vector cRule2 (int $i, float $vScale, string
      $surfaceName) {

        /***** DECLARE THE VARIABLES *****/

        global vector $cellPos[];
        global string $cellNames[];
        global float $cDiameter;
        global int $cellCount;

        /*
        $separation     The vector separating the cell centers.
        $totalSep       The total of the separation vectors for all
                        neighbors too close to cell $i.
        $unitSep        The unit vector of $totalSep.
```

```
*/
vector $neighborPos, $separation, $totalSep, $unitSep, $v2;

/*
$dist          The magnitude of $separation.
$contactRange  The tolerable distance between cells.
$v             The V component of the return vector, $v2.
*/
float $dist, $contactRange, $v, $length;

/*
$name           An index used to increment through the cell
                name array, $cellNames[].

*/
string $name;

/*
$j              The index number of the neighboring cell.

*/
int $j;

/***** INITIALIZE THE VARIABLES *****/

$dist = 0;
$contactRange = `getAttr widget.contactRange` / 100 *
  $cDiameter;
$totalSep = <<0, 0, 0>>;
$length = `getAttr ($surfaceName + ".length")`;
$v2 = << 0, 0, 0 >>;
$j = 0;
```

Main loop

A for loop is used to increment through the cell population.

```
/***** MAIN BODY *****/

for ($j = 0; $j < $cellCount; $j++) {

    // Test all but the current cell.
    if ($j != $i) {

        $separation = $cellPos[$i] - $cellPos[$j];
        $dist = mag($separation);

        // Are the cells too close to one another?
        if ($dist < $contactRange) {

            // cell $j is too close to cell $i.
            $totalSep += $separation;
        }
    }
} // End the cell name loop.
```



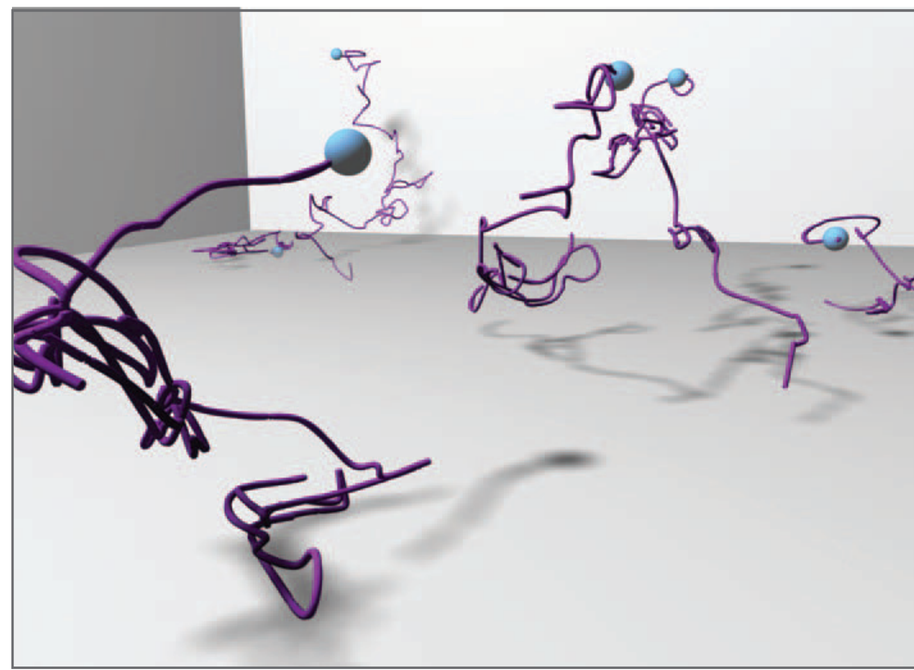


FIGURE 18.19

When the matrix is hidden, cell trajectories show clearly where cells have transferred between fibers. You can apply the technique described in *Chapter 13* to generate trajectory paths using Maya Paint Effects.

The return value

The return vector is the Z-component of the unit total separation vector, `$unitSep`. Multiplying `$unitSep.z` by `$vScale` scales the return value in terms of the universal step size for the simulation. Dividing the scaled `$unitSep.z` by the fiber length (`$length`) converts it to a parameter value in fiber surface UV space.

```
        // Calculate the contact avoidance vector.
        if ($totalSep.z  < 0) $v = -1;
        else if ($totalSep.z > 0) $v = 1;
        $v = $vScale * $v / $length;
        $v2 = << 0, $v, 0 >>;

        // Return the contact avoidance vector to the moveCells
          expression.
        return $v2;

    } // End procedure.
```

Without too much trouble, you can repurpose this procedure to get cells to move toward one another or to follow certain “leaders” based on their positions within the scaffold.

Save your file

Save `cRule2()` in a file called `cRule2.mel` within your Maya Scripts directory.

cRule3(): Transferring between fibers

Unlike the previous two procedures, this one returns a string array. The first element of the array will be either "yes" or "no", reflecting whether or not the cell will transfer fibers . If the first element is "yes" then:

(a) The second element will hold the index number of the fiber to which the cell will transfer.

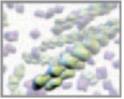
(b) The third element holds the closestPointOnSurface position value for the new fiber.

(c) The fourth element stores the vector representing the U and V attributes of the closestPointOnSurface (or cpos) node.

On the other hand, if the first element is "no" then the remaining $v3[] elements will be assigned null values. This procedure has one argument: the current cell index, $i.

While variables can be defined as vector arrays in Maya (e.g. vector $myVect[];), the same is not true for procedures. A procedure cannot return a vector array. It can, however, return a string array, which is a convenient way to return several values from a single procedure to the script that called it.

```
/* Description:
This procedure determines if the cell will transfer from its current
fiber to another fiber with which the cell is in contact.
*/

global proc string[] cRule3 (int $i) {

    /***** DECLARE THE VARIABLES *****/

    global vector $cellPos[];
    global string $cellNames[], $cposNames[], $closestSurface[];
    global string $surfaceNames[];
    global float $cDiameter, $probUp;
    global int $frameCheck, $transWait[];

    /*
    $cpos          The closestPointOnSurface position attribute
                   value for the cell's current fiber.
    $newCpos       The closestPointOnSurface position attribute
                   value for the cell's new fiber.
    */
    vector $cpos, $newCpos, $pos;

    /*
    $dist          The distance between the cell center and the
                   closestPointOnSurface position for the fiber
                   currently under scrutiny.
    $threshold     The critical distance between the cell and a new
                   fiber, below which the cell will consider a transfer.
    $prob          The probability of a transfer. Its value is
                   either the same as $probUp, or the inverse
                   probability: 1 - $probUp.
    */
    float $dist, $threshold, $prob, $u, $v, $rnd;

    /*
    $current       The cell's closest surface number (the one to
                   which it's attached).
```




```
$closest        The cell's new closest surface number (the one to
                which it may transfer).
$wait           The time in frames the cell must wait before
                checking for a transfer opportunity.
$willTrans      The result of the cell's decision to transfer or not.
*/
int $current, $closest, $wait, $willTrans, $closest, $j;

/*
$v3[]           The return string for this procedure.
$currentCpos    The cpos node corresponding the cell's current fiber.
*/
string $v3[], $currentCpos, $name;
```

For the initial `$threshold` value, use the cell's contact radius which is given by `$cDiameter/2`. You can vary this value or the cell diameter later on to adjust the sensitivity of the cell's reach.

```
/***** INITIALIZE THE VARIABLES *****/

$pos = $cellPos[$i];
$dist = 0;
$threshold = $cDiameter/2; // The cell's radius.
$current = `getAttr ($cellNames[$i] + ".closestSurface")`;
$currentCpos = $cposNames[$current];
$wait = 5;
$j = 0;
```

Main loop

A for loop is used to increment through the scaffold fibers. Each fiber is evaluated via its cpos node for proximity (`$dist`) to the cell. If `$dist` is less than `$threshold` then the procedure moves on to test the probability of transferring to that fiber. If not, then that fiber is discarded and the proximity to the next one is checked, and so on.

```
/***** MAIN BODY *****/

for ($name in $cposNames) {

    // Test all but the cell's current fiber.
    if ($name != $currentCpos) {

        // Get the closest point on the surface to the cell's
          position.
        setAttr ($name + ".inPosition") -type double3 ($pos.x)
          ($pos.y) ($pos.z);

        $cpos = `getAttr ($name + ".position")`;
        $dist = mag($cpos - $pos);

        // Determine if $dist is less than the $threshold value.
        if ($dist < $threshold) {
```

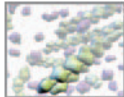
The transfer probability

In this next section you'll determine the probability of transferring to the contacted fiber. The variable `$probUp` specifies the likelihood that the cell will transfer to a fiber for which the closest point in its surface is higher (in Y) than the cell's center. In other words, the `cpos.position` Y-component is greater than the cell's `translateY` attribute. `$probUp` was set to 0.9 (or 90%) in the `resetCells` expression, a value that will help bias migration in the positive Y-direction. If the `$cellPos` Y-component is less than the cell's `translateY` value, then the probability for transferring is given by: `1 - $probUp` = 0.1 (or 10%). Since it can be difficult to see fiber transfers when they occur during a simulation run, it's sometimes helpful to print messages in the Script Editor when these events occur. Below, you'll start by assembling a message which you'll print a little further on.

```
            // Assemble a message to print in the Script editor.

            string $tmpStr = "\n" + $cellNames[$i] + " is in contact
              with " + $name;

            // Test if new contact is above or below the cell's center.
            if (($cpos.y) > ($pos.y)) {
                $prob = $probUp;
                // Add to the Script editor message.
                $tmpStr  += " ABOVE it.\n";
            }
            else {
                $prob = (1 - $probUp);
                $tmpStr += " BELOW it.\n";
            }
            // Print the message.
            print $tmpStr;
```

To determine the outcome, again draw a random number on the unit interval and compare it to the probability.

```
            // Test the probability of a transfer.
            $rnd = rand(0, 1);
            if ($rnd < $prob) {

                // The cell will transfer to a new fiber.
                $willTrans = 1;
                $newCpos = $cpos;
                $closest = $j;
            }
            else $willTrans = 0; // The cell will not transfer.

            // Calculate the new waiting time.
            $transWait[$i] = $frameCheck + $wait ;
        }
    }

    // Increment the surface index number.
    $j ++;

} // End the cposName loop.
```



Calculate the waiting time

Whether or not the cell transfers between fibers, it must now wait before checking for another opportunity to transfer. This prevents the cell from bouncing back and forth between two or more fibers with which it is in contact. The wait time is given by $wait, which you set to 5 frames earlier in this procedure. You can vary this number to govern the frequency with which a cell can actively detect contact with fibers. In this way $wait offers yet another point of control for your future enhancements of this initial migration model. If you wish, you can add a custom attribute to the control widget to store and easily change (via the Channel Box) the value for $wait.

```
// Determine the new waiting time.
$transWait[$i] = $frameCheck + $wait;
```

The return value

Now you'll build the string array $v3[] and return it to the expression that called it. The vectors representing the cpos position and UV values are constructed as strings so they can be packaged in $v3[].

```
    // Assign the return values for this procedure.
    if ($willTrans == 1) { // The cell will transfer fibers.
        // Query the u and v surface coordinates from the cpos node.
        $name = $cposNames[$closest];
        $u =`getAttr ($name + ".parameterU")`;
        $v =`getAttr ($name + ".parameterV")`;
        // Compose the return array.
        // The cell will transfer, therefore
        $v3[0] = "yes";
        // The surface it will transfer to is
        $v3[1] = $closest;
        // The cell attachment position in world space:
        $v3[2] = ("<<" + ($newCpos.x) + "," + ($newCpos.y) + ","
                    +($newCpos.z) + ">>");
        // The cell attachment position in parameter space:
        $v3[3] = ("<<" + ($u) + ","+ ($v) + ","+ (0) + ">>");
        // Announce the transfer in the Script editor.
        $tmpStr = $cellNames[$i] + " transferred from " +
        $surfaceNames[$current]  + " to " + $surfaceNames[$closest]
          + "\n";
        print $tmpStr;
    }
    else $v3 = { "no", "null", "null", "null" }; // The cell won't
      transfer.
    // Return the fiber transfer array to the moveSeeds expression.
    return $v3;
} // End procedure.
```

Save your file

Save cRule3( ) in a file called cRule3.mel within your Maya Scripts directory.

That concludes the scripting portion of this project. Next you'll open the scaffold model and source and debug the script files.

Methods: Running the simulation

Prepare your scene file

Start Maya. If it's already running, save your work. Set your project directory to the one you created at the end of *Chapter 17*.

1. **From the main menu bar, choose File → Project → New.**
2. **Navigate to cellMigrationProject and press Choose.**

Open the scene file you created in the previous project.

3. **(a) Choose File → Open Scene**

   (b) Navigate to your scene file, scaffold.ma, and press Open.

The cell migration simulation will execute properly on a scaffold model that is built to the specifications we outlined in the previous chapter. Most importantly, the `closestPointOnSurface` node, the `pointOnSurfaceInfo` node, and the control widget must possess the attributes that are queried in the current project. We have provided a finished scaffold scene file on the CD-ROM:

18_Cell_Migration/scenes/scaffold.ma

Next, set the scene preferences:

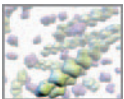
4. **Choose Window → Settings/Preferences → Preferences.**
5. **Choose Categories → Settings and make the following settings:**

   Under Working Units → Linear: centimeter.
   → Angular: degrees.
   → Time: NTSC.

6. **Choose Categories → Timeline and make the following settings:**

   Under Timeline → Playback Start: 1.
   → Playback End: 500.
   → Time, select NTSC.
   Under Playback → Looping: once.
   → Playback Speed: Play every frame.
   → Playback by 1.

7. **Press Save.**
8. **Select the Perspective view of your scene by pressing the [icon] button in the Toolbox.**



Repurpose the control widget

While building the above scripts you've used the `getAttr` command several times to query control widget custom attributes. You've then used the results to set various parameters for the model. Here you'll add those custom attributes after deleting the ones not needed for this project:

1. **Select the widget in the Outliner and then press Ctrl+A to open the Attribute Editor.**
2. **In the Attribute Editor, click on the left-most tab to select the widget transform node.**
3. **From the Attribute Editor menu bar, choose Attributes → Delete Attributes.**
4. **In the Delete Attribute window, hold down the Shift key while selecting the following attributes:**

   avoidScale, boundScale, dx, dy, dz, minSpan, maxSpan, persistMin, persistMax, sizeBias, and spacingMin.

5. **Press the Delete button then the Close button.**

Now you'll create the new attributes and set their default values. You can do this manually, through the Attribute Editor, or by entering the following script in the Script Editor. The attributes called `persistState13` and `persistState24` are the average persistence values for states 1 and 3, and states 2 and 4, respectively. States 1 and 3 pertain to movement on the fibers that is circumferential, and states 2 and 4 to longitudinal movement.

```
/* Description:
This adds custom attributes to the scaffold widget object. The
attributes are used in the cell migration simulation.
*/

// Make an array of custom attribute names you want to add to the
  widget.
$attributes = {"cDiameter", "offset", "vScale", "probUp",
               "contactRange", "persistState0", "persistState13",
               "persistState24"};

/* Make an array of initial attribute values. The second "float"
ensures all values are typed as floating point numbers, not
integers. */
float $values[] = float {10, 50, 10, 0.9, 125, 1, 2, 2};

// Add and set the custom attributes.
for ($i = 0; $i < `size $attributes`; $i++) {

    addAttr -ln $attributes[$i] -at double -dv $values[$i] widget;
    setAttr -e -keyable true ("widget." + $attributes[$i]);

}
```

The initial attribute values listed above are what we recommend starting with. Once your model is running error-free, you can vary these values within the ranges listed in Table 18.04 (and beyond!). The exception is of course cDiameter, which should only be changed when you change the diameter of the spheres that represent cell contact radius.

Parameter name	Description	Location in your scene	Recommended value range
$cDiameter	The cell diameter value used in contact avoidance and fiber transferring.	widget.cDiameter	**10**
$offset	The distance of the cell center from the surface of its fiber, as a percentage of cell radius.	widget.offset	**0–100**
$vScale	Scales the unit locomotion step size.	widget.vScale	**1–10**
$probUp	Specifies the likelihood of a cell transferring to a higher fiber with which it's in contact.	widget.probUp	**0–1**
$contactRange	The range within which cells take steps to avoid contact with one another, as a percentage of cell diameter.	widget.contactRange	**100–300**
$persistState0	The average persistence time in frames of a cell in state 0	widget.persistState0	**1–10**
$persistState13	The average persistence time in frames of a cell in states 1 or 3.	widget.persistState13	**1–10**
$persistState24	The average persistence time in frames of a cell in states 2 or 4.	widget.persistState24	**1–10**
$sProb	The state change probability matrix.	resetCells	**Sum of the values in each matrix column must equal 1.**

TABLE 18.04

Model parameters to experiment with and their recommended value ranges.

This script can be found on the CD-ROM:

18_Cell_Migration/MEL/widgetAttributes.mel

Save your scene

Save your scene file under a new name in your cellMigrationProject directory:

1. **From the main menu bar, choose File → Save Scene As.**
2. **Enter cellMigration.ma in the Save As field and then press the Save button.**

Now you're ready to load your script files in preparation for a simulation run.

Load the script files

Create the expressions

First you'll create the resetCells and moveCells expressions. If you didn't build the expression scripts earlier in the chapter, you can find them in the following files on the CD-ROM:

18_Cell_Migration/MEL/resetCells.txt
/moveCells.txt

1. **Open resetCells.mel (either the file you created or the one on the CD-ROM) in the text editor of your choice.**
2. **Ensure that the text editor is set to *not* use typographers' quotation marks.**
3. **Select and copy the entire script.**
4. **In Maya, enter `ExpressionEditor` in the Command Line to launch the Expression Editor, or select it from the menu Windows → Animation Editors → Expression Editor.**
5. **Press the New Expression button.**
6. **LMB+click in the Expression text field.**
7. **Press Ctrl+V to paste your expression into the text field.**
8. **Press the Create button at the bottom of the Expression Editor.**
9. **In the Expression Name field, replace the default name with resetCells and press Enter.**
10. **Repeat steps 5 through 9 for the moveCells expression, but name it moveCells in the Expression Name field.**

If Maya generates one or more errors when you press the Create button, you will need to debug the expression: open the Script Editor to view the specific error messages and to read the line number(s) that generated the error(s). If your text editor can display line numbers, use this feature to cross-reference the error messages to the offending lines in your script. If you are unable to resolve the errors, you can compare your script to the appropriate file (**resetCells.txt** or **moveCells.txt**) included on the CD-ROM.

11. **Press Ctrl+S to save your scene with the expressions in it.**

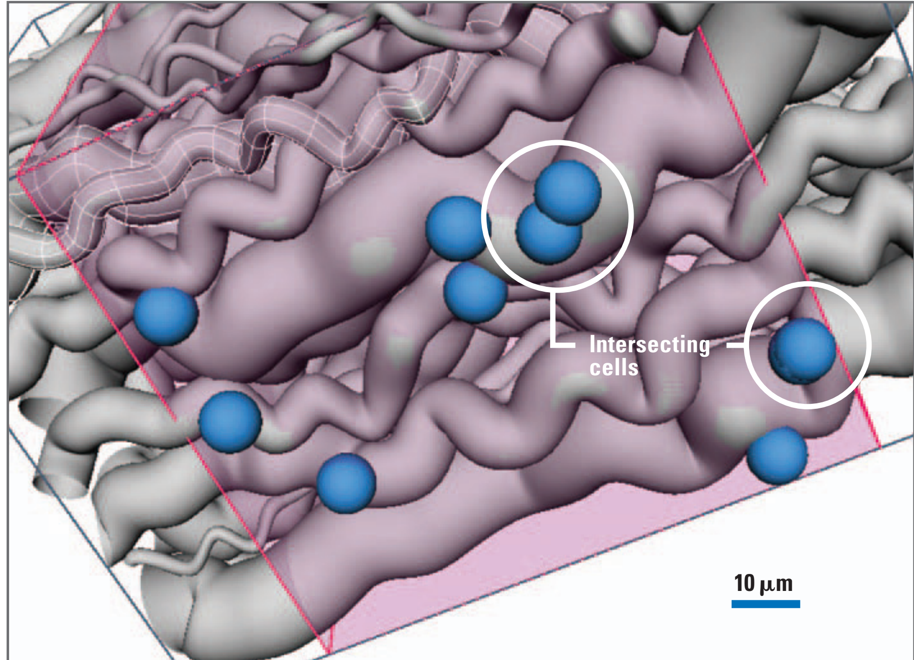


FIGURE 18.20

When you first make the cells, some of them may intersect each other due to their initial random placement on the bottom of the scaffold. The grey lines define the scaffold boundaries. The red lines define the active simulation volume. A Blue shader was applied to the cells to make them easy to see.

Source makeCells() and the rule procedures

Source the procedures one at a time and debug them as necessary.

1. **Refresh the search path contents. In the Script Editor, enter:**

```
rehash;
```

2. **Source the script files. In the Script Editor, enter:**

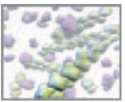
```
source "makeCells.mel";
source "cRule1.mel";
source "cRule2.mel";
source "cRule3.mel";
```

If Maya generates errors, debug the offending procedure(s) accordingly. You can cross-reference your scripts with those we've included on the CD-ROM:

18_Cell_Migration/MEL/makeCells.mel
/cRule1.mel
/cRule2.mel
/cRule3.mel

Create the cells

One you've successfully entered the two expressions in the Expression Editor and declared the four procedures without generating error messages, you're ready to create



the cell population and begin the migration! Start with 10 cells of radius 5 μm and specify a scaffold buffer length of 25 μm.

1. **Set the current time indicator (play head) to frame 1.**
2. **Enter the procedure call in the Expression Editor or the Command Line:**

```
makeCells(10, 5, 25);
```

Some cells may overlap one another due to the random nature of their initial placement (Figure 18.20). The contact avoidance procedure will sort out the overlapping.

Run the simulation

Before pressing the Play button to start the simulation, run the `resetCells` expression to initialize the global variables.

1. **Open the Expression Editor and choose Select Filter → By Expression Name.**
2. **LMB+click on resetCells in the Expression field to make it the active expression.**
3. **Press the Create button at the bottom of the Expression Editor.**
4. **Close the Expression Editor.**
5. **Press the Play button to start the simulation.**

On frame 2, the `moveCells` expression calls all three-rule procedures. If any of them is missing or contains errors, Maya will generate an error message in the feedback field of the Command Line as well as the Script Editor. If you get an error message, read it carefully to determine the source: it could be in one of the rule procedures or in moveCells itself. If you are unable to locate and correct the source of an error message, compare your script files to the ones we included on the CD-ROM. We have also included a Maya scene file called `cellMigration.ma`, which contains the scaffold, control widget, cell models, and the expressions. To run the simulation using `cellMigration.ma`, you will need to copy the accompanying rule files into your Scripts directory and run the rehash command to add them to Maya's search path (Figure 18.21).

18_Cell_Migration/scenes/cellMigration.ma

/MEL/cRule1.mel

/cRule2.mel

/cRule3.mel

If Maya generates no error messages when you press Play, then your cells should begin crawling about on the fibers as the playhead progresses along the timeline. You can stop and rewind the simulation at any time using the Playback control buttons.

Vary the model parameters

This is the fun part! Trying varying the parameters listed in Table 18.04—at first one at a time and then in combination—in order to observe their relative effects on the behavior of the model. This brings us full-circle to our discussion of individual-based modeling and emergent behavior at the beginning of the chapter. The motion of each cell is a stochastic process that emerges from the behavior rules and their parameters.

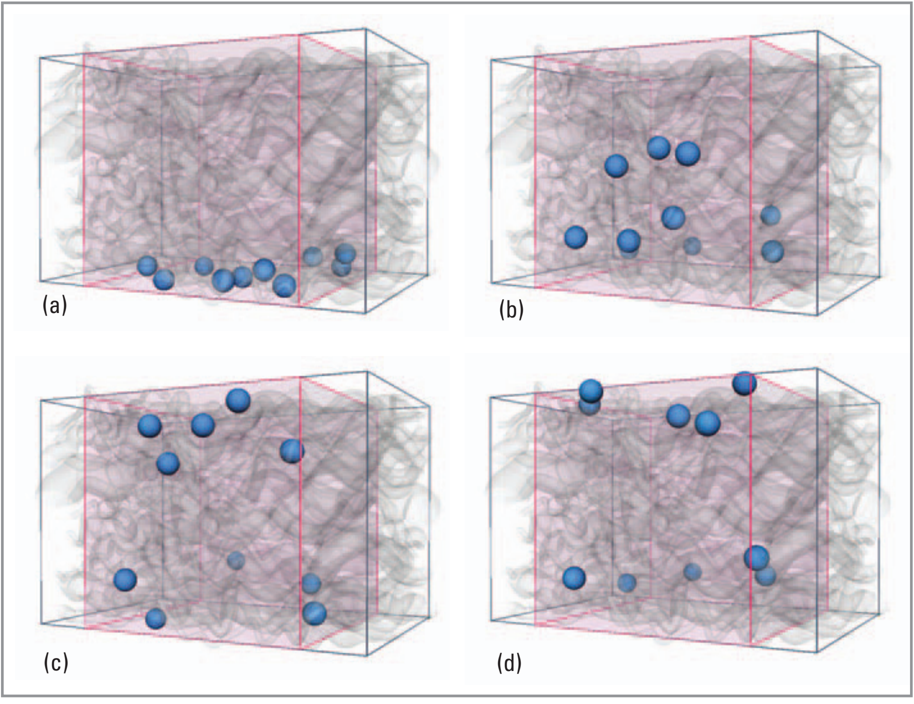


FIGURE 18.21

Four stages of the cell migration simulation at:

(a) frame 15.

(b) frame 175.

(c) frame 330.

(d) frame 500.

The fibers are transparent so that cells can be seen through the scaffold. Note that half cells found their way into the upper part of the scaffold while the rest remained essentially stuck at the bottom. This discrepancy is due the random nature of the migration rules and the opportunities for "climbing" that the cells encounter during their journey.

The distribution pattern of cells in the scaffold during a simulation run is not the result of equations governing the population as a whole—as in a **deterministic** model—but rather the novel consequence of choices made one cell at a time according to individual rules of behavior, including cell-cell and cell-matrix interactions. In this way, your Maya cell population is not unlike a flock of birds or a school of fish in that its spatial arrangement at any moment in time is the result of decisions made by the group's members.

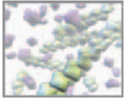
After you've put the model through its paces, choose a set of parameters for which you want to make a recording of the simulation.

Record the simulation

When you set keyframes Maya records the simulation run in the form of animation curve nodes in the DG. A recorded simulation can be played back repeatedly using the Playback controls or by scrubbing the timeline. To keyframe your simulation, remove the comments from the code in the `moveCells` expression that sets the translate attribute keyframes, as follows:

Change:

```
// setKeyframe -at translateX $cellNames[$i];
// setKeyframe -at translateY $cellNames[$i];
// setKeyframe -at translateZ $cellNames[$i];
```



To:

```
setKeyframe -at translateX $cellNames[$i];
setKeyframe -at translateY $cellNames[$i];
setKeyframe -at translateZ $cellNames[$i];
```

When the playhead reaches the end of the playback range, you will need to either delete or disable `moveCells` so that it doesn't override the cell translation keyframes when you play the recorded simulation back. To disable `moveCells`, set the frame number in its opening conditional statement to a number high enough that your scene is unlikely to ever reach it:

In the moveCells expression, change:

```
if (`currentTime -query` > 1) {
```

To:

```
if (`currentTime -query` > 10000) {
```

The `resetCells` expression will take care of positioning the cells correctly on frame 1 by using their respective `posInit` attributes. Alternately, you can uncomment the `setKeyframe` statements in `resetCells` to record keys at frame 1. We included a key-framed simulation run on the CD-ROM:

18_Cell_Migration/scenes/cellMigration_keyed.ma

Results: Data output

You'll want to export the cells' trajectories as a matrix of XYZ coordinates in a **tab-delimited** file. This file can then be used via your favorite statistical analysis software to further study the properties of your simulated cell motions. For example, do the paths behave as simple random walks or are they more complex tracks of the kind reported in laboratory studies of real cells. Are the paths simple geometric form, or do they show complicated recursive features like those of fractal meanders? Questions like these bring you to the frontier of cell migration modeling and research!

The dataOutput expression

Here you'll use an expression called `dataOutput` to write the cell positions to an external text file. The expression is included ready-made on the CD-ROM:

18_Cell_Migration/MEL/dataOutput.txt

1. **Copy dataOutput.txt to the MEL directory of your current Maya Project.**
2. **In Maya, set the current time indicator to frame 1.**
3. **Open dataOutput.txt in your text editor application.**
4. **In the variable initialization section, change the value of $end- Frame to match the final frame number for your simulation.**
5. **Copy and paste the script into a new expression in Maya's Expression Editor the same way you did with resetCells and makeCells expressions.**
6. **Name this new expression dataOutput.**

You can record the data while `moveCells` is active—rather than from a keyframed recording of the simulation. However, for this to work you'll need to ensure that `dataOutput` follows `moveCells` in the list of expressions in the Expression Editor. Otherwise, dataOutput will execute first and you'll wind up recording cell positions before they're updated by `moveCells` each frame.

7. **Press the Play button to playback the simulation and record the data.**
8. **When the last frame of your simulation is reached, the dataOutput expression will open a file dialog window. Name your data file and browse the location on your hard drive where you'd like to save the file.**
9. **Open the file in a spreadsheet application such as Microsoft Excel that accepts tab-delimited data.**

This file is small relative to your Maya scene file and therefore an efficient way to store a record of your simulation. A slight rejig of your `moveCells` expression from *Chapter 13* will allow you to read the data file you just wrote in to Maya and use it to reproduce the migration simulation results. If you do this, remember to disable the `resetCells` and `moveCells` expressions so they don't clash with `dataInput` for control over the cell geometry.

Summary

In this chapter you saw how to create a multi-agent simulation from a few straightforward rules of motion. Handy MEL constructs including the `closestPointOnSurface` and `pointOnSurfaceInfo` nodes allowed you to quickly adapt available data on 2D cell motion to the complex geometry of a 3D scaffold. Much more can be said about this initial model and its potential role in understanding mobile population behavior than we have space for here. Whether you're interested in multi-agent simulation for research purposes or for developing procedural effects in biomedical communications projects, we hope you'll continue to build on the methods and tools we've introduced here.

References

1. Armour AD, Fish JS, Woodhouse KA, Semple JL: A comparison of human and porcine acellularized dermis: Interactions with human fibroblasts in vitro. *Plastic and Reconstructive Surgery* 117(3): 845–856, 2006.
2. Noble PB, Levine MD: *Computer-Assisted Analyses of Cell Locomotion and Chemotaxis*. CRC Press, Boca Rataon, FL, 1986.
3. Bergner P-EE: Dynamics of Markovian Particles: A Kinetics of Macroscopic Particles in Open Heterogeneous Systems (website): www.bergner.se/DMP/download.htm, accessed August 4, 2007.
4. Gillespie DT: Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry* 81: 2340–2361, 1977.

This page intentionally left blank

19 Conclusion

A new kind of seeing

"Computers are useless. They can only give you answers."

—Attributed to Pablo Picasso[18]

Anton van Leeuwenhoek was a Dutch tradesman, living in 16th century Amsterdam, with no formal scientific education. Despite his lack of training and the demands of his business, he ardently pursued his passion: a new kind of seeing. He was fascinated with optics and worked to refine the microscope, then a recently developed device. His technical improvements, and his dogged curiosity, opened up a broad vista of sight-based evidence to science. He was the first to see and describe mobile single-celled organisms in pond water, human blood cells, and the wriggling of sperm cells. van Leeuwenhoek's new kind of seeing lead to countless scientific discoveries over the centuries that followed, discoveries that resulted from new questions: questions that could not even be conceived of before the microscope revealed fresh mysteries to inquisitive eyes. Looking beyond van Leeuwenhoek toward the present, we are hard pressed to name any revolutions in the sciences, the arts, and in practical engineering that have not been triggered by new ways of seeing the world.

Now, in the early years of the 21st century, radical new kinds of seeing are once again front page news as artists and scientists harness the computer as an engine for visualizing the unknown. These new kinds of seeing empower us to understand the sciences of life and health as never before. Making things visible, we have seen in this book, is vastly more than the subtle art of making dull facts palatable for the time-stressed, the uninformed, or the bored who find themselves obliged to regurgitate time-honored answers to time-honored questions. Interpretive visualization begins in the surprise of unexpected—yet immediately graspable—results that allow fresh new questions to be asked. And more than finding answers, asking new questions is the heart of the sciences and the arts in the human journey.

So in this sense Pablo Picasso, protean genius of the radically visual, was flat wrong: as a gateway to interpretive visualization, computers fuel new questions sustained by unanticipated dreams and possibilities. They will give you much more than answers.

Explanations, simulations, speculations

We have seen that Maya and MEL, in combination with knowledge from the domains of molecular and cell biology in science, can be exciting tools for the simulation and visualization of cell systems, from molecules to polymers, cells, and populations. They are part of a postmodern visual language of asking and answering questions about the previously unseeable. They empower you to create moving images that can be explanations, simulations, or (in a more exploratory vein) speculations about answers to compelling new questions. These results can be put to use in research, in education, and in much-needed debates about where science is taking us:

- *Explanations*: fulfilling a growing need for advanced visual tools in the research communication, professional education, and public information realms. Education in science and science literacy are areas of growing need for visual explanations (Figure 19.01).
- *Simulations*: drawing on established databases of biomolecule structure, reaction dynamics, and other experimentally derived data about cell structure and function, tools like Maya and MEL can be used to recreate and interrogate lifelike situations, which would otherwise be remote or unavailable to researchers.
- *Speculations*: in the best tradition of scientific inquiry, dynamic models can be used to ask the "What if?" questions that drive innovation in areas like drug

September 21, 1932

Linus Pauling

Freshman Chemistry.

I believe that a book would be valuable to young students which gave them concrete pictures of molecules as we now picture them. Ionic substances could well be described as containing spherical ions of radii given by "crystal radii", the electrons staying mostly within. NaCl would be

etc.

Covalent molecules would be :Ö::C::Ö: and :O:::C:Ö: etc

SO_2 :Ö:S::Ö: ⇄

H_2SO_4

NH_3

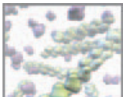
H_2O

H^+ $Sb(OH)_6^-$ H^+

FIGURE 19.01

Nobel laureate chemist Linus Pauling championed interpretive visualization, set out in the form of striking illustrations and models, for the pursuit of scientific knowledge. On this page dated September 21, 1932, Pauling explains how visualizing 3D form of molecules could help students of chemistry. Eight decades later Pauling's argument is still fresh, capturing the essence of why interpretive visualization is so important in all areas of science.

Courtesy of the Archives, California Institute of Technology.



discovery and biomaterials development. What if we could make an artificial tissue matrix of unprecedented strength? What if we could alter crucial signaling mechanisms in dying or cancerous cells? What if we could bioengineer our human bodies to thrive for 500 years, rather than our traditional four score and seven? Speculative responses to questions like these don't necessarily provide definitive answers right away; but they give clues about which of the many avenues of possible exploration may prove most fruitful.

Maya's role

Maya was designed as a general-purpose 3D modeling, animation, and rendering application. Maya is not a tool where specialized capabilities in cellular science and medicine are built-in. At present these must be created by you as computable models. Your models express the relations of cause and effect, structure and function, you deem essential to your project and then are coded by you in one of Maya's programming languages. The benefits of a Maya-based workflow compared to writing all your own software from scratch is that many other capabilities—all the graphical user interface efficiencies, geometry modeling functions, dynamics, interactive display, and sophisticated rendering capabilities—are already present, robustly built, and tested in the world's most demanding professional 3D visual production situations.

Most of the advantages we have cited for Maya and MEL also apply to other scriptable high-end packages for 3D animation and rendering. To a greater or lesser degree, packages such as Autodesk 3ds Max[1], Maxon's Cinema 4D[2], Softimage XSI[3], Side Effects Software's Houdini[4], and Newtek's Lightwave[5] (forgive us if we have missed other deserving products) possess excellent modeling, animation, scripting, rendering, and dynamics capabilities. The general approaches we have outlined should, with the appropriate effort on your part, translate to these packages, though the details of implementation and performance may differ considerably. And while our emphasis has been on commercial top-tier packages, we must not leave this point without stressing the exciting potential—still largely unexplored in molecular and cellular iVis as we go to press—of lower cost, shareware, or freeware packages such as Blender[6], the Visualization Toolkit[7], and the Torque Game Engine[8]. Fueled by the creative energies of the open source and personal computation movements worldwide, these tools of visual discovery can deliver impressive quality within their design envelope. We encourage you to investigate them, together with the packages like VMD[9] written specifically for jobs in molecular and cellular modeling and visualization, and readily available over the Internet. When the needs of your project fit inside the style and capabilities of these existing bioscience tools, you may find them the fast track to productivity and successful results. If your needs push outside the envelope, packages such as Maya and MEL will let you cut your own path. In all of these situations, we hope that you will find yourself well served by the methodologies of problem definition, analysis, and 3D iVis workflow planning and execution you have learned in this book through Maya and MEL.

The path so far

Part 1 of this book was devoted to setting the appropriate biological and technological background for the rest of the book. The Introduction revealed the book's major

themes, briefly looking at the power of vision, the hierarchical structure of life, and the history of our tool of choice, Maya. With the help of *Chapter 02*'s historical perspective, we probed how computers work and how computer programs function; even if we opened the book as newcomers to computing or programming or 3D graphics, this allowed us to understand what it means to say MEL—our book's computational focus—is in essence an imperatively structured scripting language for 3D animation on von Neumann machines. We also saw intriguing commonalities between the strategies of information processing in computers and those used in living cells. Far from being a poor fit between the organic freedom of life and the rigid binary world of the computer, there are good reasons to explore biology specifically in terms of its computational nature. These features of biology make it especially amenable to meaningful computer-based simulation. In *Chapter 03* we looked at the traditional history of animation and discussed the lexicon and workflow animators use. We then explored how these aspects of traditional animation are adapted for use in digital animation in general, and biological animation in particular.

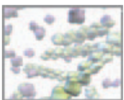
After *Part 1* had set the stage, the chapters of *Part 2* brought you hands-on with the technology platform—Maya—that delivers the visual computing power of MEL to your desktop. The interface of Maya was introduced, and a series of chapters explored the specifics of modeling, lighting, camera setup, shading, animating, and rendering in virtual worlds. In other chapters we began to lay the technical groundwork for our later explorations by introducing the ability to use dynamic simulations, the ability to write MEL scripts, and basic approaches for importing or exporting user-defined data.

In *Part 3* you embarked on a series of directed projects, which built on your growing base of Maya knowledge and let you plan and execute MEL applications at the frontiers of biology. Building a protein gave an opportunity to build one of the big organic polymers that sustain cell structure and catalyze its chemical reactions. *Chapter 15* described how to simulate actin self-assembly, from the single protein to filamentary protein arrays comprising the cell's dynamic skeleton. *Chapter 16* revealed an approach to the basic cycle of amoeboid-type cell movement, fundamental to normal tissue development, immune responses, and cancer. Growing an extracellular matrix (ECM) scaffold explored creating an example of the fibrous ECM, the support framework for the cells of almost every tissue in the body. Finally, simulating 3D cell migration showed how to simulate populations of migrating cells as they invade and occupy an ECM fiber matrix.

These projects build around initial models that showcase Maya tools and techniques, useful approaches that you can expand, refine, and improve upon in your own explorations. The models and MEL code are first steps to help power your continued learning.

The future

There are many opportunities for extending, expanding, and improving on the work you have undertaken in these projects. There are numerous problems in molecular, cell, and tissue biology that will benefit from computational visualization. We have only touched on the possibility of coding more performance-dependent simulations in C++ rather than MEL, or in the Python scripting interface for Maya released as our book was nearing completion. The C++ API also provides a potential link to specialized



packages for cell and molecular simulation (such as VMD[9] and E-Cell[10]) and to bioinformatic databases; in some situations, researchers may choose to run simulations in such custom software and then import those results to Maya via a C++ plug-in, so leveraging the advanced visualization tools offered by Maya. And the growing popularity of Python for general-purpose scripting eases the entry into Maya programming for many new users.

Much work remains in defining new methods and setting best practices in cell and molecular visualization. Current methods of graphical display and numerical processing will soon prove inadequate to the task of manipulating and exploring the extraordinarily intricate and dynamic simulations enabled by increasingly complex software and powerful new hardware. Is computer technology up to the challenge? We think so. Indeed the future looks very bright for computational biology[11–16], as it does for in silico visual simulation as a key tool for biomedical research and discovery. Both are riding a faster-than-exponential growth curve in the pace of computer technology, which we've illustrated schematically in Figure 19.02 alongside a few of the remarkable advances in biology, computers, and visual computing over the last 150 years. During that time the computing power of calculating machines has increased astonishingly in both its capacity and its efficiency. Many measures of each are in use and each has its own loudly outspoken bands of advocates, opponents, pundits, and critics.

In the figure we have used one of the simplest and most traditional measures, which brings together a wide range of historical data by estimating KIPS, the thousands (kilo-, K) of instructions per second (IPS) the device can execute per thousand dollars (kiloBuck) of machine cost, corrected to 2006 currency values[17]. This interpretive visualization of computing history gives one specific glimpse—crude and provisional, but the same general point is made by all the competing measures—of the upward surge in progress over a time in history that embraces the early mechanical and electromechanical devices (black dots), the first von Neumann stored program machines like the EDSAC (green dots), the vintage mainframes and minicomputers of the 1950s and 1960s (orange dots), and on into the streamlined microcomputer architectures and cluster-based supercomputers of today. The modest capacity of the human brain for deliberated, rote pencil-and-paper calculating is marked by the pink dot (and reference line) at the bottom left just before the year 1900, a time when "computer" meant a person employed to do such arithmetic.

Note that the vertical scale in our diagram is logarithmic, so as we trace computing over the last century we see a surge of over a billion, billion times increase in the capacity delivered per dollar, with roughly another factor of a billion forecast—if current trends are sustained!—by the time scientists anticipate the first simulations, in silico, of complete living cells that track the position and actions of each and every molecule in the cell.

Compared to biology and computer engineering, the practices of 3D computer animation and computational biology are relative newcomers with accelerating innovation trends of their own. Once a domain restricted to the privileged elite, who access the most powerful and expensive computers and graphics software, interpretive visualization in silico is now open—via affordable technology—to the expanding community of media artists, research scientists, and mathematicians dedicated to the frontiers of molecular and cellular biology. Imagine the new possibilities, and the new questions, you will uncover in the years ahead.

KIPS per 2006 kiloBuck

10^{14}, 10^{13}, 10^{12}, 10^{11}, 10^{10}, 10^{9}, 10^{8}, 10^{7}, 10^{6}, 10^{5}, 10^{4}, 10^{3}, 10^{2}, 10^{1}, 1, 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}, 10^{-7}, 10^{-8}, 10^{-9}

- (Electro)Mechanical
- Human
- Vintage
- Mainframes
- Small Business
- Minicomputers
- Microcomputers
- Workstations
- Supercomputers
- Sony Playstation 2
- Intel TeraFLOPS PC

1860 1880 1900 1920 1940 1960 1980 2000 2020 2040 2060

Year

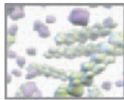
2001 Human genome draft completed
1996 Dolly the sheep cloned
1990 Human Genome Project launches
1972 Recombinant DNA
1961–68 Triplet genetic code cracked: Nirenberg, Matthaei, Ochoa, Khorana
1959 Jacob-Monod "operon" theory of control
1953 DNA double helix: Watson and Crick
1944 Genes = DNA: Avery, MacLeod, McCarty
1926 Gene theory of inheritance: Thomas Hunt Morgan
1905 Gene regulation: Bateson and Punnett
1902 Gene concept: Walter Sutton
1869 DNA discovered
1865 Gregor Mendel and genetics
1859 Charles Darwin and evolution

2007 Intel teraFLOPS chip
1989 mental ray renderer launches
1988 Pixar RenderMan patent
1983 Alias founded; Bill Reeves and particle systems (Lucasfilm)
1981 IBM PC
1975 Bill Gates launches Microsoft
1975 Phong shading algorithm
1972 C programming language: Dennis Ritchie
1971 Gouraud shading algorithm
1960 DEC PDP-1 minicomputer
1954 Fortran: John Backus
1949 EDSAC computer
1945 von Neumann machine architecture
1936 Turing machine: launch of computer science
1919 IBM tabulator
1911 Monroe Calculator
1885 Cathode Ray Tube (CRT)

2030–60 Forecast date for first all-molecule simulation of a complete living cell
2006 First all-atom simulation of a simple life form; satellite tobacco mosaic virus: Freddolino et al
1993 RasMol molecular graphics: Roger Sayle
1976 First molecular dynamics simulation of proteins: Karplus and McCammon
1964 Birth of molecular computer graphics, MIT Kluge: Cyrus Levinthal
1962 CPK molecular models; US patent 3170246 filed
1952 First computer simulations of cell chemistry pattern formation: Alan Turing

FIGURE 19.02

The explosive growth in computing power during modern times, visualized amidst some of the key events in computer engineering, computer graphics and animation, and in research biology. Note that the vertical scale is logarithmic, based on successive multiplications of factors of 10.



References

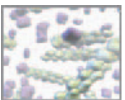
1. Autodesk 3ds Max (website): http://usa.autodesk.com/, accessed October 14, 2007.
2. Maxon's Cinema 4D (website): http://www.maxon.net/, accessed October 14, 2007.
3. Softimage XSI (website): http://www.blender.org/, accessed October 14, 2007.
4. Side Effects Software's Houdini (website): http://www.sidefx.com/, accessed October 14, 2007.
5. Newtek's Lightwave (website): http://www.newtek.com/lightwave/, accessed October 14, 2007.
6. Blender (website): http://www.blender.org/, accessed October 14, 2007.
7. The Visualization Toolkit (website): http://www.vtk.org/, accessed October 14, 2007.
8. Torque Game Engine (website): http://www.garagegames.com/products/torque/tge/, accessed October 14, 2007.
9. Humphrey W, Dalke A, Schulten K: VMD—Visual Molecular Dynamics. *Journal of Molecular Graphics* 14: 33–38, 1996.
10. The E-Cell Project (website): http://www.e-cell.org/, accessed October 14, 2007.
11. Ma'ayan A, Blitzer RD, Iyengar R: Toward predictive models of mammalian cells. *Annual Review of Biophysics and Biomolecular Structure* 34: 319–349, 2005.
12. Bader DA: Computational biology and high-performance computing. *Communications of the ACM* 47: 34–41, 2004.
13. Stewart I: *Life's Other Secret: The New Mathematics of the Living World*. John Wiley, New York, 1998.
14. Watts DJ: *Six Degrees: The Science of a Connected Age*. WW Norton, New York, 2003.
15. Penrose R: *The Road to Reality: A Complete Guide to the Laws of the Universe*. Jonathan Cape, London, 2004.
16. Nowak MA: *Evolutionary Dynamics: Exploring the Equations of Life*. The Belknap Press of Harvard University Press, Cambridge, MA, 2006.
17. Sources consulted for the computer performance/price data used in Figure 19.02 include: Brain M: How microprocessors work (article and tabulation online): http://www.computer.howstuffworks.com/microprocessor1.htm, accessed August 4, 2007; Ceruzzi PE: Moore's law and technological determinism: Reflections on the history of technology, *Technology and Culture* 46: 584–593, 2005; Dongarra J: Overview of high performance computing, *Computer Innovation 6016: The 9th International Conference/Exhibition on High Performance Computing in Asia-Pacific Region*, Beijing China, November 30 to December 3, 2005 (lecture slides online): http://www.netlib.org/utk/people/JackDongarra/SLIDES/hpcasia-1105.pdf, accessed August 4, 2007; Ein-Dor P: Grosch's law revisited: CPU power and the cost of computation, *Communications of the ACM* 28:142–151, 1985; Held J, Bautista J, Koehl S, editors: From a few cores to many: a tera-scale computing research overview, Intel White Paper, 2006 (report online): http://www.download.intel.com/research/platform/terascale/terascale_overview_paper.pdf, accessed August 4, 2007; Kurzweil R: *The Age*

of Spiritual Machines: When Computers Exceed Human Intelligence, Viking, New York, 1999; Lynch BD, Rao HR, Lin WT: Economic analysis of microcomputer hardware, *Communications of the ACM* 33:119–129, 1990; MeasuringWorth: Purchasing power of money in the United States from 1774 to 2006 (calculator online): http://www.measuringworth.com/calculators/ppowerus/, accessed August 4, 2007; Moravec H: *Robot: Mere Machine to Transcendent Mind*, Oxford University Press, New York, 1999; Moravec H: Processor performance 1892–2002 (tabulation online): http://www.frc.ri.cmu.edu/~hpm/book97/ch3/processor.list, accessed August 4, 2007; *Technology News*: IBM and compatible mainframe specifications (tabulation online): http://www.tech-news.com/publib/index.html, accessed August 4, 2007; Smith RE: A historical overview of computer architecture, *IEEE Annals of the History of Computing* 10:277–303, 1989.

18. This piquant assertion about the limits of computers and computation is widely attributed to Pablo Picasso, and as quoted here appeared in Clifford A. Pickover's mind-stretching book *Computers, Pattern, Chaos and Beauty: Graphics from an Unseen World*, St. Martin's Press, New York, 1990, page 10.

This page intentionally left blank

Further reading



Why?

Perhaps you are a creative artist bent on producing the next revolution in interpretive visualization. Or maybe you are a curiosity-driven science trainee exploring the mysteries of cells and other forms of living matter. You might be a teacher immersed in the challenges of communicating science or other novel subjects to a modern audience. Or, like many of us, you might be a private citizen who simply wants to increase your knowledge, appreciation, and capability in realms crucial to scientific discovery and artistic expression. All these wonderful activities set you on a path of lifelong learning that takes you across the frontiers of many subjects. In this part of the book we've listed resources to help you further the experiences you've had with our material—taking you deeper into the foundations, theory, and practical methods of the subjects we have explored.

What's here

Arranged to parallel the order in which the topics unfold in our book, these are titles we use in our own research and teaching, and have found to be informative and inspiring. Of course our list assays just a tiny fraction of what is available. There are many, many fine learning resources from excellent, hard working authors in each of the areas our list touches. If you do not see a favorite title named here, it is no indication we consider it unimportant. These are vast, rapidly growing topics, so as you go even further with Maya and MEL you will discover a wide variety of learning materials written in diverse styles by authors each with their own unique slant on your favorite subjects.

We therefore encourage you to read widely: if a resource we name does not quite fit your learning style or specific interests, chances are you can quickly find material that does aided by our list and the search tools available to you over the World Wide Web and at your local library. We do apologize to colleagues in these wide-ranging subjects for the limitations of time, space—not to mention our own knowledge—that have precluded inclusion of their work here. Wherever possible we have documented material published recently to assure a useful degree of timeliness and availability. But there are also a few classics from yesteryear—some still in print after decades of inspiring learning and discovery—that we could not in good faith omit, and therefore recommend to you most heartily.

Mostly books

We have focused mostly on books in compiling this section, rather than on magazine articles or web-based presentations. Throughout *In Silico* you will find many references to articles and websites we consider vitally interesting to your command of Maya, MEL, and cell science. Despite the convenience of URLs and the tempting brevity of magazine articles, well written, peer-reviewed books will continue to give you a rich medium you can carry, jot in, mark up, flip through, stack, dog ear, and otherwise fit to your learning needs as you explore further, in depth. To supplement these book-based resources, near the end of this section you'll find our current "Top 10" must-see URLs for your web-based explorations.

Also, we give you fair caution: while the listed readings will give you next words on many of *In Silico*'s topics, they will not give you the last word on any of them. You live amidst revolutionary times in the arts, the sciences, and the high technologies,

with discoveries and innovations racing past one another at an astonishing pace. Compared to the very latest research, information in the books we list will be somewhat dated, perhaps even obsolete, by the time our book reaches you. This is an inevitable consequence of the hyper-progressive times in which we live. If you wish to advance from additional basic knowledge yet onward, toward the bleeding edge of innovation and discovery, you will eventually need access to state-of-the-art journals and periodicals in visual computing, interpretive visualization, and the sciences. Unfortunately, access to these media often is more restricted, and more costly, for the individual learner than single core texts, but those of you satisfied with nothing short of the absolute frontier will find this section ends with URLs to some of the very highest impact sources, where you can follow the latest developments. We recommend you discuss your interests in accessing such specialized materials with the reference staff of your community library or your local college or university library.

And . . .

A final note: If you are browsing this section before commencing your reading and work with the main text and project material in our book, please remember this is a Further Reading section. It is not a list of the things we expect you to know before you can dive in to *In Silico*. Quite the opposite! We have structured the book to be robustly self-contained once it is in your hands. The sources listed in this part of the book will let you go further still once your core in silico skills are in place—indeed as far as your goals and your imagination beckon.

Interpretive visualization (iVis)

iVis: Fundamentals

Card SK, Mackinlay JD, Shneiderman B, editors: *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, San Francisco, 1999.

Tufte ER: *Envisioning Information*. Graphics Press, Cheshire, CT, 1991.

Tufte ER: *Visual Explanations: Images and Quantities, Evidence and Narrative*. Graphics Press, Cheshire, CT, 1997.

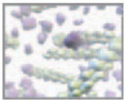
Ware C: *Information Visualization: Perception for Design*, 2nd ed. Morgan Kaufmann, San Francisco, CA, 2004.

iVis: History and philosophy

Anker S, Nelkin D: *The Molecular Gaze: Art in the Genetic Age*. Cold Spring Harbor Laboratory Press, Cold Spring Harbor, NY, 2004.

Baigrie BS, editor: *Picturing Knowledge: Historical and Philosophical Problems Concerning the Use of Art in Science*. University of Toronto Press, Toronto, 1996.

Gamwell L: *Exploring the Invisible: Art, Science, and the Spiritual*. Princeton University Press, Princeton, NJ, 2002.



Kemp M: *Seen/Unseen: Art, Science, and Intuition from Leonardo to the Hubble Telescope*. Oxford University Press, Oxford, UK, 2006.

Waddington CH: *Behind Appearance: A Study of the Relations Between Painting and the Natural Sciences in This Century*. MIT Press, Cambridge, MA, 1970.

Elkins J: *The Domain of Images*. Cornell University Press, Ithaca, NY, 1999.

Pauwels L: *Visual Cultures of Science: Rethinking Representational Practices in Knowledge Building and Science Communication*. Dartmouth College Press, Hanover, NH, 2006.

iVis in Action: Molecules, cells, and tissues

Branden C, Tooze J: *Introduction to Protein Structure*, 2nd ed. Garland, New York, 1999.

De la Flor M: *The Digital Biomedical Illustration Handbook*. Charles River Media, Hingham, MA, 2004.

Goodsell DS: *The Machinery of Life*. Springer-Verlag, New York, 1993.

Goodsell DS: *Bionanotechnology: Lessons from Nature*. Wiley-Liss, Hoboken, NJ, 2004.

Leach AR: *Molecular Modelling: Principles and Applications*, 2nd ed. Prentice Hall, Harlow, UK, 2001.

Pauling L: *The Nature of the Chemical Bond and the Structure of Molecules and Crystals: An Introduction to Modern Structural Chemistry*, 3rd ed. Cornell University Press, Ithaca, NY, 1964.

Pauling L, Hayward R: *The Architecture of Molecules*. WH Freeman, San Francisco, CA, 1964.

Perkins JA: A history of molecular representation, part one: 1800 to the 1960s. *Journal of Biocommunication* 31(1), 2005 (serial online): http://www.jbio communication.org/31-1/features3.html, accessed October 14, 2007.

Perkins JA: A history of molecular representation, part two: 1960s—present. *Journal of Biocommunication* 31(2), 2006 (serial online): http://www.jbio communication.org/31-2/feature2.html, accessed October 14, 2007.

Computers and computing

Architectures (traditional to radical)

Amos M: *Genesis Machines: The New Science of Biocomputing*. Atlantic Books, London, 2006.

Hennessy JL, Patterson DA: *Computer Architecture: A Quantitative Approach*, 4th ed. Morgan Kaufmann, Boston, MA, 2007.

Lloyd S: *Programming the Universe: A Quantum Computer Scientist Takes on the Cosmos*. Vintage Books/Random House, New York, 2006.

Rojas R, Hashagen U, editors: *The First Computers: History and Architectures.* MIT Press, Cambridge, MA, 2002.

History

Aspray W: *John von Neumann and the Origins of Modern Computing*. MIT Press, Cambridge, MA, 1990.

Campbell-Kelly M, Aspray W: *Computer: A History of the Information Machine*, 2nd ed. Westview Press, Boulder, CO, 2004.

Ceruzzi PE: *A History of Modern Computing*, 2nd ed. MIT Press, Cambridge, MA, 2003.

Mindell DA: *Between Human and Machine: Feedback, Control, and Computing Before Cybernetics*. Johns Hopkins University Press, Baltimore, MD, 2002.

Wurster C: *Computers: An Illustrated History*. TASCHEN, Köln, 2002.

Programming languages

Sammet J: *Programming Languages: History and Fundamentals*. Prentice-Hall, Englewood Cliffs, NJ, 1969.

Sebesta RW: *Concepts of Programming Languages*, 8th ed. Pearson Education/Addison-Wesley, Boston, MA, 2007.

Scott ML: *Programming Language Pragmatics*, 2nd ed. Morgan Kaufmann, Boston, MA, 2006.

3D computer graphics and animation

Fundamentals

Foley JD, van Dam A, Feiner SK, Hughes JF: *Computer Graphics: Principles and Practice in C*. Addison-Wesley Professional, Boston, MA, 1995.

Hill Jr. FS, Kelley SM: *Computer Graphics Using OpenGL*, 3rd ed. Pearson Education/Prentice-Hall, Upper Saddle River, NJ, 2006.

Kerlow IV: *The Art of 3-D Computer Animation and Effects*, 3rd ed. John Wiley & Sons, Hoboken, NJ, 2003.

Watt AH: *3D Computer Graphics*, 3rd ed. Addison-Wesley, Reading, MA, 2000.

Learning Maya

Alias Learning Tools: *Learning Maya 7: Foundation*. Sybex, Alameda CA, 2005. (*Learning Maya 7* is a good general introduction from the *Learning Maya* series, a collection of instructional books and DVDs covering every aspect of the Maya modeling, animation, effects, and rendering workflow. While the examples and tutorials do not treat scientific subjects explicitly, the techniques covered are applicable to virtually any subject matter. Titles are listed by category and available to order through the Learning Maya website: www.learning-maya.com.)

MEL programming and Maya's API

Gould DAD: *Complete Maya Programming, vol I: An Extensive Guide to MEL and the C++ API*. Morgan Kaufmann, San Francisco, CA, 2003.

Gould DAD: *Complete Maya Programming, vol II: An In-depth Guide to 3D Fundamentals, Geometry, and Modeling*. Morgan Kaufmann, Boston, MA, 2005.

Stripinis D: *The MEL Companion: Maya Scripting for 3D Artists*. Charles River Media, Hingham, MA, 2003.

Wilkins MR, Kazmier C: *MEL Scripting for Maya Animators*. Morgan Kaufmann, San Francisco, CA, 2005.

Cell science

Fundamentals

Alberts B, Johnson A, Lewis J, Raff M, Roberts K, Walter P: *Molecular Biology of the Cell*, 5th ed. Garland Science, Boca Raton, FL, 2007.

Cooper GM, Hausman RE: *The Cell: A Molecular Approach*, 4th ed. Sinauer, Sunderland, MA, 2007.

Pollard TD, Earnshaw WC: *Cell Biology*, 2nd ed. W.B. Saunders, New York, 2007.

Whitford D: *Proteins: Structure and Function*. John Wiley & Sons, Hoboken, NJ, 2005.

Agents and walkers

Berg HC: *Random Walks in Biology*, 2nd ed. Princeton University Press, Princeton, NJ, 1993.

Gillespie DT: *Markov Processes: An Introduction for Physical Scientists*. Academic Press, San Diego, CA, 1992.

Rudnick J, Gaspari G: *Elements of the Random Walk: An Introduction for Advanced Students and Researchers*. Cambridge University Press, Cambridge, UK, 2004.

Cell migration and the cytoskeleton

Wedlich D: *Cell Migration in Development and Disease*. John Wiley & Sons, Hoboken, NJ, 2005.

Physical principles (biophysics)

Bourg DM: *Physics for Game Developers*. O'Reilly, Sebastopol, CA, 2002.

Dill KA, Bromberg S: *Molecular Driving Forces: Statistical Thermodynamics in Chemistry and Biology*. Garland, New York, 2003.

Howard J: *Mechanics of Motor Proteins and the Cytoskeleton*. Sinauer, Sunderland, 2001.

Jackson MB: *Molecular and Cellular Biophysics*. Cambridge University Press, New York, 2006.

Kennedy J, Eberhart RC, Shi Y: *Swarm Intelligence*. Morgan Kaufmann, San Francisco, CA, 2001.

Lauffenburger DA, Linderman JJ: *Receptors: Models for Binding, Trafficking, and Signaling*. Oxford University Press, New York, 1993.

Murray JD: *Mathematical Biology, vol 1: An Introduction*, 3rd ed. Springer-Verlag, New York, 2002.

Murray JD: *Mathematical Biology, vol 2: Spatial Models and Biomedical Applications*, 3rd ed. Springer-Verlag, New York, 2003.

Schlick T: *Molecular Modeling and Simulation: An Interdisciplinary Guide*. Springer-Verlag, New York, 2002.

Systems biology

Alon U: *An Introduction to Systems Biology: Design Principles of Biological Circuits*. Chapman & Hall/CRC, Boca Raton, FL, 2007.

Murphy MP, O'Neill LAJ, editors: *What Is Life? The Next Fifty Years: Speculations on the Future of Biology*, Cambridge University Press, New York, 1995.

Skyttner L: *General Systems Theory: Problems, Perspectives, Practice*, 2nd ed. World Scientific, Hackensack, NJ, 2005.

Math brush-up

Dunn F, Parberry I: *3D Math Primer for Graphics and Game Development*. Wordware Press, Plano, TX, 2002.

Maynard Smith J: *Mathematical Ideas in Biology*. Cambridge University Press, Cambridge, UK, 1968.

Neuhauser C: *Calculus for Biology and Medicine*, 2nd ed. Prentice Hall, Upper Saddle River, NJ, 2004.

Press WH, Teukolsky SA, Vetterling WT, Flannery BP: *Numerical Recipes: The Art of Scientific Computing*, 3rd ed. Cambridge University Press, New York, NY, 2007.

In silico "Top 10" URLS

Interpretive visualization

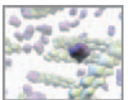
acg.media.mit.edu/people/fry/

www.edwardtufte.com

Computer graphics

www.highend3d.com/maya

www.cgsociety.org



www.red3d.com

www.siggraph.org

Cell and molecular biology

www.molecularmovies.com

cellimages.ascb.org

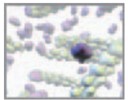
mgl.scripps.edu/people/goodsell/pdb

www.umass.edu/microbio/rasmol/history.htm

The bleeding edge

Annual Reviews	www.annualreviews.org
Cell	www.cellpress.com
The Cell Migration Gateway	www.cellmigration.org/
Journal of Cell Biology	www.jcb.org
Journal of Biocommunication	www.jbiocommunication.org
Journal of Molecular Graphics and Modeling	www.elsevier.com/wps/find/journaldescription.cws_home/525012/description#description
Leonardo	muse.jhu.edu/journals/leonardo/
Nature	www.nature.com
Reviews of Modern Physics	rmp.aps.org
Tissue Engineering	www.liebertpub.com/ten
Science	www.sciencemag.org
The Visual Computer	www.springer.com/west/home/computerimaging?SGWID=4-149-70-1058675-0

Glossary



actin Actin is an essential structural protein (~45 kD) that polymerizes into actin microfilaments, part of the cytoskeleton of most eukaryotic cells.

adenosine diphosphate (ADP) ADP results from one of adenosine triphosphate's (ATP) three phosphate groups leaving ATP, a chemical reaction that results in the release of energy useful in cellular metabolism.

adenosine triphosphate (ATP) ATP is a small molecule which acts as the fundamental unit of energy transport in the cell. It is produced during photosynthesis (in plants) and cellular respiration (in animals).

alpha channel An alpha channel is a grayscale image channel in some image and animation files, which acts as an indicator of image transparency. Usually the light part of the alpha indicates opacity and the dark parts transparency. Adding an alpha channel is a rendering option that adds flexibility in the editing and compositing process.

ambient occlusion Ambient occlusion is a shader effect that simulates the attenuation of light (from ambient sources) on a surface where it comes close to another surface. This effect can increase render times, but will often make lighting appear more photorealistic. Acronym: AO

angiogenesis Angiogenesis is the process of new blood vessel formation. It is a focus of interest in cancer research, since growing tumors require angiogenesis to supply them with blood.

angle of view Also known as field of view, this term refers to the angular extent of an image received by a camera.

animatic An animatic is a preproduction film composed of animated storyboard frames or simple 3D elements arranged and moving as they will in the completed final shot. It is used to iron out timing and camera movement issues before final animation begins. Sometimes also known as the layout stage or a story reel.

animation Animation is the process of bringing inanimate or synthetic objects to life in the medium of film or television by depicting their movement.

animation curve An animation curve is a graphical representation of the change of some attribute over time. Manipulation of the shape of an animation curve can significantly change the perceived quality of an animated object or effect.

animation expression A set of instructions used to animate one or more attributes. Maya's expression language is, for the most part, the same as MEL. Typically, an expression evaluates every time the Maya frame number changes, meaning that the instructions are processed in regular time increments.

aperture The aperture is the width of the opening through which light enters a camera. In 3D graphics (in which there is, of course,

no real aperture) this setting is relevant to the synthesis of depth of field and motion blur effects.

.avi The Audio Video Interleave video format file extension: fileName.avi. This is an older, but still common, Windows video file format.

back light In a typical 3-point light setup, a back light is positioned above and behind the subject, causing a fringe of edge illumination on the subject that helps to distinguish it from the background.

Bezier handle Bezier handles are common in vector illustration programs like Adobe Illustrator and consist of the edit points for Bezier splines that do not reside on the curve itself. Bezier handles are similar to the second and third control vertices in a NURBS curve span in Maya.

boolean modeling A boolean operation in modeling allows one shape to modify another by means of union (one shape plus the other shape), difference (one shape minus the other shape) and intersection (the volume common to both shapes).

boolean values Boolean values are logical operators in MEL that refer to true (1) and false (0) values.

bump map A bump map is a grayscale image map that is used to simulate a textured surface in a surface shader. Bump maps do not distort the underlying geometry; rather, they perturb the surface normals of the underlying object at each pixel in order to create their effect.

breadboard Breadboard is a term derived from electronic circuit design, where physical breadboards are special generic circuit boards that allow for the quick assembly and connection of electronic components. In software engineering it is generally used to describe preliminary versions of code that are designed to be fleshed out into fully working prototypes.

cancellous bone Cancellous bone is the low-density spongy or trabecular bone that occupies the central portions of many medium- and large-size bones.

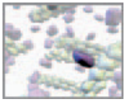
caustic The word caustic, when used in optic or computer graphics contexts, refers to the patterns created by reflected and refracted light; for example, the patterns of light on the bottom of a swimming pool.

CD4 lymphocyte A CD4 lymphocyte is a type of white blood cell essential to immune function; they are also known as helper T cells.

cel A cel—an artifact of traditional animation—is a transparent sheet upon which individual frames of animated activity are drawn. Multiple cels can be layered together over an unchanging background to efficiently partition the work involved in making hand-drawn animation of a

	complex scene in which multiple characters and special effects appear.
cell	A cell is the basic functional unit of life, usually comprising a cell wall or membrane enclosing an aqueous gel of organelles, proteins, and genetic material.
cell cycle	The cell cycle refers to the normal sequence of events in a replicating cell.
central processing unit (CPU)	The CPU is the circuitry that controls and executes software instructions; it is the "brain" of the computer.
CG	Acronym for computer graphics.
CGI	Acronym for computer-generated imagery.
channel	A channel is one part of a digital image file, usually comprising either the red, green, or blue components of a color image. Other channels can be added to an image file, such as an alpha channel (a grayscale channel which codes for transparency) or a depth channel (a grayscale channel that represents distant objects as dark and near objects as light).
chemoattractant	Chemoattractant is a general term for any one of a number of intercellular and extracellular messenger molecules that serve to attract particular mobile cells.
chemotaxis	Chemotaxis is the motion of mobile cells along a concentration gradient (increasing or decreasing) of a chemoattractant substance.
clipping plane	The clipping plane is an arbitrary plane close to and perpendicular to the camera; objects or parts of objects between the camera and the clipping plane will not display or render.
control vertex (CV)	A CV is a point that helps define a NURBS curve in Maya. Generally, a minimum of four CVs are required to create a NURBS curve: the first and last sit on the curve itself, and the middle two influence its curvature.
CPK	Acronym for the Corey–Pauling–Koltun specification for space-filling molecular models.
cytokinesis	Cytokinesis is the final stage of mitosis (cell division) or meiosis (where the parent cell splits into two daughter cells).
cytoskeleton	The cytoskeleton is a dynamic scaffold of long, fibrous protein molecules stretching through the interior of eukaryotic cells, allowing them to maintain shape and change it when necessary. Composed of actin filaments, intermediate filaments, and microtubules, the cytoskeleton is also essential for numerous other cellular processes, from cell movement and the transport of metabolic products within the cell, to apoptosis (programmed cell death).

DAG	Acronym for the Directed Acyclic Graph of a Maya scene, which is Maya's way of efficiently representing the hierarchy of elements in a scene.
debug	"To debug" refers to the process of finding and removing or correcting errors in software code. One etymology of the term finds it dating from the earliest years of computer engineering, when insects fluttered or crawled into the warm, bulky circuitry.
Dependency Graph (DG)	A graphical view in Maya (in the Hypergraph window) that represents the data flow model of scene elements (or nodes) in terms of outputs on one node connecting to inputs on another.
depth map	A depth map is a grayscale-rendered image in which untextured scene objects emerge from a black background and become whiter as they get closer to the camera. Depth maps can be used for various postproduction effects, such as fog and depth of field.
depth of field (DOF)	DOF refers to the range of distances from the camera where objects appear to be in focus; if a scene has a "shallow DOF", then objects outside a narrow plane of focus will appear blurry. In conventional photography, lens variables—such as aperture (f-stop) and focal length—can be manipulated to create shallow or deep DOF effects. A shallow DOF effect is often simulated in CG (either as part of the 3D rendering process or as a post effect), either to increase the photorealism of the image or to selectively focus the viewer's attention within the 3D scene.
deterministic	Deterministic processes are those in which each unique process can be robustly predicted based on its initial conditions and on known inputs to the process as it changes through time (its so-called boundary condition); contrast this with stochastic process (see below) in which, at most, the probability envelope for a whole spectrum of possible outcomes can be predicted.
diffusion	Diffusion is a spontaneous process of particle motion from an area of high concentration to an area of low concentration. It results from the moment-by-moment buffeting of each molecule by collisions with its neighbors.
displacement map	Displacement maps are texture images used to deform surface geometry in a 3D program; usually, light areas of a displacement map push the geometry outwards (in the direction of surface normals), and dark areas inwards.
dot notation	A common form of notation in computer programming where spaces are replaced by periods, for example, objectName.attributeName.
emergent behavior	Refers to complex, adaptive behavior produced in a system by the cumulative effects of many simple interactions.



emitter An object in a 3D program that serves as a source position for the emission of particles.

endocytosis The process in which a cell absorbs molecules from the outside by forming an endocytotic vesicle from its cell membrane.

.eps File extension for the Encapsulated PostScript vector file format.

eukaryote One of the two main lineages of living cell (the other being prokaryotes). Eukaryotic cells have an internal organization characterized by membrane bound organelles, the most prominent being the cell's nucleus.

expression The word expression refers to two things in Maya. The first is a mathematical or logical statement composed of one or more operands (or values) and one or more operators. The second is an animation expression (see above).

extracellular matrix (ECM) ECM is a vital structural component of animal tissues. It is a 3D web of fibrous molecules running outside of and between cells (thus extracellular). The ECM is primarily composed of collagen.

field Abbreviated name for a force field in Maya Dynamics.

field of view See angle of view.

file texture A texture for application to a 3D object that is derived from an image file, such as a TIFF or Photoshop file.

floating point A string of digits that represents a real or decimal number; in other words, a number where the decimal point can be placed (float to) anywhere in the string.

focal length A term from photography and cinematography which refers to the optical power of a lens system. A short focal length lens will have a wide field of view (wide angle) and a long focal length lens will have a narrow field of view (zoomed in).

force field A Maya scene object that exerts some sort of influence (such as gravity or wind) upon distant objects in a dynamic simulation; they can be local or global in their influence.

frames per second (fps) The standard term for the rate at which individual film or video images (frames) are displayed.

function syntax MEL commands can be formatted in function syntax or imperative (command) syntax. Function syntax takes the form:

```
$myVariable = functionName (parameter1, parameter2);
```

geometric primitive Primitives are basic geometric objects, such as spheres, cones, and cubes, which are part of a default set of simple objects that are often used to build more complex objects.

global illumination (GI) An approach to representing realistic direct and diffuse lighting in a computer-generated scene. Any one of a number

of algorithms, or combinations of algorithms, may be used to generate GI, but its hallmark is the representation of the diffuse inter-surface reflection of light characteristic of real-world scenes.

global variable A variable that is global in scope, meaning it's available to all procedures and animation expressions within a Maya scene. Any variables not explicitly declared as global will be available only within the procedure or expression in which they are declared; in other words, they are local variables.

graphics processor Modern computers use graphics processors (sometimes referred to as graphics cards or GPUs for Graphics Processing Units) to accelerate the process of drawing 2D and 3D images on the display device using special hardware circuitry.

haptotaxis The directional migration of living cells, guided by the physical nature of the substrate on which they are moving (typically the substrate's adhesivity for the cells).

hardware render(ing) Rendering performed by the graphics processor. Hardware rendering can be very fast, but not all textures and effects can be hardware rendered.

High-dynamic range imaging (HDRI) HDRI refers to computer graphics formats and imaging methods that allow for the capture of real-world light intensity range and the use of that range in computer graphics applications. Conventional computer-based images (such as 24-bit TIFF files) have a very limited intensity range, while the real world has light intensity contrast range of 60,000:1 or higher. HDRI technologies allow for storing and processing images in that much larger real-world range.

hotkey A hotkey is simply a key on the computer keyboard assigned to execute a particular function when it is pressed.

HSV Acronym for the hue, saturation, value color model which, along with RGB, is one of the default color models in Maya.

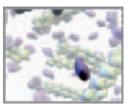
hull Line drawn between control vertices of a NURBS curve or surface. Hulls do not sit on the NURBS curve or surface, but can be selected and manipulated to influence the shape of the curve or surface.

hydrolysis A chemical reaction involving water; it is a type of reaction that occurs when polymers are broken down.

Hypergraph A window in Maya which displays a graphical view of your scene in one of two ways: as a scene hierarchy or as a Dependency Graph.

Hypershade A window in Maya that provides access to materials, textures, lights, and special effects.

hyperthreading Hyperthreading is a technique used on some microprocessors that emulates the presence of a second processor, speeding some threaded operations like rendering.



ID — Acronym for industrial design.

IK — Acronym for inverse kinematics.

imperative syntax — MEL commands can be formatted in function syntax or imperative (command) syntax; imperative syntax takes the form: `functionName parameter1 parameter2`.

in vivo — A term from Latin meaning "in life"; used in science to indicate phenomena observed in a living organism.

in vitro — A term from Latin meaning "in glass"; used in science to indicate phenomena observed in a test tube, Petri dish, or other experimental apparatus.

in silico — A term derived by analogy to in vitro and in vivo to refer to biological processes simulated in computers. Most current computers use silicon-based materials in their hardware circuitry.

interactive photorealistic rendering (IPR) — A rendering mode in Maya that allows for a part of the scene to re-render each time changes are made, speeding the process of fine-tuning materials and lights.

interpretive visualization (iVis) — iVis is the research discipline that deals with the formulation and investigation of new models of pictorial representation in science. The goal of iVis is making visible the principal, relevant elements of organization and change within a model system or data set. iVis is informed by principles of physical theory, computer graphics, human perception, and information design, among other disciplines.

inverse kinematics (IK) — An software approach for determining the correct configuration of a jointed structure given the joint constraints and the position of a terminal joint or effector.

isoparametric curve (isoparm) — A curve component of a NURBS surface (defined by the control vertices), defining the shape of the surface in either the U- or V-direction.

keyboard shortcut — See hotkey.

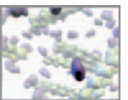
keyframe — In traditional animation, a keyframe is a drawing by the animator of an important or extreme part of an object's movement; frames between the keyframes are drawn by secondary animators known as "inbetweeners". In computer graphics, a keyframe is a point on the timeline where the animator sets object parameters such as location, rotation, or scale; the computer interpolates the changing parameters between keyframes.

key light — In a typical 3-point light setup, a key light is positioned above and to the camera's left of the subject, providing the principal source of illumination.

keyset — Another name for a Maya animation curve.

level of detail (LOD) — The complexity of the displayed representation of a scene or object in a 3D application; often the LOD of the interactive

	workspace is reduced to allow for speedy display, and increased at render time for the creation of final-quality images.
ligand	A molecule which binds (usually through a non-covalent bond) to another, usually much larger, molecule.
local coordinates	A coordinate system measured relative to the transform node of a particular object. World coordinates are measured from the world origin.
lofting	A method for creating a NURBS surface composed of a series of discrete NURBS curves linked in sequence, much like the hull of a ship is defined by its ribs.
Markov process	A stochastic process in which future changes depend on the system's current state, rather than any past states.
Maya ASCII (ma)	A Maya scene file format written in ASCII code via MEL (Maya Embedded Language) script. A Maya ASCII file can be opened and edited in a text editor, independent of the Maya application.
Maya binary (mb)	A Maya scene file format written in binary, or computer machine code. Maya binary files are smaller than Maya ASCII files but cannot be edited in a text editor.
Maya Embedded Language (MEL)	A C-like interpreted scripting language built in to Maya.
memory caching	A method of writing memory intensive operations (such as dynamic simulations or particle positions) to disk in order to reduce the resource demands during rendering.
MolVis	A short form for molecular visualization.
monomer	A single molecule that can be joined by chemical bonds to identical or related molecules to form a polymer.
morphometry	The process of measuring the dimensions and shape of an object.
motion capture (Mocap)	An animation method based on recording the motion of real-world organisms or objects and applying that motion data to objects in a 3D program.
.mov	The QuickTime video format file extension: fileName.mov.
non-photorealistic rendering (NPR)	A catch-all term for rendering approaches which do not seek to emulate photographic images of reality; sometimes used to refer to one particular NPR approach: toon rendering, which is meant to look like traditional cel animation.
NTSC	The analog broadcast television format for North America (as well as parts of South America and Asia). NTSC stands for the National Television System Committee.
NURBS	NURBS stands for Non-Uniform Rational B-Splines, a class of mathematically defined geometric models used widely in computer graphics.



NURBS curve NURBS curves (or splines) are resolution-independent, mathematically defined curves. They are often used as animation paths, function curves, and as the basis for the construction of surface geometry.

PAL The analog broadcast television format for most of Europe, Asia, Australia, Africa, and South America. PAL stands for Phase Alternating Line.

parallax The apparent motion of objects created by the motion of the observer.

parenting The process of creating organized and functional scene hierarchies by making some objects subordinate to others within a scene hierarchy.

particle systems A dynamic system used in a 3D program to simulate phenomena (such as smoke, dust, fire, or fluids) that are composed of great numbers of small particles whose behavior is governed by the influence of forces like gravity and wind.

pencil test Another term for the animatic. This term is more often encountered in traditional, pencil-and-paper hand-drawn animation workflows.

pixelation The appearance of individual pixels in a bitmap image resulting from the image being displayed at size larger than its intended resolution.

photorealism Images created to evoke various photographic effects (highlights, depth of field, realistic shading, and so on) such that they appear as if they could have been produced by a real-world camera.

physics engine A software component that allows for the simulation of physical forces and their effects on an object's shape and motion, taking into account factors such as mass, gravity, friction, and air resistance.

playblast A screen-resolution, hardware-rendered preview of an animation created in Maya.

PLE Acronym for the Maya Personal Learning Edition.

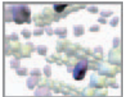
poly count The number of polygons comprising an object or scene. Reducing poly count can lower memory usage and/or decrease the render time of a scene.

polypeptide An alternate term for proteins, which are composed of multiple peptides (amino acid polymers).

post effect An image or visualization effect that is added after the rendering stage in an editing or compositing program like After Effects.

previsualization A preproduction technique in filmmaking where complex scenes are choreographed as computer animation before shooting begins.

procedural animation	Animation that is generated algorithmically, as opposed to being keyframed.
procedural texture	Textures that are generated algorithmically, as opposed to being derived from an image file.
pseudopod (pl. pseudopodia)	A dynamic cytoplasmic protrusion on a living cell, often used for locomotion, sensing the environment, or engulfing foreign matter.
pseudorandom number	A number generated by a computer algorithm—called a pseudorandom number generator (PRNG for short)—that approximates the properties of a true random number. Because it was determined by a mathematical formula, a pseudorandom number cannot be truly random, but can be repeated predictably. Maya's PRNG, the rand() function, generates numbers from the uniform probability distribution.
radiosity	One of a number of algorithms used to generate global illumination renderings.
ramp	A color gradient.
raster graphics	Computer graphics images composed of pixels.
reaction rate	The speed at which a chemical reaction takes place, given the concentration of reactants and environmental constraints such as temperature and pressure.
render layer	In a Maya scene, different elements—mainly models, effects, and lights—can be partitioned into different "render layers". Each layer is rendered separately so that the resulting image depicts only those elements assigned to the layer. The layers are subsequently combined in a compositing program, where special effects can be applied to individual layers. A render layer can also be used to produce a specific "channel" such as color, alpha, shadow, or specular, to name a few. In this latter context, a render layer is sometimes referred to as a "pass".
regenerative medicine	A branch of medical and biomedical engineering research concerned with the potential to repair or restore tissues, organs, and limbs via the use of stem cells, engineered tissues, and genetic engineering.
render engine	A render engine refers to a software component of a 3D animation system, designed to produce finished images from geometry and animation data. Maya, for instance, has four built-in render engines: the Maya Software, Maya Hardware, mental ray, and Maya Vector renderers.
render farm	A render farm is a collection of multiple computers configured to cooperatively render 3D scenes, thus accelerating the rendering process.
rigid body	A dynamics simulation object that behaves as if it were made from a rigid material like hard plastic or steel.



scene In Maya, a scene is the 3D environment, including models, animation, lights, and cameras, contained in one computer file.

scene hierarchy A view in Maya's Hypergraph window which visually represents the ranking of scene objects within nested parent-child relationships, depicted as an outline-like ordering of object node boxes. Parent objects control various parameters of their child objects, and child objects themselves can be parents of other objects.

scope The domain in which a variable operates: local variables operate only within the procedure or expression in which they are declared; global variables are global in scope, meaning they're accessible to all procedures and expressions operating within a Maya scene.

search path The directory path or paths that Maya looks to for executable MEL scripts; directories can be defined as part of the search path in the Maya.env file.

soft body A dynamics simulation object in Maya which behaves as if it were made from a flexible material like cloth or gelatin.

spline See NURBS curve. The term spline originated in shipbuilding where it described a thin piece of wood used to define hull shape. The spline was bent into a smooth curve by metal weights—the equivalent of control points on a Maya spline.

stochastic A process that displays random or probabilistic behavior.

story reel In some studios an animatic is called a story reel.

sub-surface scattering Sub-surface scattering is the name for a shading effect that simulates the penetration, scattering, and re-emission of light in semi-transparent or translucent surfaces. Its use can increase the realism of depiction of surfaces such as wax, milk, or skin.

tangent Another name for a Bezier handle in Maya.

tessellation In CG, tessellation refers the pattern of polygons created (as the result of a modeling operation) in order to specify a renderable surface. In Maya, for example, NURBS models are ultimately tessellated for rendering, and changes to the tessellation settings can affect the quality of the rendered result.

texture mapping Texture mapping refers to the application of 2D images or procedural textures to the surface of a model to change its appearance when rendered.

three-point (3-point) lighting A traditional photographic approach to lighting, consisting of a "key" light (to provide the main source of illumination), a "fill" light (to reduce the harsh shadows created by the key light), and a "back" light (to create edge illumination on the object of interest).

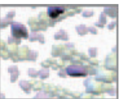
toon rendering	Toon rendering (which is a contraction of "cartoon" rendering) is a non-photorealistic rendering approach that mimics the linear outline and flat color fills of traditional cel animation.
treadmilling	Treadmilling is a term for a dynamic behavior of cytoskeletal elements (such as actin filaments). In treadmilling, the filament appears to be constant in length; in reality, actin monomers are disassociating from the minus-end, and attaching to the plus-end.
van der Waals radius	A radius that defines the "contact surface" of an atom. For instance, the van der Waals radius of a carbon atom is 1.7Å. Acronym: vdW radius.
vector	A 1D data array, specifying direction and magnitude.
vector graphics	Images that are comprised of resolution-independent vector descriptions of graphical objects.
virion	A whole virus particle.
virtual memory	An operating system technique that uses part of the hard disk to simulate the presence of additional random access memory (RAM). Virtual memory allows the effective expansion of the usable amount of memory over the amount of RAM hardware circuitry installed in the computer.
visualization	The act of creating a clear visual impression or representation of something, whether in the mind or in some form of external visual media.
.wmv	The Windows Media Player video format file extension: fileName.wmv.
world coordinates	A coordinate system as measured from the world origin. Local coordinates are relative to the transform node of a particular object.
world origin	The center of the virtual world in Maya, where the value on all three spatial axes is 0, 0, 0.

This page intentionally left blank

Index

".mel" file extension, 412
.state attribute, 416, 430
".txt" file extension, 412
  for animation expression files, 412
"\n", new line character, 319, 335
\n break notation, 466
1j6z.pdb, 351, 352, 396
2D animatic, 57–8
2D cell research, 532, 533
2D protein array, 384
2D texture, 196
  see also Procedural texture
3-point lighting rig, 235, 240
3D animatic, 66
3D Brownian diffusion, 399
3D camera, 64
  movements of, 93
3D cell research, 532, 533
3D protein arrays, 384
3D scene, 58, 60
3D Studio Max, 16
3D texture, 196
  see also Procedural texture
5-state Markov process, 528, 529

Abyss, The, 15
Actin, 346–8
  F-actin, 386, 389–90, 402
  G-actin, 386, 395–7, 422
  model, 370, 371, 373
Actin depolymerizing factor (ADF), 392
Actin filaments, 11, 347, 348, 385, 390, 402
Actin geometry, 394
  F-actin model, 397–9
  G-actin template model, 395–7
Actin protein model, 370, 371
Actin reactions:
  bound nucleotides, 390-1
  reaction rates, 391–2
Active panel, 88
`AddAttr` command, 465, 496
Adenosine diphosphate (ADP), 390–1
Adenosine triphosphate (ATP), 346, 347, 390
  CPK model of, 368–9
Adobe After Effects, 66, 259
Advanced rendering techniques:
  with mental ray, for Maya renderer, 249
    ambient occlusion, 250, 251
    caustics, 250
    global illumination, 250, 251
    image-based lighting, 251–2
    realism about photorealism, 252
    Render Layers, 252
    subsurface scattering, 250
Agent-oriented programming (AOP), 40
Aim locator, 225
Airy's differential equation, 24
Alias|Wavefront (A|W), 14
Alias Research, 14, 15
Alpha channels, 194, 220
Ambient Color, 206
Ambient occlusion (AO), 64, 250, 251
Amino acids, 8
Angle of View, 217, 219
Animation, 61, 138
  animator's workflow, 49
    postproduction, 66–7
    preproduction, 51–8
    production, 58–66
  and film perception, 46–9
  deleting keys, 144
  and film perception, 46–9
  graphing, 142–3
  keyframe animation, 145–51
    vs. procedural animation, 138–9
  menu set, 139–40
  micrographic look, 52–3
  nodes:
    in Hypergraph and Attribute Editor, 151
  non-photorealistic looks, 53, 55–6
  photorealistic look, 52
  playback settings, 145
  procedural animation, 151–4

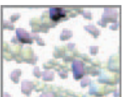


Animation (*contd.*)
setting keys, 140–1
time units, 144
Animation assistants, 62
Animation controls, 146, 147
Animation curve, 62, 143, 155
editing, 147–8, 149
see also Interpolation
Animation expression, 151–2, 278, 292, 309
creation, 153–4
expression command, 297–8
Expression Editor, 295
converting units, 296
Create and Edit buttons, 296
expression node, 139, 154, 294–5
expression syntax, 296–7
line breaks in, 300–1
stand-alone animation expressions, 298–300
Animation nodes, 151, 152
Animator's workflow:
postproduction, 66–7
preproduction, 51–8
production, 58–66
Apparent motion, 48, 49
Apple, 15, 66
Apple key, *see* Command key
Application-oriented programming language, 266
Application Programming Interface (API), 16, 72, 262
Area light, 236, 237
Array index management, 29
Arrays, 275–6
molecular arrays, 10
Assemblers, 25, 26
Assembly languages, 25, 28
associate() procedure, 425–8
Association reaction, 392, 401, 407, 421
Association reaction rate, 404–7
ATOM, 352, 359, 360
Atoms, 8, 348–9
creation, in CPK model, 362–4
as spheres, 350
Attribute data types, 287
Attribute Editor, 79, 80, 96, 98, 99, 124–6, 144, 151, 535
see also Camera Attribute Editor
Attributes, in MEL:
getting, setting, and connecting, 286–7
string attributes, 287
type flag, 287
Auto keyframing, 142
Autodesk, 14, 72
Autodesk 3ds Max, 313, 578
Autodesk MotionBuilder, 313
Autodesk StudioTools, 15
Autodesk VIZ, 313
Avoidance vectors, 400, 190, 536
Axis indicators, 93

Back light, 63, 235, 236, 238
Backslash character, 275, 300, 316
Backus, John, 29
Bacterial flagellum, 10, 11
Barbed end, 347, 389–90
BASIC, 28
Batch Rendering, 244–5, 252
common Render Settings, 253–5
Maya Software Render Settings, 255–6
Render, 256–7
software vs. hardware rendering, 257
Berry, Drew, 201
Bézier handles, 148
Binary digit, 23
Binary string, 25
Binary switching, 38
bindPose node, 468
Blank lines, 283
Blank spaces, 283, 319
Blender, 578
Blocks, 278
Boolean, 116
Bounding Box, 90, 490–1
Bounding vector, 422, 423, 503
Brinsmead, Duncan, 201
Brownian motion, 172
Brownian random walks, 393
Brushes, 331, 333
Buckminsterfullerene, 314
BuckyBall, 314
bump channel, 206
Bump maps, 198, 199

C++ Application programming interface (C++ API), 28, 40, 312, 579
C++ plug-in, 580
Camera attributes, 217, 218–19, 222
Camera Display Options, 221
Camera movements:
Dolly, 93
keyboard/mouse combinations, 93
tracking, 93
tumbling, 93
Cameras, 64, 216, 217
creation, 223
two-panel view set up, 223–4
Camera Attribute Editor:
camera attributes, 218–19
Depth of Field attribute, 219–20
Display Options, 221

Environment attributes, 221
Film Back attributes, 219
mental ray attributes, 219
Movement Options, 221–2
orthographic views, 222
Output Settings, 220
Cast shadows, 209, 232, 234
Caustics, 246, 250, 252
CD4 lymphocytes, 322
ceil() function, 288–9
Cell shader technique, 65
Cell-cell signaling, 449, 524–5, 526, 536–8
Cell crawling, 445–6, 454, 455, 456
algorithm, 453, 525, 526, 528
fast and slow movers, 448
model definition:
cell behavior, 450
cell model, 449–50
cellular scale, 450–1
chemotactic signal, 450
substrate, 450
navigation nomenclature, 448–9
protrusion nomenclature, 448
pseudopod generation, 451–3
retraction, 448
and walks, 446–8
Cell locomotion model:
algorithm design, 453, 455, 456, 457
algorithm encoding:
cellCrawl expression, 466–75
cell crawling, 445–6
and walks, 446–8
data I/O, 476
geometry, 457
cell deformation, 463–4
cell resetting, 464
cell rigging, 461–2
cell shaping, with Lattice Deformer, 459, 460
cell's position, resetting of, 461
custom attributes addition, 464–5
history deletion, 459, 461
joints, 462–3
joints to skin, binding, 463
shader, creation and application, 459
smoothening, 458–9
model definition:
cell behavior, 450
cell model, 449–50
cellular scale, 450–1
chemotactic signal, 450
substrate, 450
navigation nomenclature, 448–9
protrusion nomenclature, 448
pseudopod generation:
animation, using joints, 451–3
troubleshooting, 476–7
Cell micrographs, 53
Cell migration:
as emergent behavior, 521–3
model definition:
boundary conditions, 527
cell behavior, 525
cell-cell signaling, 526
cell geometry, 526
spatial and temporal scale, 527, 528
substrate, 526, 527
model design, 528–38
nomenclature, 524–5
in scaffolds, 523
simulation model, 539
Cell migration data, 322–3
visualization, 325
algorithm encoding, 326–7
algorithm planning, 324, 326
animation, playing, 337
data file, 322–3
data visualization, 323–4
debugging, 337
moveCells.txt, 327–36
script running, 336–7
spatial and temporal scales, 323
summary report, 324–6, 335
Cell organization, 10–11
CellCenter, 461–2, 474
cellCrawl expression, 466
cell model, resetting, 469–70
crawl cycle increment, 474–5
crawl cycle setup, 470–4
print commands, 470
variables initialization, 468
Cellular scale, 450–1
Channel box, 97–8, 113, 144
and sphere transformation, 116–17
Channel Control editor, 98
Character rig, 444, 451–2
Checker texture node, 194, 195
Chemoattractant, 449, 547
Chemokinesis, 448
Chemotactic signal, 450
Chemotaxis, 448–9, 451, 524, 535–6
Cinema 4D, 16, 578
-clear flag (-cl), 472
Clip planes, 219
closestPointOnSurface (cpos) node, 400, 434, 533, 534
closestSurface, 542

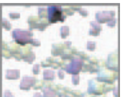


collide() procedure, 400, 432–4
Collision forces, 161
Collision vector, 421, 422
Color Chooser, 205, 206
Color wheel, 378
Colored lights, 378
colorPP attribute, 173
Command argument, 282
Command key, 74
Command Line, 94, 95, 266, 267
Command line render, 244, 245
Comments, 270, 327–9
Compatible Time-Sharing System (CTSS), 36
Compiler, 28, 29–30
Compositing plan, 65
Computational biology, 22, 35
  compiler, 27–8, 30
  conditional control, 33–4
  high level programming languages, 26–7
  information and process, 22–3
  interpreter, 27–8
  language and program, 23–5
  low level programming languages, 26–7
  OOPs and agents, 39–40
  stored programs, 30–3
Computer-based interpretive visualization, 35
Computer-generated CPK model, 349, 351
Computer-generated imagery (CGI), 4, 14, 15, 263
Computer graphics (CG), 4
Concentration volume, 394, 404
Conditional control, of program execution, 33–4
Conditional statements, 288–9
Confocal microscopy, 53
Conformational changes, 388, 397
connectAttr, 297
Connection Editor, 79, 127–8
Construction history, 80–1
  geometry modeling, 122–8
Container, particles in:
  attributes, 174–5
  collision, between particle and cylinder, 176–7
  container creation, 173
  emitter node, 175
  inter-particle collisions, 178–81
  particle data caching, 183–4
  particle emitter creation, 173–4
  particle motion, 178
  particle shape node, 175–6
  per particle color, 181–3
Control vertex (CV), 104, 452
  and polygon primitive deformation, 120
  and sphere deformation, 118
CPK models, 344, 345, 348–51, 368–72, 371, 395,
cpk() procedure, 356–7, 368
Create button, 296
Create menu, 102–3
Creation options, 103
Critical concentration (C_c), 392
cRule1(), 554–8
cRule2(), 558–60
cRule3(), 552, 561–4
Curl, Robert, 314
CurrentTime MEL command, 297
Curve degree, 130
Curve flow, 163–4
cycleCheck, 170–1
Cytoskeleton, 10–11, 12, 347, 348, 384

Dalton (DA), 344
Dalton, John, 342
Data conversion, 273–4
Data input/output, 312
  cell migration visualization:
    algorithm encoding, 326–7
    algorithm visualization, 324–5
    animation, playing, 337
    data file, 322–3
    data visualization, 323–4
    debugging, 337
    moveCells.txt, 327–36
    script running, 336–7
    spatial and temporal scales, 323
    summary report planning, 324–6, 335
  MEL, for reading and writing files, 315
    data reading, 318–20
    data writing, 320–2
    file, opening and closing, 316–18
    file path, 316
  translators, 313–15
Data management plan, 65–6
Data reading, using MEL commands:
  feof command, 320
  fgetline command, 318
  fgetword command, 319
  fread command, 319
  frewind command, 318
Data visualization, 323–4
Data writing, using MEL commands:
  fflush command, 321–2
  fprint, 320–1
  fwrite, 321
dataOutput expression, 572–3
De Humani corporis Fabrica, 4

Debugging, 267, 306–8, 371–2, 440–1
Default light, 63, 90, 232
DefaultValue, 465
Deoxyribonucleic acid, *see* DNA
Dependency Graph (DG), 16, 161, 192, 263, 459
  and biology, 82
  and DG nodes, 78–81
Depth channel, 220, 254
Depth Map Shadows, 234, 378–9
`Depth of Field` attribute, 219–20
Dermal ECM scaffold, 481
  model definition, 483
    fiber orientation and intersections, 485
    fiber size distribution, 484–5
    scaffold dimensions, 484
Design-space, 6, 7
Device Aspect Ratio, 226
DG nodes, 78–80, 99
`Diffuse` channel, 207
`diffuse()` procedure, 400, 431–2
Diffusion vector, 400, 422
Diffusive motion, 393
Digital Equipment Corporation (DEC), 26
Directed Acyclic Graph (DAG), 81
  and biology, 82
Directedness coefficient (D_c), 324, 335
Disney, Walt, 15, 46
Disney-style animation, 57, 61
Displacement maps, 198, 199
Displacement materials, 193
Display layer, 97
Display Options, 221
`dissociate()` procedure, 403, 410, 434–7
Dissociation reaction, 392
Dissociation reaction rate, 408–9
DNA, 10, 11, 342
  computer-generated CPK model of, 349
  plastic CPK models of, 349
`do…while` loop, 290–1
Dolly, and camera movement, 93
Domain-oriented programming language, 265
Dope Sheet, 142–3, 144
Dot notation, 153, 286
Double transformation, 133, 505
Downstream node, 79, 123, 127
Dream Quest Images, 15
Dynamation, 15, 16
Dynamic Relationships Editor, 165, 166
Dynamic simulations, *see* Dynamics
Dynamic typing, 273
Dynamics, 158, 160
  collision forces, 161
  container, particles in, 173–84
  Dynamic Relationship Editor, 165, 166
  Dynamics engine, 161
  fields, 161–2
  `nCloth`, 165
  Nucleus, 165
  particle objects, 162–4
    attributes, 162
    curve flow, 163–4
    emitters, 162, 163
    goals, 162–3
    rendering, 163, 164
  rigid bodies, 164, 166–72
  soft bodies, 165

E-Cell, 579
Ease in, spline interpolation, 143
Ease out, spline interpolation, 143
ECM scaffold growing, 480
  algorithm design, 492, 493
    fiber axis, 486
    fiber surfaces, 492
    fibers randomization, 491
    NURBES spheres, 486–7
    rule-based design, 487–91
    seeds resetting, 491
    timeline, modeling with, 487
  algorithm encoding:
    `makeSeeds()` procedure, 494–7
    `moveSeeds` expression, 500–6
    `resetSeeds` expression, 497–500
    `rule` procedures, 507–12
  dermis:
    fiber orientation and intersections, 485
    fiber size, packing density, shape, 484–5
    parameters of, 481–3
    scaffold dimensions, 484
  expressions creation, 513–14
  Maya scene preparation, 512
  model playing, 515–16
  parameter effects, 517–18
  scaffold parameters inspection, 515, 516
  scene preparation, 514–15
  script elements sourcing, 512–13
  seeds making, 513
Edit button, 296
Edit mode, 282
Edit Points (EPs), 104–5
EDSAC, 24, 25, 33, 34, 35
Electronic Systems Laboratory (ESL), 36
Emitter, 161, 162, 163, 175
Endocytosis, 385
Endothelial cells, 448
Environment textures, 196
EP Curve Tool, 131
Exit on Completion, 103



Explanations, 576
Explicit typing, 273, 415
expression command, 297–8
Expression Editor, 151, 153, 293, 295–6
Expression node, 139, 154, 294–5
Expression syntax, 296–7
Extracellular matrix (ECM) proteins, 165
Extrude tool, 132, 486

F-actin, 386, 402
 model, 397–9
 reactions, 390–2
 structure of, 389–90
 see also Actin
Fade in, 423
Fade out, 423
faderShader() procedure, 410, 429–31
Far Clip Plane, 514
FBX, 313
fCheck, 257
 playback in, 258–9
 saving from, 259
fclose command, 321
fgetline, 318
fgetword, 318, 319, 359
Fields, in Maya, 161–2
 radial field:
  Manipulator Tool for, 178–9
  particle object, connection with, 179
  particles, as source, 179–81
 Turbulence field, 170, 172, 178
File opening and closing, 316–17
File browser, 317–18
"File end-of-file" (feof) command, 320
File header, 304
File path, for reading and writing, 316
File Texture, 196–7
fileBrowserDialog command, 317
fileDialog command, 358
fileTest command, 316
Fill light, 63, 235–6, 237
Fills, 247
Film Back attributes, 219
Filmmaking, 51
 script treatment in, 56–7
Final Cut Pro, 66
Flags:
 and default tool settings, 281–2
 for extrude command, 505
Flash Player, 247
Flat Shade All, 90
Flat Shade Selected Items, 90
Flicker fusion, 48, 49
Floating point, 116, 270
Focal Length, 217, 219
Footage, 64, 66
fopen command, 316, 317, 327
for-in loop, 289, 290
for loop, 289–90
Force attribute, 181
-forceElement (-fe) command, 424
Fortran, 28, 29, 90
Four panel view, 112, 114
fprint, 320–1
Frame All, 224
Frame rate, 61
Frame Selection, 224
frame variable, 153, 297
fread, 318, 319
frewind, 318
Function syntax, 282–3
Future of Maya, 579–82
fwrite, 320, 321

G-actin protein, 386
G-actin template model, 395–7
gauss command, 400
GeoConnector node, 177
Geometric primitives, 102–3
Geometry modeling, 58–9
getAttr command, 297
Gillespie algorithm, 530
Global illumination (GI), 64–5, 250, 251
Global procedure, 292
Global variable, 276–7, 498
Glue program, 265
Graph Editor, 142, 143, 147–8
 graph view, 148–50
 keys, for move and deletion, 150–1
 outliner, 148
 toolbar, 150
Graphics Processing Units (GPUs), 64
Graphics processor, 163, 190
Guide spline:
 drawing, 130–1
 profile snap, 132

Haptotaxis, 449, 524–35
Hardware Render Buffer (HRB), 246
Hardware renderers, 163, 176, 246
Hardware rendering, 190
 vs. software rendering, 246
Hardware Texturing, 90
heightRatio attribute, 414, 416, 417
Help Library, 73, 74, 89–90, 268, 274, 284, 291, 295, 309–10, 458
Hemoglobin, 204
High dynamic range imaging (HDRI), 251, 252
High-level programming languages, 26–7
High-LOD model, 395
History icon, 110
Holmes/Lorenz model, 396

Home-built tools, 16
Hotbox, 85–6, 168
Hotkey, 74–5
Hourihan, Jim, 15
HSV (Hue, Saturation, and Value), 205
Hulls, 105
  scaling, 118–19
Human sperm cell, 3D computer model of, 445
Hydrolysis, 390
  and phosphate release rates, 402, 407–8
Hypergraph panel, 91
Hypershade, 190, 192
Hyperthreading, 246

In silico approach, 4, 14, 42, 309
In silico biology, 42, 147, 155, 160, 222
In vitro methods, 4
In vivo methods, 4
IBM, 7094, 30
if...else statement, 288
if statement, 331, 420
Image-based lighting (IBL), 251–2
Image File Output, 253–4
Image Plane, 221
Image Size, 254–5
Imperative syntax, 282, 283
Impulses, *see* Collision forces
Incandescence, 206
Industrial Light and Magic (ILM), 15
Initial State, 170
Interactive Creation, 103
Interactive Photorealistic Rendering (IPR), 202, 239–40, 376
internalVar command, 326
Inter-particle collisions, 178–81
Interpolation, 62, 143
Interpreter, 27, 264
  vs. compiler, 28
Interpretive visualization (iVis), 8, 17–19
Isoparms, 105

Jaggies, 255
Janczyn, Joyce, 273, 284
Jmol, 17
Joint tool, 462
Jurassic Park, 15

Keep Image, 203
Key-framed animation, 61–2
Key light, 63, 235
Keyboard/mouse combinations, 93
Keyboard shortcut, *see* Hotkey
Keyframe animation
  curves editing, 147–8
  Graph Editor graph view, 148–50
  Graph Editor outliner, 148
  Graph Editor toolbar, 150
  keyframe setting, 146–7
  keys setting, 140–1
  moving keys, 150–1
  play, scrub, and stop, 147
  preparation, 145–6
  vs. procedural animation, 138–9
Keyframes, 61, 138
  and memory, 139
Keysets, *see* Animation curve
Kinemation, 15, 16
Kroto, Harold, 314

Lambert shader, 204–5
Lamellopodia, 448
Lattice Deformer, 117, 459
Lattice points, 459, 460
Layer Editors, 96–7
Layered texture node, 196
Layouts, 91
Level of detail (LOD), 344–5, 395
Ligand, 449
Light linking, 238, 239
Lighting, 232, 233, 234
  for hemoglobin scene, 235
    IPR, previewing with, 239–40
    Light linking, 238, 239
    lights creation, 236
    lights placement, 237, 240–1
    shadow casting, 238
  shadows, 232, 234
Lighting menu, 89, 90, 234
Lights, for animation:
  back light, 63, 235, 236
  fill light, 63, 235–6
  key light, 63, 235
Lightwave, 578
Line breaks, in animation expressions, 268, 300–1
Linear interpolation, 143
Local axis, 93
Local coordinates, 92–3
Lofting, 105
Logic errors, 307–8, 371–2
"Look", for cell science animation:
  micrographic look, 52–3, 54
  non-photorealistic looks, 53, 55–6
  photorealistic look, 52
Look Through Selected, 91
Loops, 289
  do...while loop, 290–1
  for loop, 289–90
  for-in loop, 290
  while loop, 290
Low-level programming languages, 26–7
ls command, 358, 423

Lymphocytes, 165, 448
Lysozyme, structure of, 344

Mac OS equivalents, 74
Mac OS X, 259, 454, 476–7
Machine language, 23
Macromolecular self-assembly,
see Self-assembly, of macromolecules
Main Menu bar, 83–4
makeCells() procedure, 540–5
makeSeeds() procedure, 491, 494–7
Manipulator Tool, 178–9
Marking menus, 84–5
Markov process, 528, 529, 530, 531
Material marking menu, 207
Material node, 193–4
Mathematical and logical expressions, 277
blocks, 278
operators, 278–80
Maya, 1, 13, 16, 263, 578
construction history, 80–1
dependency graph and DG nodes,
78–80
DG, DAG, and biology, 82
explanations, 576, 577
future, 579–82
getting started, 72
Help and instructions, 73, 74, 75
hotkeys, 74–5
Maya Complete, 72
Maya Personal Learning Edition, 72
Maya project, 76–8
Maya Unlimited, 72
Release notes, 74
scene file, 76
start Maya, 75
system requirements, 73
user profile, 75, 76
history, 14–16
and interpretive visualization, 17
program architecture, 78
MEL, 17–19
scene hierarchy and DAG, 81–2
simulations, 576
user interface (UI), 82
Attribute Editor, 98
Channel Box, 97–8
Channel Control editor, 98
Command Line and Script Editor, 95
Layer Editors, 96–7
Main Menu bar, 83–4
menus, working with, 84–6
Outliner, 87–8
playback controls, 95
Plug-ins, 98–9
Preferences, 95–6
Range slider, 94, 95
scene viewing, through camera,
93–4
shelves, 86–7
Status Line, 86
Time Slider, 94, 95
title bar, 83
workspace and panel menus,
88–91
XYZ coordinate system and vectors,
91–3
workflow, 14
Maya, 2008, 72, 313
Maya ASCII, 76, 77
Maya-based workflow, 578
Maya Binary, 76, 77
Maya cameras, *see* Cameras
Maya Complete, 72
Maya Dynamics, *see* Dynamics
Maya Embedded Language (MEL), 16, 72,
262, 301–2
animation expressions, 292
expression command, 297–8
Expression Editor, 295–6
expression node, 294–5
expression syntax, 296–7
line breaks in, 300–1
stand-alone animation expressions,
298–300
attributes in, 286–7
building, 302–6
conditional statements, 288–9
commands, 315
debugging, 306
logic errors, 307–8
syntax errors, 307
loops:
do...while loop, 290–1
for loop, 289–90
for-in loop, 290
while loop, 290
mathematical and logical expressions,
277
blocks, 278
operators, 278–80
Maya's search path, 269
MEL, *see* Maya Embedded Language
(MEL)
MEL command, 280
blank lines, 283
blank spaces, 283
command modes, 282
flags and default tool settings, 281–2
function syntax, 283

functions, 283–4
imperative syntax, 282, 283
reference library, 284
shelf button from, 284–6
top 10 commands, 284, 285
MEL input:
Command Line, 266, 267
Script Editor, 267–8
sourcing, 268–9
origins, 263–4
procedures, 291–2
syntax:
CASE sensitivity, 270
comments, 270
quotation marks, 270
statement termination, 269
terminology, 264–6
values, 270
variables:
arrays, 275–6
data conversion, 273–4
declaration and assigning, 271–2
dynamic typing, 273
global variables, 276–7
matrices, 276
naming, 271
strings, 274, 275
type casting, 274
vectors, 274
MEL interpreter, 27, 262
MEL statement, 262
Maya Hardware renderer, 244, 246, 257
Maya link library (.mll), 313
Maya materials, 193–4
Maya Personal Learning Edition (PLE), 72
Maya program architecture, 78
Maya project, 76–8
Maya scene, 58, 77, 78, 512
Maya Software renderer, 244, 245–6, 372
anti-aliasing quality, 255–6
Maya Unlimited, 72
Maya Vector renderer, 244, 246–8
Memory caching, 171–2
mental ray, for Maya renderer, 244, 248–9
advanced rendering techniques with, 249
ambient occlusion, 251
caustics, 250
global illumination, 251
image-based lighting, 251–2
realism about photorealism, 252
Render Layers, 252
subsurface scattering, 250
Micrographic look, for cell science animation, 52–3
Minicomputers, 26, 36
Minus end, 390, 398
Mobile cell populations, 521–2
Model type, 59, 109
Modeling geometry, 102
construction history, 125–7
Attribute Editor, 124–5
Connection Editor, 127–8
Hypergraph revisiting, 122–4
NURBS modeling:
NURBS curves (splines), 104–5
NURBS “fiber”, *see* NURBS “fiber” creation
NURBS surfaces, 105–7
surface menu set, 103, 104
NURBS primitive modeling, 109–17
polygon primitive deformation, 119–22
polygonal modeling, 107
menu set, 108
subdivision surfaces, 109
sphere deformation, 117–19
Molecular arrays, 10
Molecular visualization (MolVis), 345–6
Molecules, 8
Motion capture (Mocap), 312
Motion path, 227–8, 227, 228, 260
Motion perception, 48–9
Move Tool, 116
Move Tool Manipulator handles, 113–14
moveCells() expression, 548–54
moveCells.txt, 237
moveSeeds, 487, 490–1, 500–6
Multiscale Models tool, 396
Myosin, 11, 347

National Television System Committee (NTSC), 48, 144
Navigation nomenclature, 448–9
nCloth objects, 165
Neuron, 47, 480
Non-photorealistic looks, 53, 55–6
Non-photorealistic rendering (NPR), 55–6, 189, 190
Non-Uniform Rational B-Splines (NURBS) modeling, 59
NURBS surfaces:
components, 105–6
normals, 106–7
splines:
components, 104–5
surface menu set, 103, 104
Nucleus, 165, 424, 465
NURBS curves, *see* Splines

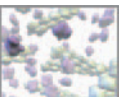
NURBS "fiber" creation:
guide spline:
and profile snapping, 132
drawing, 130–1
profile spline creation, 131
scene view set up, 129–30
surface extrusion, 132–3
tube alteration, through history connections, 133–4
NURBS primitive modeling:
sphere:
creation, 110–11
deformation, 117–19
Four panel view, 114
Move Tool settings, 116
renaming, 113
rotation, 114–15
scaling, 115–16
selection, 111–12
transformation, using channel box, 116–17
translation, 113–14
NURBS surface, 59, 105–7

objExists command, 331
Occlusion, 251
Omni light, in Cinema 4D, 63
Operator overloading, 280
Operators, 278–80
Option boxes, 86, 110
Option key, 74
Organizational hierarchy:
amino acids, 8, 9
atoms and molecules, 8
cell organization, 10–11, 12
DNA, 10
micro to macro, 12
molecular arrays, 10, 11
proteins, 10
tissues and organs, 11–12, 13
Origin Axis, 92
Orthographic camera, 91, 217, 218
Orthographic views, 114, 222
Outliner, 81, 87–8, 398
Output Settings, 220
Overscan, 221, 226

Paint effects, 201–2
Pairing reaction and diffusion:
diffusion time, 393
reaction time, 392–3
PAL, 144, 254
Panel, 91
Panel Editor, 91
Panel menus, 88–9
Panels menu, 89, 90–1
Parameters, 532
of cell migration, 522–3
of dermis, 481–3
effects, 516–17
model variations, 570–1
Particle attributes, 162
Particle data caching, 183–4
Particle emitters, 162, 163
creation, 173–4
Particle shape node, 175–6
Particles, 162–4, 165
attributes, 162
in container, 173
attribute settings, 174–5
collision with cylinder, 176–7
emitter node, 175
inter-particle collisions, 178–81
motion randomization, 178
data caching, 183–4
emitter creation, 173–4
per particle color, 181–3
shape node, 175–6
curve flow, 163–4
emitters, 162, 163
goals, 162–3
rendering, 163, 164
Pencil Curve Tool, 131
Pencil tests, 58
Per particle attributes, 162, 176, 287
Per particle color, 181–3
Per particle expression, 182–3
Persistence of vision, 48
Perspective camera, 91, 217–18, 229, 337, 438
Phase-contrast microscopy, 53, 54
Photorealism, 52, 64–5, 252
Photorealistic look, 52
Physics engine, 17, 62
Pixar, 15
Pixelation, 196
Play every frame, 145, 146, 457
Playback:
settings, 145
using fCheck, 257–9
Playback control, 95, 258
Playback Speed, 457
playbackOptions command, 329, 334
Playblast, 160, 517
creation, 184–5, 229
Playfair, William, 4
Plug-ins, 78, 98–9, 312, 314
Point light, 63, 236
Pointed end, 389–90
pointOnCurve, 434, 504
pointOnCurveInfo, 434
pointOnSurface, 434, 565

pointOnSurfaceInfo node, 434, 533, 535, 543, 565, 573
Points, 90
Poly count, 107
polyCube, 458
Polygon menu set, 108
Polygon primitive deformation, 119
  control vertex, moving, 120
  edge, moving, 121
    sphere, smoothening, 121–2
  polygon face, moving and scaling, 120–1
Polygon sphere, 106
  construction history:
  creation, 119–20
Polygon surface, 15, 58, 107, 120
Polygonal modeling, 107
  menu set, 108
  subdivision surfaces, 109
Polygonal surface model, 396
polySmooth, 458
polySmoothFace node, 121, 134
polySphere, 124, 281, 282, 458
polyTweak, 124, 126
Popup Help, 73, 208
posi, *see* pointOnSurfaceInfo node
Position markers, 227–8
Postproduction stage, in animator's workflow, 66–7
PowerAnimator, 15, 16, 263
Preconfigured shading network, 201
pPlane1, 210, 211
Preferences settings, 95–6
Preproduction stage, in animator's workflow, 51
  2D animatic, 57–8
  animation's "look", 52
    micrographic look, 52–3
    non-photorealistic looks, 53, 55–6
    photorealistic look, 52
  storyboard, 57
  treatment and script, 56–7
print command, 299, 300, 308, 372, 470
Procedural animation, 61, 62
  expression, 151–2
    creation, 153–4
    nodes, 153
  vs. keyframe animation, 138–9
Procedural modeling, 60
Procedural texture, 139, 196
Procedures, of MEL statements, 291–2
  sourcing, 292
Production stage, in animator's workflow:
  3D animation, 66
  3D scene, 58
  animation, 61–2
  cameras, 64
  dynamics, 62–3
  frame rate, 61
  geometry modeling, 58–9
  lights, 63
  procedural modeling, 60
  rendering, 64–6
  shading, 64
  volumetric modeling, 59–60
Profile spline creation, 131
Profilin, 392, 394, 407
Program:
  and language, 23–5, 26–7
Protein building:
  algorithm design, 354
  algorithm encoding:
    atoms creation, 362–4
    cpk() procedure, 356–7
    error checking, 359
    main loop, 359
    MEL script composing, 354–5
    molecule check, 357–8
    PDB file opening, 358–9
    record reading, 360–2
    record type, 359
    scene hierarchy organization, 365–6
    vanDerSphere() procedure, 366–7
  level of detail, 344–5
  macromolecules visualization, 342
  MEL script preparation:
    actin, 346–8
    ATP, 346
    CPK look, 348–51
    data, 351–3
    models, naming, 353–4
  molecule rendering, 372–80
  script running, on ATP, 368–72
  visualization freeware, 345–6
  wires, ribbons, and surfaces, 342–4
Protein Data Bank (PDB), 342, 395, 396
  atoms creation, 362–4
  file format, 351–3
  file opening, 358–9
    file examination, 368
  record reading, 360–2
Proteins, 10
Pseudopodia *see* Pseudopods
Pseudopods, 140, 441, 448
  generation of, 451–3
Pull-down menus, 83–4
Python scripting interface, 72

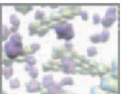
Query mode, 282
Quick Layout buttons, 94
Quotation marks, and Maya, 270, 514



Radial field:
Manipulation Tool for, 178–9
particle object connection to, 179
particles, as source, 179–81
Radiosity, *see* Global illumination
Ramp material node, 197, 198
Ramp shader, 65
Ramps, 197–8
rand() function, 308, 426, 471, 487, 496, 508, 535, 555
Random number generation, in Maya, 308–9
Range Slider, 94, 95
Raster graphics image, 246
Raytraced shadows, 232, 234
Raytracing, 234, 245, 246, 248
Reaction events:
and diffusion, 399
association, 401
dissociation, 402–3
hydrolysis and phosphate release, 402
Reaction rates, 391–2
and probabilities:
association reaction rate, 404–7
dissociation reaction rate, 408–9, 410
hydrolysis and phosphate release rates, 407–8
visualization requirements, 403–4
Reaction volume, 404, 405
Realism:
about photorealism, 252
rebuildCurve command, 504
Receive shadows, 234
Regenerative medicine, 522
Regulatory factor, 37, 38, 384, 385
rehash command, 269, 292, 303, 306, 440, 441, 570
Release notes, 74
Reload button, 296
Remove Image, 203
Render engine, 64, 244
Render farm, 66
Render file naming, 253
Render Layer, 97, 251, 252
Render Log, 257
Render menu set, 190, 191
Render nodes, 124, 191, 192
naming, 208–9
Render Settings, 213–14, 220, 244, 245, 379
camera attributes, adjusting, 225–6
image file Output, 253–4
image size, 254–5
Maya Software Render Settings, 255–6
Render View, 190, 260
images, keeping and removing, 203
IPR, previewing with, 202
Renderers, 188, 244
Rendering, 64–6, 163, 188, 244
batch rendering, 244–5, 252
common Render Settings, 253–5
Maya Software Render Settings, 255–6
software vs. hardware rendering, 257
Hardware Render Buffer, 246
Maya Hardware renderer, 246
Maya Software renderer, 245–6
Maya Vector renderer, 246–8
mental ray for Maya renderer, 248–9
advanced techniques, 249–52
playback using fcheck, 257–9
Render Settings, 244
economy of, 189–90
Render menu set, 190
rendering style, 189
shading, 191
black and white, 199
bump maps, 198, 199
displacement maps, 198, 199
Hypershade, 192
materials, 193–4
Maya Paint Effects, 201–2
preconfigured Maya shading networks, 201
ramps, 197–8
render nodes, 192
render settings, 213–14
Render View, 202–3
shaders, making and assigning, 207–9
shading engine nodes, 201
shading group node, assigning to object, 207, 208
texture nodes, 194–7
texture placement nodes, 200
textured background plane, 209–13
UV coordinates, life on surface, 199–200
reorder command, 419
Research Collaboratory for Structural Bioinformatics (RCSB), 342
resetCells expression, 545–8, 554, 570
resetSeeds expression, 497–500
Resolution gate, 225–6
rgbPP attribute, 181
creation, 182
for software render types, 183
Rigging technique, 139, 140
Rigid binding, 463
Rigid body dynamics, 164, 166–72
active, 172

animation playing, 170–1
creation, 169
memory caching, 171–2
normal direction, 168
objects, creation and positioning, 167
passive, 170
rigidSolver node, 169
Turbulence field creation, 170
RigidSolver attributes, 169
Root mean square (rms), 393
Rotate Tool Manipulator, 114–15
Rotational diffusion, 388, 400
Rule-based design, for ECM scaffolds:
rule1(), 487, 488, 489
rule2(), 487, 489–90
rule3(), 487, 490–1

Saved Layouts, 91
Scaffold model, 520
algorithm encoding, 538
cRule1(), 554–8
cRule2(), 558–60
cRule3(), 561–5
makeCells() procedure, 540–5
moveCells() expression, 548–54
resetCells() expression, 545–8
cell migration:
as emergent behavior, 521–3
nomenclature, 524–5
in scaffolds, 523
dataOutput expression, 572–3
model definition:
boundary conditions, 527
cell behavior, 525
cell-cell signaling, 526
cell geometry, 526
spatial and temporal scales, 527, 528
as substrate, for cell migration, 526, 527
model design:
cell-cell signaling, 536–8
haptotaxis, 528–36
simulation, running, 565–72
Scale Tool Manipulator, 114, 115–16
Scan-line rendering, 245
Scanning electron microscopy (SEM), 53, 54, 483, 485
Scene file:
Maya ASCII, 76
Maya Binary, 76
preparation, 266–7, 437–8, 454, 457, 565
Scene hierarchy, 81–2
organization, 365–6
Scene view, 88
setting up, 129–30
through camera, 93
see also Workspace
Script Editor, 95, 267–8, 317, 327, 355, 412, 439, 459, 470, 566, 568, 570
Script Editor History panel, 428
Script formatting, in filmmaking, 56–7
Script loading, 475–6
Script running:
Data I/O, 476
troubleshoot, 476–7
Script sourcing, 268–9
Scripting, 264–6
see also Maya Embedded Language (MEL) scripting
Scripting languages, 265
Scrubbing, 95
Search path, 269, 306, 441, 570
Seeding density, 526
Seeing, 46–8, 576
motion and animation, 48–9
wetware for, 5–6
select -clear statement, 419
Selection modes:
and masks, 119, 180
Self-assembly, of macromolecules, 384
actin geometry, 394
F-actin model, 397–9
G-actin template model, 395–7
algorithm design, 409–11
algorithm, encoding of:
reset expression, 412–19
selfAssembly expression, 420
collisions, 400
diffusion, 399–400
F-actin, structure of, 389–90
problem overview, 385
actin reactions, 390–2
model conditions, 393–4
pairing reaction and diffusion, 392–3
reaction events:
association, 401
dissociation, 402–3
hydrolysis and phosphate release, 402
reaction rates and probabilities:
association reaction rate, 404–7
dissociation reaction rate, 408–9, 410
hydrolysis and phosphate release rates, 407–8
visualization requirements, 403–4
simulation, running of:
and debugging, 440–1
scene file preparation, 437–8
script files loading, 439–40
selfAssembly expression, 420–5

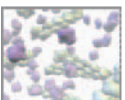


Sensor-sensor interaction, 388
Sensory impulses, 46
`setAttr`, 278, 297, 496, 499, 553
Setting Keys:
  in Attribute Editor, 141
  using channel box, 140
  using hotkey, 141
SG node, *see* Shading engine nodes
Shader Network Library, 201
Shading, 64, 188, 191, 203
  black and white, 199
  bump maps, 198, 199
  common material attributes:
    Ambient Color, 206
    bump channel, 206
    color, 205, 206
    Diffuse channel, 207
    Incandescence, 206
    Translucence, 207
    Transparency, 206
  displacement maps, 198, 199
  Hypershade, 192
  materials, 193–4
  Maya Paint Effects, 201–2
  preconfigured Maya shading networks, 201
  ramps, 197–8
  render nodes, 192
  render settings, 213–14
  Render View:
    images, keeping and removing, 203
    previewing, with IPR, 202
  shaders, making and assigning, 207
    color attributes adjustment, 209
    duplication, 209
    render nodes, naming, 208–9
  shading engine nodes, 201
  shading network to object, assigning ways, 207, 208
  surface material creation, 204–5
  texture nodes, 194–7
  texture placement nodes, 200
  textured background plane, 209–13
  UV coordinates, life on surface, 199–200
  see also Rendering
Shading engine nodes, 201
Shading menu, 89–90
Shading networks, 191–2, 201
  duplication, 209
`shadingNode` command, 366, 367
Shadows, in Maya, 232, 234, 235, 238
Shape node, 80, 175
`shareOneBrush` command, 333
Shelf button, 284–6, 369, 370
Shelves, 86–7
Shift key, 75
Short Bone Radius, 462
Short range apparent motion, 48, 49
Showtime, 15
Side Effects Software's Houdini, 578
Silicon Graphics Inc. (SGI), 15, 16, 263
Simulations, 576
  running, 570
    cells creation, 569–70
    control widget, repurposing, 566–7
    error debugging, 440–1
    model parameters variation, 570–1
    recording, 571–2
    scene file preparation, 437–8, 564–5
    scene saving, 568
    script files, loading, 439–40, 568–9
`sin()` function, 153, 154
Skeleton animation, 452
`skinCluster` command, 463, 473
Smalley, Richard, 314
Smooth bind tool, 463
Smooth binding, 463
Smooth Shade All, 90
Smooth Shade Selected Items, 90
Snap to grids, 129
Snapping, 129
Soft bodies, 160, 165
SoftImage, 16
Softimage XSI, 578
Software renderer, 163, 176, 245–6
Software rendering, 190
  vs. hardware rendering, 257
Sophia, 15, 16, 263
Space bar, 91
Spatial scale:
  and temporal scales, 323, 527
Specular Roll Off, 351
Specularity, 193, 241
Sphere:
  atoms as, 350
  deformation, 117
    control vertex, moving, 118
    hull, scaling, 118–19
    selection modes and masks, 119
  NURBS primitive modeling:
    creation, 110–11
    four panel view, 114
    Move Tool settings, 116
    renaming, 113
    rotation, 114–15
    scaling, 115–16
    scene saving, 117
    selection, 111–12
    transformation, using channel box, 116–17

translation, 113–14
smoothening, 121–2
sphere command, 367, 466, 496, 542
sphrand command, 308, 430
Spline-based modeling, 15
Spline curves, 104, 324
Spline interpolation, 143
Splines, 59, 103
components of, 104–5
Stand-alone animation expressions, 298–300
Start Frame, 176
Start Maya, 75
project, setting up, 109–10
State change probabilities, 554, 555–7
State change probability matrix, 529, 530, 547
Status Line, 86
Stochastic approach, 385
Stored programs, 30–3
Story reel, 58
Storyboard, 57, 58
Strings, 274, 275
Strokes, 247
Subdivision surface (sub-D), 109
Subscript notation, 406
Subsurface scattering, 64, 189, 248, 249, 250
Superscript notation, 406
Surface emitter, 162, 163
Surface index number, 542
Surface materials, 193
creation, 204–5
Surfaces menu set, 103, 104
swf file, 247
switch...case statement, 288
Syntax errors, 307, 371
System requirements:
monitors, 73
mouse, 73

Tangents, 148
Targa format, 253–4
Taylor, Mike, 292
Tcl, 16, 263
Tear Off, 84, 91
Tear Off Copy, 91
Terminator 2:Judgment Day, 15
Tessellation, 105, 106, 198
Text editors, for computer code writing, 302
Texture mapping, 194, 200
Texture nodes, 194–7
Texture placement nodes, 200
Texture ramp node, 197
Textured background plane:
creation and positioning, 209–10
grid texture creation, 211–12
material assignment, 210–11
material node, texture connection, 212–13
The Advanced Visualizer (TAV), 15
Third-party applications, 259
Thompson Digital Images (TDI), 14, 15, 16
Time node, 154, 155, 295
Time Range attribute, 227
Time Slider, 93–4
Time variable, 297
Time working units, 144
Timeline, 61, 94, 95, 144, 171
modeling, 487
Tissue architecture, 480
Tissue morphometry, 481
Tissues and organs, 11–12, 13
Title bar, 83
tokenizeList command, 329
Toolbox, 94
Tools Settings, 96, 116
Toon rendering technique, 241
Toon shading, 65, 247
Torque Game Engine, 578
Tracing, 267
Tracking, camera movement, 93
Traction, 448
Transferring, 525
Transform node, 80, 111, 124, 398
Transform tools, 94, 112, 113
Translational diffusion constant, 400
Translational movement, 387
Translators, 313–15
Translucence, 207, 250
Transmission electron microscopy (TEM), 53
Transparency channel, 206
Treadmilling, 385–6
flowchart, 411
ready-made file, 399
Tumble, and camera movement, 93
Tuning region, 240
Turbulence field, 162, 170, 172, 178
Turing, Alan, 33
Turing machine, 33
Tweak node, 124, 126, 134
Type casting, 274

UCSF Chimera, 17, 344, 395, 396
underworld node, 434
Unix shell scripting, 263, 282
Up axis, 92
Update View, 145
Upstream node, 79, 123, 127
Uropodia, 448
USCF Chimera, 345
Use All Lights, 90



User interface (UI), 13, 16, 25, 41, 72, 82, 102, 191, 262
  Attribute Editor, 98, 99
  Channel Box, 97–8
  Channel Control editor, 98
  Command Line, 95
  Layer Editors, 96–7
  Main Menu bar, 83–4
  menus, working with:
    Hotbox, 85–6
    Marking menus, 84–5
    option boxes, 86
    Tear Off menus, 84
  Outliner, 87–8
  playback controls, 95
  Plug-ins, 98–9
  Preferences, 95–6
  Range Slider, 94, 95
  scene viewing, through camera, 93–4
  Script Editor, 95
  shelves, 86–7
  Status Line, 86
  Time Slider, 94
  Title bar, 83
  workspace and panels:
    active panel, 88
    Lighting menu, 89, 90
    Panel menus, 88–9
    Panels menu, 89, 90–1
    Shading menu, 89–90
  XYZ coordinate system and vectors, 91
    axis indicators, 92–3
    Up axis, 92
    UV coordinates, 93
User profile, 75, 76
userPrefs.mel file, 95
UV coordinates, 93, 199–200
UV mapping, *see* Texture mapping
UV Texture Editor (UTE), 199, 200

Values, in Maya, 270
Van der Waals (vdW) radius, 344, 345, 350, 353, 356
Van der Waals collision, 386
Van Leeuwenhoek, Anton, 576
vanDerSphere(), 363, 364, 369
  procedure, 366–7
Variables:
  arrays, 275–6
  data conversion, 273–4
  declaration and assigning, 271–2
  dynamic typing, 273
  global variables, 276–7
  initialization, 499
  matrices, 276
  naming, 271
  strings, 274, 275
  type casting, 274
  types, 272
  vectors, 274
Vector array, 276
Vector graphics image, 246–7
Vector renderer plug-in, 248
Vectors, 274
Velocity curve, 62
VFX studios, 15
View Axis, 92
Vintage mainframes, 580
Virtual computer, 27, 28, 29, 264
Virtual memory, 139
Visible Human Project, 60
Vision, anatomy of, 47
Visual exploration, 4
Visual Molecular Dynamics (VMD), 17, 578, 579
Visual perception, 5
Visualization:
  algorithm, planning, 324
  challenge, 392–4
  of macromolecules, 342
  model requirements, 403
  in science, 6–8
Visualization freeware, 345–6
Visualization Toolkit, 578
Volumetric materials, 193
Volumetric modeling, 59–60
Volumetric tools, 60
Von Neumann, John, 30, 31, 32, 34, 39
Von Neumann machine, 25, 31, 41, 43, 579, 580
Voxels, 59, 60

Wavefront RLA, 253
Wavefront Technologies, 14, 263
Wetware, 5–6
while loop, 290
Wilkes program, 24
Wireframe, 89, 90
Workspace, 88–90
  active panel, 88
  Lighting menu, 89, 90
  Panel menus, 88–9
  Panels menu, 89, 90–1
  Shading menu, 89–90
World coordinates, 92
World origin, 91, 92
Write (w) mode, 316

X-ray shading, 167
XYZ coordinate system and vectors, 91
  axis indicators, 92–3
  Up axis, 92
  UV coordinates, 93