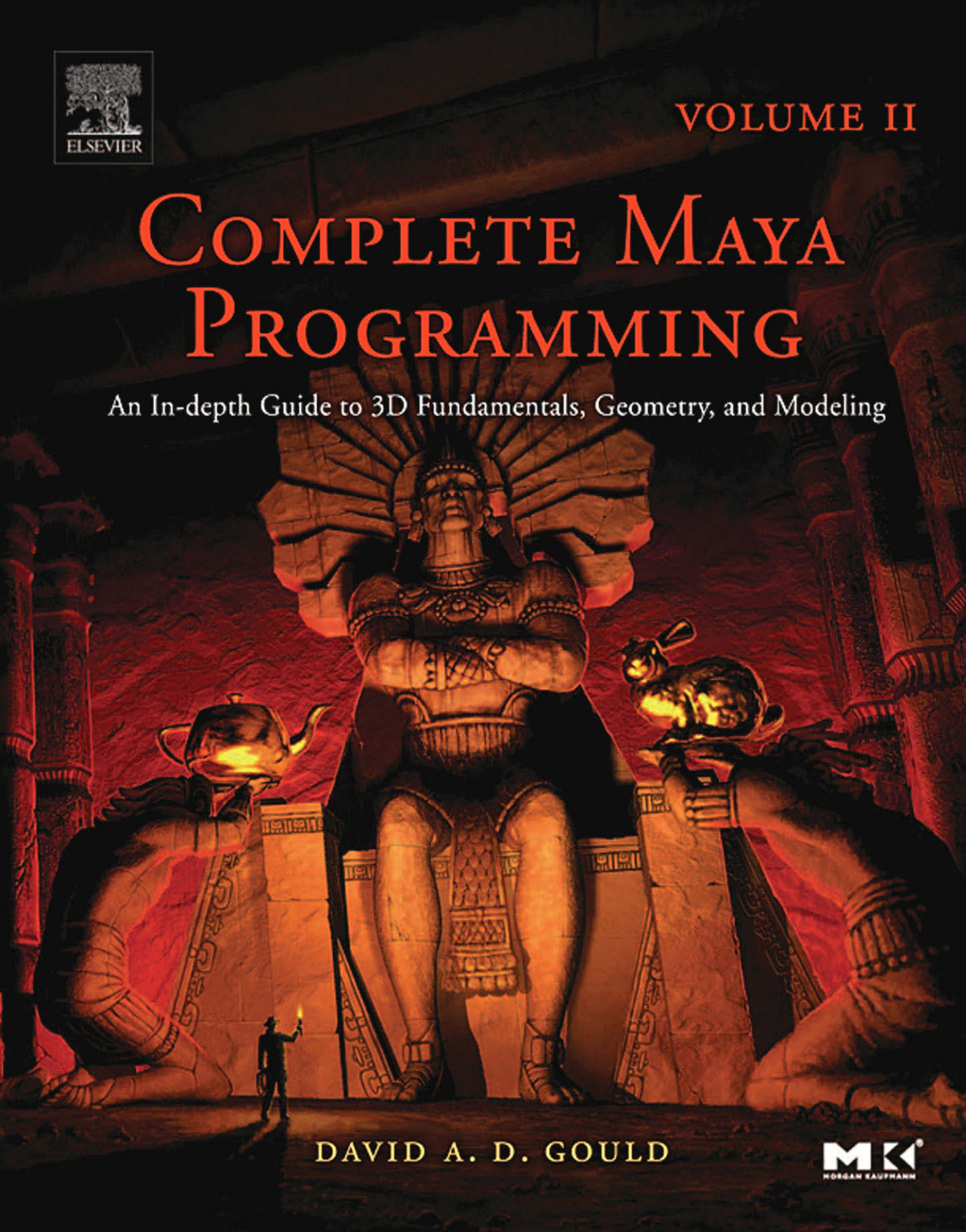
ELSEVIER
VOLUME II
COMPLETE MAYA PROGRAMMING
An In-depth Guide to 3D Fundamentals, Geometry, and Modeling
DAVID A. D. GOULD
MK
MORGAN KAUFMANN

*A great follow-up to Volume I! Volume II is an in-depth guide to the mathematical and geometric concepts indispensable to advanced Maya programmers.*

—Larry Gritz, Exluna/NVIDIA

From Volume I of *Complete Maya Programming*:

*David's book is an excellent learning tool and reference for novice and veteran Maya developers alike. Maya developers can become more productive with MEL and the Maya API by applying what they learn from this book.*

—Tracy Narine, Maya API Technical Lead, Alias

*David Gould is an expert at using, programming, and teaching Maya, and it shows. People who need to program Maya will find this book essential. Even Maya users who don't intend to do extensive programming should read this book for a better understanding of what's going on under the hood. Compact yet thorough, it covers both MEL and the C++ API, and is written to be informative for both novice and expert programmers. Highly recommended!*

—Larry Gritz, Exluna/NVIDIA, co-author of *Advanced RenderMan*

*This book should be required reading for all Maya programmers, novice and expert alike. For the novice, it provides a thorough and wonderfully well thought-out hands-on tutorial and introduction to Maya. The book's greatest contribution, however, is that in it David shares his deep understanding of Maya's fundamental concepts and architecture, so that even the expert can learn to more effectively exploit Maya's rich and powerful programming interfaces.*

—Philip J. Schneider, Industrial Light & Magic, co-author of *Geometric Tools for Computer Graphics*

*Having provided a technical review of David Gould's Complete Maya Programming, I must say that this book is the definitive text for scripting and plug-in development for Maya. Never before has there been such a concise and clearly written guide to programming for Maya. Any user smart enough to pick up this book would be better off for it.*

—Chris Rock, technical director at "a Large Animation Studio in Northern California"

*If you ever wanted to open the Maya toolbox, this is your guide. With clear step-by-step instructions, you will soon be able to customize and improve the application, as well as create your own extensions, either through the MEL scripting language or the full C++ API.*

—Christophe Hery, Industrial Light & Magic

## The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling

*Complete Maya Programming Volume II: An In-depth Guide to 3D Fundamentals, Geometry, and Modeling*
David A. D. Gould

*High Dynamic Range Imaging: Data Acquisition, Manipulation, and Display*
Erik Reinhard, Greg Ward, Sumanta Pattanaik, and Paul Debevec

*MEL Scripting for Maya Animators, Second Edition*
Mark R. Wilkins and Chris Kazmier

*Advanced Graphics Programming Using OpenGL*
Tom McReynolds and David Blythe

*Digital Geometry: Geometric Methods for Digital Picture Analysis*
Reinhard Klette and Azriel Rosenfeld

*Digital Video and HDTV: Algorithms and Interfaces*
Charles Poynton

*Real-Time Shader Programming*
Ron Fosner

*Complete Maya Programming: An Extensive Guide to MEL and the C++ API*
David A. D. Gould

*Texturing & Modeling: A Procedural Approach, Third Edition*
David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley

*Geometric Tools for Computer Graphics*
Philip Schneider and David H. Eberly

*Understanding Virtual Reality: Interface, Application, and Design*
William B. Sherman and Alan R. Craig

*Jim Blinn's Corner: Notation, Notation, Notation*
Jim Blinn

*Level of Detail for 3D Graphics*
David Luebke, Martin Reddy, Jonathan D. Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner

*Pyramid Algorithms: A Dynamic Programming Approach to Curves and Surfaces for Geometric Modeling*
Ron Goldman

*Non-Photorealistic Computer Graphics: Modeling, Rendering, and Animation*
Thomas Strothotte and Stefan Schlechtweg

*Curves and Surfaces for CAGD: A Practical Guide, Fifth Edition*
Gerald Farin

*Subdivision Methods for Geometric Design: A Constructive Approach*
Joe Warren and Henrik Weimer

*Computer Animation: Algorithms and Techniques*
Rick Parent

*The Computer Animator's Technical Handbook*
Lynn Pocock and Judson Rosebush

*Advanced RenderMan: Creating CGI for Motion Pictures*
Anthony A. Apodaca and Larry Gritz

*Curves and Surfaces in Geometric Modeling: Theory and Algorithms*
Jean Gallier

*Andrew Glassner's Notebook: Recreational Computer Graphics*
Andrew S. Glassner

*Warping and Morphing of Graphical Objects*
Jonas Gomes, Lucia Darsa, Bruno Costa, and Luiz Velho

*Jim Blinn's Corner: Dirty Pixels*
Jim Blinn

*Rendering with Radiance: The Art and Science of Lighting Visualization*
Greg Ward Larson and Rob Shakespeare

*Introduction to Implicit Surfaces*
Edited by Jules Bloomenthal

*Jim Blinn's Corner: A Trip Down the Graphics Pipeline*
Jim Blinn

*Interactive Curves and Surfaces: A Multimedia Tutorial on CAGD*
Alyn Rockwood and Peter Chambers

*Wavelets for Computer Graphics: Theory and Applications*
Eric J. Stollnitz, Tony D. DeRose, and David H. Salesin

*Principles of Digital Image Synthesis*
Andrew S. Glassner

*Radiosity & Global Illumination*
François X. Sillion and Claude Puech

*User Interface Management Systems: Models and Algorithms*
Dan R. Olsen, Jr.

*Making Them Move: Mechanics, Control, and Animation of Articulated Figures*
Edited by Norman I. Badler, Brian A. Barsky, and David Zeltzer

*Geometric and Solid Modeling: An Introduction*
Christoph M. Hoffmann

*An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*
Richard H. Bartels, John C. Beatty, and Brian A. Barsky

# COMPLETE MAYA PROGRAMMING

## VOLUME II

An In-depth Guide to 3D Fundamentals, Geometry, and Modeling

David A. D. Gould

ELSEVIER

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

Morgan Kaufmann Publishers is an imprint of Elsevier

MORGAN KAUFMANN PUBLISHERS

| | |
|---|---|
| Senior Editor | Tim Cox |
| Publishing Services Manager | Simon G. Crump |
| Senior Project Manager | Angela G. Dooley |
| Assistant Editor | Richard Camp |
| Editorial Assistant | Jessica Evans |
| Cover Design Direction | Cate Rickard Barr |
| Cover Illustration | Sean Platter, Studio Splatter |
| Text Design Direction | Julio Esperas |
| Composition | Integra Software Services Pvt. Ltd., Pondicherry, India |
| Technical Illustration | Dartmouth Publishing Inc. |
| Copyeditor | Daril Bentley |
| Proofreader | Phyllis Coyne et al. |
| Indexer | Northwind Editorial |
| Interior Printer | Maple-Vail Manufacturing Group |
| Cover Printer | Phoenix Color Corp. |

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

*This book is printed on acid-free paper.*



Library of Congress: Application submitted

ISBN: 0-12-088482-8
ISBN: 978-0-12-088482-8

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

Printed in the United States of America
05 06 07 08 09 5 4 3 2 1

To Agnes, my foundation.

# About the Author

With over thirteen years of experience in the computer graphics industry, David Gould has pursued the dual paths of programmer and artist. This rare ability to combine both the technical and artistic has won him many awards and credits. He has played a key role in the development of an eclectic mix of technology, including an award-winning laser rendering system for Pangolin. He developed software for controlling the Kuper motion-control rig, as well as the Monkey stop-motion puppet. He personally developed Illustrate, the market leading toon and technical illustration renderer. This renderer is used by NASA, British Aerospace, Walt Disney Imagineering, and Sony Pictures Entertainment, among others.

David's career has spanned a wide variety of companies and continents. In Paris, he supervised the production of 3D stereoscopic scientific films, including the award winning film *Inside the Cell*. In London he developed a patented facial animation system. Further broadening his experiences, he worked in New York in the post-production industry where he contributed to many high-profile commercials.

While at Walt Disney Feature Animation, Los Angeles, David developed cutting-edge animation and modeling technology that was used in the production of their animated feature films. He diversified further by joining Exluna, Berkeley, the software company founded by former Pixar rendering researchers, including Larry Gritz. While there, he played an active role in the design and development of Entropy, the Renderman-compatible renderer, as well as other products. David continued his rendering development efforts while at NVIDIA, in Santa Clara, California, by aiding in the design of their future 3D graphics chips.

David has since joined the academy awarding winning studio WETA Digital in New Zealand. Having completed work on *The Lord of the Rings* trilogy he is currently working on *King Kong*. His varied roles in production include research and development, shader writing, lighting, and effects.

# Contents

# Preface

Given the depth and breadth of Maya's programming functionality, it became quickly clear that a single book couldn't possibly cover it all. The first volume focused on giving the reader a solid understanding of how Maya works and on its two programming interfaces: MEL and the C++ application programming interface (API). This book extends on that work, while paying particular attention to the areas of geometry and modeling. Clearly, in order to have a deeper understanding of these areas it is important to first understand the fundamentals of computer graphics, and in particular the mathematical foundations on which they are built. This book, therefore, explains the fundamental building blocks of computer graphics so that a complete understanding of geometry and modeling is possible.

Although the mathematics and principles of computer graphics are explained in other books, I felt it necessary to place these in the context of Maya programming. So, rather than explain the theory alone, sample source code and scripts are provided so that the reader can see how the mathematics and principles can be directly applied and implemented. Many of the examples can be used directly in your own implementations.

Because the first book was focused on teaching the fundamentals of MEL and the C++ API, these two areas were covered separately. This book takes a more problem-solving approach. The utility of a particular mathematical concept is explained together with both the MEL and C++ source code used to implement the concept. The key is to understand the concept; the syntax then follows. By building up a wider understanding of computer graphics concepts, you will have a larger toolbox of solutions from which to draw when tackling your own problems.

This book contains a great deal of knowledge I have accumulated over the years. Much of it is taken from experience, books, courses, and notes. All of this information isn't of much use if you can't readily access it. As such, another important goal of this book is to provide the reader with a pertinent reference resource. By dividing the book by concept, rather than by programming language, it is easy to refer to particular sections as needed. The subject index has been extensively expanded and is more useful for finding when and where a particular function or command can be used.

Although every attempt was made to present each area with an equal emphasis on MEL and the C++ API, it will soon become obvious to the reader that the C++ API is clearly more powerful and versatile when it comes to handling larger and more complex problems. With the ever-increasing need for more detailed and complex models, it becomes even more important that your solution work quickly and efficiently. In addition to the speed gains it makes possible, the C++ API offers a great many convenience classes. For example, because MEL doesn't have direct support for quaternions, you would need to implement them yourself. The C++ API has the **MQuaternion** class, which provides a complete implementation of quaternions. This class can be used directly in your plug-ins. You can also rest assured that the class has been extensively tested so that it is guaranteed to be both robust and stable. Integrating your solutions into Maya in a clean and seamless manner is often only possible through the C++ API. Your users will be more appreciative of a solution that resembles the standard Maya implementation than one that is compromised simply because of MEL's limitations. Admittedly, the learning curve for the C++ language is steeper than that for MEL, but in the long run the additional functionality provided by knowing C++ will pay off. You will have a greater scope for handling more diverse and complex problems, some of which may be difficult, if not impossible, to implement in MEL. Ideally, a solid knowledge of both programming interfaces will give you the maximum freedom of choice.

## ACKNOWLEDGEMENTS

The process of writing a book can be likened to a marathon and like any successful athelete the role of the support staff is critical to their success. I would like to make a particular acknowledgement to my editor, Tim Cox, and his very capable assistants Richard Camp and Jessie Evans. They pressed ahead, undaunted, as the book continued to grow ever larger and more complex.

If each page looks as outstanding as it does it is due to the professionalism and hard work of Angela Dooley and her great team of designers and copy editors.

To my reviewers I'd like to thank them for their critical eye and abundant feedback. Their ideas and opinions assisted me greatly in defining the book's core goals and focus. My dream team of reviewers included Scott Firestone, Bryan Ewert, Christophe Hery, Michael Lucas, André Mazzone, Philip Schneider, and Andre Weissflog.

This Page Intentionally Left Blank

CHAPTER 1

# Introduction

This book endeavors to build upon your existing experience in Maya programming. As such, this book assumes that you are already familiar with basic MEL and/or C++ API programming. If you have never written MEL scripts or C++ plug-ins, you are highly encouraged to read the first volume. It covers all of the basics of Maya programming, as well as how Maya works internally. This knowledge will be critical when developing more advanced scripts and plug-ins.

All too often your work in Maya will involve problem solving. Although it is often easy to formulate a solution in general abstract terms, the task of actually implementing the solution can be daunting. If you read the first volume, you have a strong understanding of how Maya works, as well as of the MEL and C++ API programming interfaces. Thus, the actual task of writing a script or plug-in shouldn't be too difficult. The next step will be to apply your knowledge of computer graphics principles and mathematics to implement the solution.

This often proves to be the greatest hurdle. The most common reason for not being able to implement a solution is due to a lack of understanding of computer graphics principles. Without a solid grasp of basic computer graphics concepts, all problem solving will become more difficult. Computer graphics is based on mathematics, and many people are quite reluctant to learn mathematics. A common reason for this is that the mathematics is presented in abstract and theoretical terms with little application to solving real-world problems. This book presents the most important fundamental mathematical concepts without diverging into esoteric mathematical areas that have little practical use. For instance, the *dot product* is covered in detail. This mathematical operation is used extensively in computer graphics for calculating such things as angles, rotations, sidedness, lengths, areas, and the amount of light reflected from a surface. All of this is possible from an operation

that involves little more than a few multiplications and additions. Independently of Maya programming, a solid grasp of these computer graphics concepts will hold you in good stead for all of your future work. As your knowledge increases, you will become more adept at combining and using these mathematical building blocks to create more robust and efficient solutions.

The explanation of each mathematical concept is accompanied by ample source code and scripts that demonstrate how to implement the concept. The source code and scripts can be used as a starting point for your own solutions. The entire spectrum of computer graphics concepts through to geometry and modeling is covered.

The most fundamental building blocks of computer graphics are points and vectors. Many problems can be solved using the simple point and vector operations, and thus understanding them will provide a strong foundation for all further chapters. Rotations merit their own chapter in that they can often be the source of much confusion. There are many ways to represent rotations and orientations, and thus it is important to understand their advantages and disadvantages in order to best apply them to your work. Integral to computer graphics is the process of transforming (scaling, shearing, rotating, translating, and projecting) objects. Transformations are most efficiently implemented using matrices, covered in this book in detail. Transformations provide an important level of abstraction for building hierarchies of objects. Being able to retrieve and transform points at any level in a hierarchy are particularly useful skills in many computer graphics applications.

Progressing from the basic building blocks, the next topic covered is geometry. Geometry uses points and vectors to represent more complex shapes such as curves and surfaces. All of Maya's supported geometry types are covered in their own respective chapters. Each chapter covers the tasks of displaying, editing, and creating each geometry type. A detailed explanation of the components that make up each type is also given. The most basic, yet most pervasive, type of geometry — polygonal meshes — is covered first. NURBS curves and surfaces are subsequently explained in detail.

Finally, the increasingly popular geometry type, subdivision surfaces, is covered. Each different type of geometry has its strengths and weaknesses. Some are better suited for games development, whereas others are more appropriate for industrial design. The reader will gain a greater understanding of each geometry type's advantages and disadvantages, so that an informed decision can be made as to which one is best to use. The process of writing geometry importers and exporters is greatly simplified once you have a greater understanding of Maya's various geometry types. Custom modeling tools can also be created that are adapted to a given geometry

type. Developing your own tools will provide you with a far greater level of control and functionality than those provided with Maya.

## 1.1 EXAMPLE FILES

Note that all files used in this book are available at:

www.davidgould.com

Information available at the site includes the following.

- MEL scripts, C++ source code, and makefiles for all examples in the book
- Additional example MEL scripts
- Additional example C++ source code
- Errata for this book
- Continually updated glossary
- Updated list of other relevant web sites and online resources

### 1.1.1 COMPILING EXAMPLE PLUG-INS

New versions of both Maya and C++ compilers are being constantly released. Rather than provide potentially outdated and obsolete instructions in this book, the complete set of instructions for downloading and compiling the companion files for this book are available at:

www.davidgould.com

Here you will find all C++ source code and makefiles necessary to compile the plug-ins on your own computer. There are also instructions for creating your own plug-in makefiles from scratch.

### 1.1.2 SOURCING EXAMPLE MEL SCRIPTS

To source any of the example MEL scripts, Maya must be able to locate them. It is possible to include the complete path to the `source` command, but if multiple MEL scripts are being sourced it is easier to set up the `MAYA_SCRIPT_PATH` environment

variable correctly. This variable simply needs to be updated, as follows, to include the directory where the example MEL scripts are located.

1. Open the **Script Editor**.
2. Execute the following to initialize the `$exampleScripts` string to the path of the directory containing the example MEL scripts.

```
string $exampleScripts = <example_mel_scripts_directory>;
```

For example:

```
string $exampleScripts = "C:/DavidGould/MEL Scripts";
```

When specifying a path in Windows with backslashes, be sure to use two backslashes. A single backslash will be interpreted as an escape sequence. The same path with backslashes would therefore be written as follows.

```
string $exampleScripts = "C:\\DavidGould\\MEL Scripts";
```

Maya will automatically convert all directory paths with backslashes to forward slashes.

3. Execute the following:

```
string $newScriptPath=$exampleScripts + ";" +`getenv "MAYA_SCRIPT_PATH"`;
putenv "MAYA_SCRIPT_PATH" $newScriptPath;
```

The first line initializes the `$newScriptPath` variable to the example MEL scripts path and then appends the current setting for the `MAYA_SCRIPT_PATH` variable. The second line uses the `putenv` command to set the `MAYA_SCRIPT_PATH` variable to the path.

With the `MAYA_SCRIPT_PATH` environment variable now updated, sourcing any MEL script can be done the same way. For example, to source the **foobar.mel** script the following code would be executed:

```
source foobar.mel;
```

The previous steps need to be completed for each new Maya session. Thus, if Maya is restarted the previous steps should be performed.

## 1.2 EXECUTING MEL CODE IN THE SCRIPT EDITOR

There are a lot of snippets of MEL code throughout this book. Many readers will want to type this MEL code into the **Script Editor** and then execute it. This will work fine in most cases. There will often be a problem when you execute different blocks of MEL code that use the same variable name but assume a different type. This problem is demonstrated in Maya as follows.

1. Open the **Script Editor**.
2. Execute the following.

```
$myVar = 1.0
```

   The result is displayed.

```
// Result: 1 //
```

   This creates the `$myVar` and assigns it an initial value.

3. To see what would happen if there were another piece of MEL code that used the same variable but as a different type, execute the following.

```
$myVar = "hi"
```

   This produces an error.

```
// Warning: Converting string "hi" to a float value of 0. //
// Result: 0 //
```

4. Execute the following.

```
whatIs "$myVar"
```

The variable's data type is printed out.

```
// Result: float variable //
```

The problem is that although you are executing another piece of MEL code the `$myVar` variable still exists. The attempt to assign a string to it failed because the variable is already defined as a float. Once the data type (string, float, int, and so on) is defined for a variable it can't be changed.

The underlying problem is that all variables defined in the **Script Editor** are automatically made global variables, even if you don't explicitly make them. Thus, executing the statement

```
$myVar = 1.0
```

in a script would make it a local variable. This same statement executed in the **Script Editor** is the equivalent of writing

```
global $myVar = 1.0
```

The variable is implicitly made global. Once a variable is global there is no way to delete it. The only way is to restart Maya and thereby remove all global variables and start afresh. Note that this behavior also extends to procedures. Any procedure defined in the **Script Editor** will automatically become global.

What is needed is a way of defining the variable to be local. Unfortunately there is no explicit keyword (an opposite to the `global` keyword) that makes a variable local. This is, however, a way of implicitly making a variable local. By using code blocks, a variable is implicitly made local. At the end of the code block the variable is automatically deleted. This is precisely what is needed to ensure that running several sections of MEL code doesn't define the same global variable. This also prevents a "contamination" of the global name space of variables with variables you intended only for learning and testing.

5. Restart Maya by closing it and then opening it again.
6. Execute the following in the **Script Editor**.

```
{
$myVar = 1.0;
print $myVar;
}
```

The value of `$myVar` is printed out.

```
1
```

Because the definition of the variable was enclosed in braces, this created a separate code block. All variables defined within the block are automatically local. When the closing brace (}) is reached, all variables defined within the block are deleted. This ensures that the `$myVar` variable no longer exists after the code is executed and prevents it being added to the list of global variables.

7. Execute the following.

```
{
$myVar = "hi";
print $myVar;
}
```

The value of `$myVar` is printed out.

```
hi
```

There was no error this time when `$myVar` was defined because it is local to the block and is deleted when the block is finished.

Thus, the general rule is that if you ever intend on executing MEL code in the **Script Editor** simply enclose it in braces to ensure that it runs as a separate block. There may be times when you want to determine if a given variable is global. The following MEL procedure is designed to return `true` if the supplied variable is global, and `false` otherwise.

```
global proc int isGlobal( string $var )
{
  string $globals[] = 'env';
  for( $glob in $globals )
  {
   if( $glob == $var )
     return true;
  }
  return false;
}
```

This procedure can then be used to test if `$myVar` is a global variable.

```
isGlobal( "$myVar" )
```

The result is 0 (false). Note that the variable name is enclosed in quotation marks ("). This ensures that the variable name is passed to the procedure and not its value. Also note that this procedure is a global procedure and thus can be called from anywhere within Maya (script, **Script Editor**, and so on) once it is defined.

CHAPTER 2

# Points

Points and vectors provide the fundamental building blocks upon which geometry is based. Before covering the specifics of Maya's point and vector types it is important to understand the mathematical basis for points and vectors.

## 2.1 DIMENSIONS

The dimension of a point is the number of coordinates it has. Maya doesn't provide an explicit 2D point or vector, although a 3D point or vector can be used for the same purpose. Maya provides 3D points in MEL and 4D points (homogenous points) in the C++ API.

## 2.2 CARTESIAN COORDINATES

A 3D Cartesian point is represented as follows.

$$\mathbf{p} = (x, y, z)$$

Cartesian coordinates are based on distances from the origin *(0,0,0)*. Each coordinate is a distance measured along an axis, starting at the origin. Because each of the axes is perpendicular to the others, the combination of coordinates defines a precise position in space.

For 3D points, the three coordinates define the distance along the standard X *(1,0,0)*, Y *(0,1,0)*, and Z *(0,0,1)* axes. Figure 2.1 shows a point with Cartesian coordinates (3, 4, 1). This point is located by starting at the origin and moving three

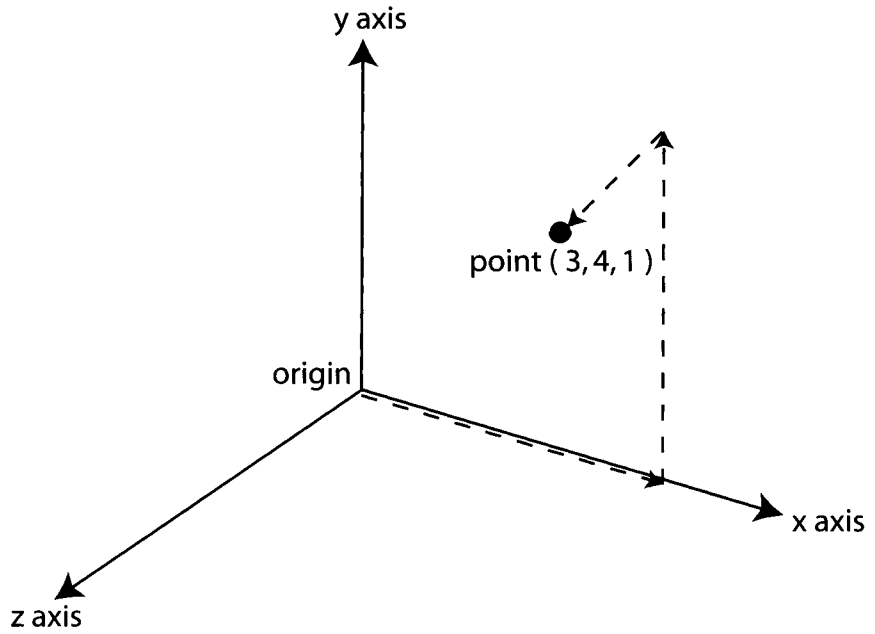


**FIGURE 2.1** Cartesian coordinates.

units along the X axis. Next, move four units along the Y axis, followed by one unit along the Z axis. This is the final position of the point.

## 2.3 HOMOGENEOUS COORDINATES

A point can also be represented in homogeneous coordinates. Such a point has four dimensions and is represented as follows.

$$\mathbf{p} = (x, y, z, w)$$

The additional coordinate, **w**, can be thought of as providing a scaling of the point. Keeping the **x**, **y**, and **z** components the same and simply varying the **w** component will produce a series of points along a line. The line runs through the origin and the point (**x**, **y**, **z**). Homogeneous coordinates are particularly useful for calculating projections. A projection is where a 3D point is projected onto a 2D point. A good example of this is the perspective projection, wherein a point in the scene is projected onto the image plane. The result is a 2D pixel in the image plane.

The addition of another coordinate is also very useful for applying more generalized transformations to points. This is covered in the transformation section.

## 2.4 POLAR AND SPHERICAL COORDINATES

A 2D point can be represented using polar coordinates, as follows.

$$\mathbf{p = (r, \theta)}$$

The **r** coordinate is a distance from the origin. The θ (Greek theta symbol) is the angle (in radians) rotated around from the X axis. (See **Section 4.1** for further details on angles and angle units.) The direction of rotation is counterclockwise. Figure 2.2 shows a point at polar coordinates (1.5, 0.78). The angle 0.78 is 45 degrees in radians.

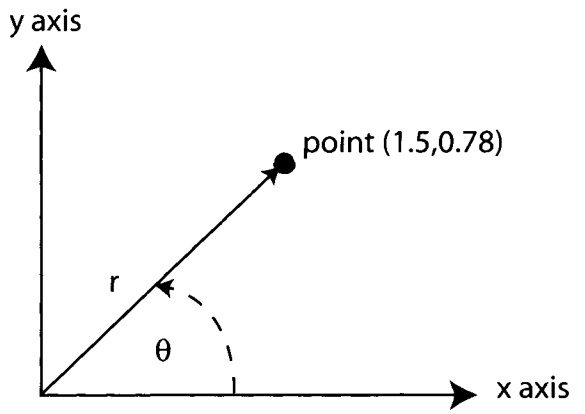


**FIGURE 2.2** Polar coordinates.

To represent a 3D point in a similar manner, an additional angle — ϕ (Greek phi symbol) — is needed.

$$\mathbf{p = (r, \phi, \theta)}$$

The point is now located on a sphere with radius **r**. The θ angle specifies the rotation about the *Z* axis from the *X* axis. The ϕ angle is rotation from the *Z* axis. Both rotations are counterclockwise. Figure 2.3 shows a point with spherical coordinates (1, 0.78, 0.78 ). Note that the vertical axis is the *Z* axis.

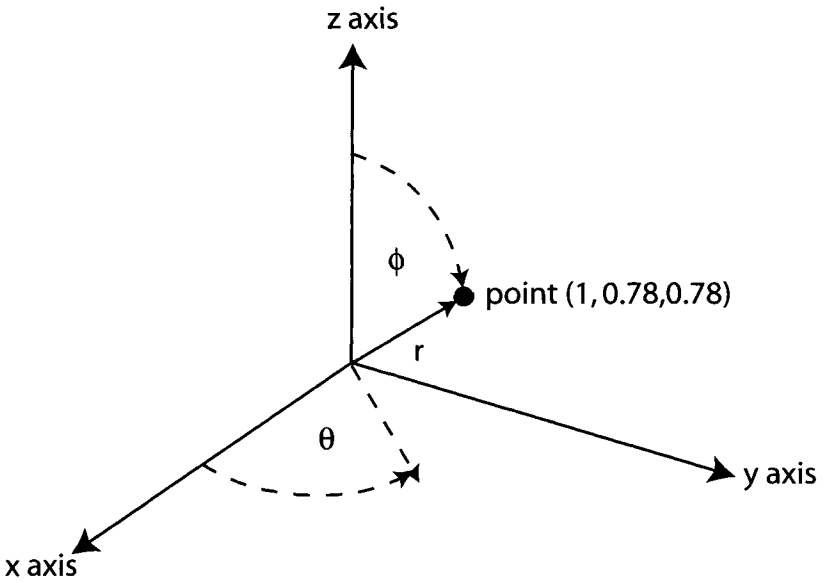


**FIGURE 2.3** Spherical coordinates.

The final position is constructed as follows. Move from the origin along the $Z$ axis by a distance of r. Rotate the position around the $Y$ axis by an angle of $\phi$. The point is on the $X$-$Z$ plane. Rotate the point around the $Z$ axis by the angle $\theta$. The point is now in its final position.

The $\theta$ angle has a range of 0 to $\pi$ radians (0 to 180 degrees). The $\phi$ angle has a range of 0 to $\pi/2$ radians (0 to 90 degrees).

## 2.5 CONVERSIONS

This section defines how to convert points between the various representations.

### 2.5.1 CARTESIAN TO HOMOGENEOUS

Any point with $n$ dimensions can be converted to a point with $n + 1$ dimensions by multiplying each coordinate by a scalar. The $n + 1th$ coordinate is then set to this scalar. Thus, to convert from a 3D Cartesian point

$$\mathbf{p} = (x, y, z)$$

to a 4D homogeneous point

$$\mathbf{p' = (x', y', z', w)}$$

the original coordinates are multiplied by a scalar. The simplest scalar is obviously 1.

$$\begin{aligned}\mathbf{p'} &= \mathbf{(1 \cdot x, 1 \cdot y, 1 \cdot z, 1)} \\ &= \mathbf{(x', y', z', 1)}\end{aligned}$$

### 2.5.2 HOMOGENEOUS TO CARTESIAN

To convert from a homogeneous point back to a Cartesian point, the opposite operation is performed. All coordinates are divided by the last coordinate, w.

$$\begin{aligned}\mathbf{p} &= \mathbf{(x' / w, y' / w, z' / w, w / w)} \\ &= \mathbf{(x, y, z, 1)} \\ &= \mathbf{(x, y, z)}\end{aligned}$$

When implementing this formula it is important to check for $w = 0$. This will cause a division-by-zero error. If $w = 0$ the vector can be immediately set to the zero vector.

### 2.5.3 CARTESIAN TO POLAR

Because a polar coordinate only has two dimensions, the z coordinate is ignored. To convert the Cartesian coordinates

$$\mathbf{p = (x, y, 0)}$$

to polar coordinates, the r coordinate is calculated as the distance from the point to the origin. The angle is the arc tangent of the y and x values.

$$\begin{aligned}\mathbf{p'} &= \mathbf{(r, \theta)} \\ &= \mathbf{(\sqrt{x^2 + y^2}, \tan^{-1}(y, x))}\end{aligned}$$

### 2.5.4 POLAR TO CARTESIAN

The polar coordinate

$$\mathbf{p = (r, \theta)}$$

is converted to Cartesian coordinates as follows.

$$\begin{aligned} p' &= (x, y, z) \\ &= (r\cos(\theta), r\sin(\theta), 0) \end{aligned}$$

### 2.5.5 CARTESIAN TO SPHERICAL

To convert a Cartesian point

$$p = (x, y, z)$$

to spherical coordinates, use the following.

$$p' = (r, \phi, \theta)$$

where

$$\begin{aligned} r &= \sqrt{x^2 + y^2 + z^2} \\ \phi &= \tan^{-1}(\sqrt{x^2 + y^2}, z) \\ \theta &= \tan^{-1}(y, x) \end{aligned}$$

### 2.5.6 SPHERICAL TO CARTESIAN

To convert the spherical coordinates

$$p = (r, \phi, \theta)$$

to Cartesian coordinates, use the following.

$$p' = (x, y, z)$$

where

$$\begin{aligned} x &= r\sin(\phi)\cos(\theta) \\ y &= r\sin(\phi)\sin(\theta) \\ z &= r\cos(\phi) \end{aligned}$$

## 2.6 MEL

MEL's use of the term *vector* is more closely related to the computer science definition of a vector: a one-dimensional array of scalars. As such, there are very few restrictions on what operations can be performed on a vector.

A vector's elements can be accessed through its x, y, and z components. Vectors can be added, subtracted, and multiplied by other vectors, resulting in another vector. These operations are simply performed in a component-wise fashion.

Because a vector has just three components, it can only be used to represent Cartesian coordinates. The lack of a fourth component prevents it from being used in homogenous calculations. A point is defined as follows.

```
vector $p;                          // Initialized as (0,0,0)
vector $p = 3;                      // Initialized as (3,3,3)
vector $p = <<4.5, 3.8, 3.2>>;      // Initialized as (4.5, 3.8, 3.2)
```

Although the vector data type is convenient for performing vector operations, many of Maya's MEL commands and procedures don't support vector operations. For instance, a quite common task is getting the position of a transform. The command

```
getAttr transform1.translate;
```

will return an array of three floating-point numbers:

```
// Result: 0 0 0 //
```

From Maya 6.0 onward, it is valid to explicitly assign this array of three scalars to a vector as follows.

```
vector $t = `getAttr transform1.translate`;
```

In earlier versions of Maya, this would have caused an error. In all versions it isn't possible to directly assign a vector to an attribute.

```
vector $t = << 1, 2, 3 >>;
setAttr transform1.translate $t; // Causes an error
```

Instead, the vector must be assigned in a component-wise fashion.

```
setAttr transform1.translate ($t.x) ($t.y) ($t.z); // OK
```

## 2.7 C++ API

The C++ class for points is the **MPoint** class. The **MPoint** class is a homogeneous point with four coordinates: `x`, `y`, `z`, `w`. Each coordinate is stored as a `double`. There also exists a `float` variation of this class, **MFloatPoint**. The default constructor initializes the coordinates to the following.

```
MPoint pt; // x=y=z=0, w=1
```

The point can be converted from a Cartesian point to a homogeneous point via the `homogenize` function.

```
MPoint pt;
pt.homogenize(); // pt = (w*x, w*y, w*z, w)
```

This function simply multiplies each component by the `w` component. Note that if `w` is 0 then a zero vector (*0,0,0,0*) will result. To convert the point from a homogeneous point to a Cartesian point the `cartesianize` function is used.

```
MPoint pt;
pt.cartesianize(); // pt = (x/w, y/w, z/w, 1)
```

This function is the inverse of the `homogenize` function and thus divides all components by `w`. There also exists a final conversion function, `rationalize`, that works similarly to `cartesianize`, but instead of setting `w` to 1 at the end it leaves it.

```
MPoint pt;
pt.rationalize(); // pt = (x/w, y/w, z/w, w)
```

It is important to note that Maya doesn't explicitly store which form (Cartesian, homogeneous, rational) the point is in. It is up to the developer to ensure that only those functions that are valid for a given form are used. There is nothing to prevent the `rationalize` function from being called twice, which will clearly

result in an incorrect point. For convenience the predefined class instance, `origin`, exists.

```
MPoint::origin // point at (0,0,0)
```

These can be used like regular class instances, as in the following example.

```
MPoint p0;
if( p0 == MPoint::origin )
 MGlobal::displayInfo( "point is at the origin" );
```

## 2.8 LOCATORS

Maya doesn't have a geometry shape for a single point. However, locators can be used for this purpose. Locators are visual guides that are drawn in viewports but are not rendered. To create a locator at a given position, use the following.

```
spaceLocator -p 1 3 6;
```

Because a locator has its own transform node it can be scaled, rotated, and translated like any other geometry. The position of a locator can be retrieved in object and world space as follows.

```
xform -query -objectSpace -translation;
xform -query -worldSpace -translation;
```

This Page Intentionally Left Blank

CHAPTER 3

# Vectors

A vector has both a direction and a *magnitude*. The magnitude of a vector is simply its length. Vectors are often used to define the difference between points, which is the *displacement* from the first point to the second. A series of vectors (combined with an origin) can also define a coordinate frame. This can then be used for defining a custom space in which to define other points and vectors.

In Maya, all vectors are 3D. The three components are named x, y, and z. It is important to understand that while many books show vectors located somewhere in space, vectors don't have a location. A vector is a relative movement, an offset. In terms of position, a vector has no meaning without a point. The vector is added to the point to give a new point. At no time does the vector represent a position.

It is sometimes more intuitive to think of vectors as arrows sticking out of the origin. This helps understand that they are not positions but simply directions. Imagining all vectors being grouped around the origin makes such operations as comparing two vectors, flipping their direction, or rotating them more intuitive.

## 3.1 MEL

A vector is defined using the `vector` data type. Because MEL doesn't make the distinction between points and vectors, all operations that can be performed on points can be applied to vectors.

## 3.2 C++ API

The C++ API makes a distinction between points and vectors. Whereas points are represented by the **MPoint** class, vectors are represented by the **MVector** class. The operations that can be performed on them conform to the mathematical rules set out previously. The **MVector** class has three coordinates: `x`, `y`, and `z`. All coordinates use the `double` data type. There also exists a `float` variation of this class, **MFloatVector**. The default constructor initializes the instance to the zero vector.

```
MVector vec; // x=y=z=0
```

Even though Maya makes the mathematical distinction between points and vectors, for convenience the **MPoint** and **MVector** classes can be easily converted from each other.

```
MPoint pt;
MVector vec;
pt = vec;
vec = pt;
```

Instances of **MFloatPoint** and **MFloatVector** can be converted to instances of **MPoint** and **MVector**, respectively. Note that when converting an **MPoint** to an **MVector** Maya assumes that the point is already in Cartesian coordinates. If it isn't, simply call the `cartesianize` function before assignment. For convenience, several predefined instances of **MVector** exist.

```
MVector::zero  // vector (0,0,0)
MVector::xAxis // vector (1,0,0)
MVector::yAxis // vector (0,1,0)
MVector::zAxis // vector (0,0,1)
MVector::xNegAxis // vector (-1,0,0)
MVector::yNegAxis // vector (0,-1,0)
MVector::zNegAxis // vector (0,0,-1)
```

These can be used like regular class instances, as in the following example.

```
MVector v0;
if( v0 == MVector::xAxis )
   MGlobal::displayInfo( "vector is the same as the x axis" );
```