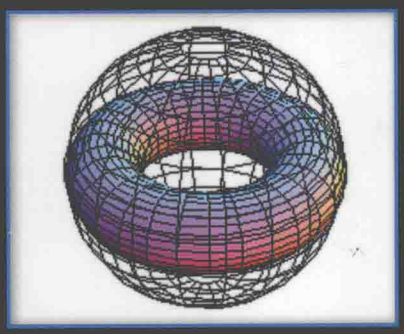
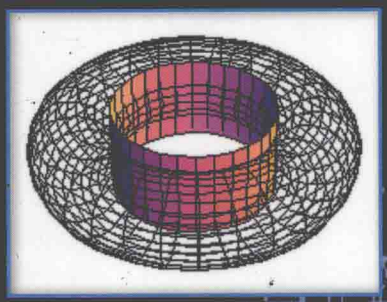
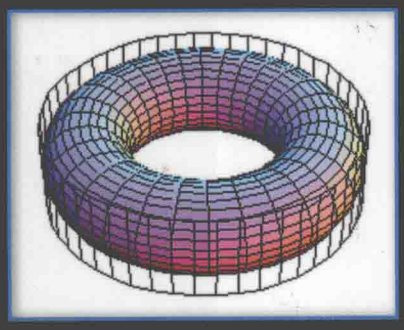
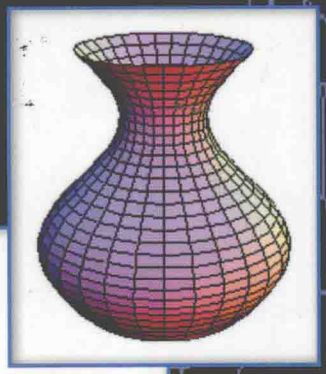
# An Integrated Introduction to

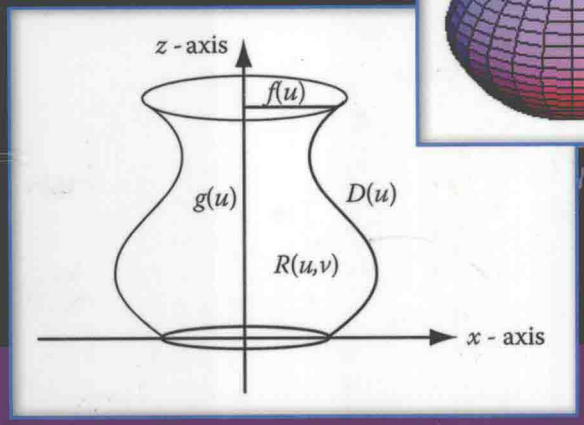
# Computer Graphics and Geometric Modeling












Ronald Goldman

CRC Press
Taylor & Francis Group

# An Integrated Introduction to *Computer Graphics and Geometric Modeling*

Ronald Goldman



CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

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



Printed in the United States of America on acid-free paper
10 9 8 7 6 5 4 3 2 1

International Standard Book Number: 978-1-4398-0334-9 (Hardback)



**Library of Congress Cataloging-in-Publication Data**

Goldman, Ron, 1947-
An integrated introduction to computer graphics and geometric modeling / Ronald Goldman.
p. cm.
Includes bibliographical references and index.
ISBN 978-1-4398-0334-9 (alk. paper)
1. Computer graphics. 2. Three-dimensional display systems. 3. Curves on surfaces--Mathematical models. I. Title.

T385.G6397 2009
006.6--dc22 2008054783

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

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

# An Integrated Introduction to *Computer Graphics and Geometric Modeling*

# *Foreword*

The field of computer graphics has reached a level of maturity, as evidenced by the graphics capability that is ubiquitous on even entry-level personal computers and by the prevalence of graphics software that is used to create stunning images and animations by artists who need no knowledge of what is going on under the hood. At the same time, the field of computer graphics continues to expand and evolve, as evidenced by the increasing number of research papers written on the topic from one year to the next. This rapid growth poses a challenge for the author of a new book on computer graphics, who must assess what subset of the ever-expanding body of knowledge will be of greatest benefit to the students, and will have the longest shelf life.

Several good books have been written on computer graphics over the years, and many of them are currently available in advanced editions. They span a spectrum from encyclopedic to nuts-and-bolts programming. This book offers a fresh approach. It is discretized into "lectures," organized to fill a semester-long introductory course with one chapter for each of the 30 class periods. The topics chosen cover most of the key concepts of computer graphics. The lectures on mathematical foundations and on geometric modeling are particularly strong. The book is void of programming examples, since the transitory nature of graphics languages would soon render such material outdated.

The author has a distinguished career as a developer, researcher, and educator in computer graphics. After earning a PhD in mathematics, he worked for many years in the young computer graphics and computer-aided design (CAD) industries, where he contributed to early graphics software development. While thus employed, he took an interest in mentoring several PhD students at various universities across the country, even though it was not formally part of his job. He did this partly because he loves research, but even more because he loves helping students succeed. As one of those fortunate students, I can attest to his infectious enthusiasm for his subject matter, his lucid explanations, his noise-free writing style, and his mathematical rigor. Dr. Goldman has dedicated the past 20 years of his career to teaching and research as a professor of computer science at the University of Waterloo and Rice University. The pedagogical style of this book has been refined during his many years of teaching this material. He is an excellent mentor of students and I am pleased that his reach will be extended through the publication of this book.

**Thomas W. Sederberg**
*Brigham Young University*

*To the Logos,*
*–Wordsmith and Mathematician Incarnate–*
*Co-Eternal Consort of the Creator,*
*Who on the First Day spoke the Word and made Light,*
*And saw that it was Good.*

**יהי אור**

*Fiat Lux.*
*Let there be Light.*

*Genesis 1:3*

# *Preface*

*Behold, the days come, saith the LORD, that I will make a new covenant...*

– Jeremiah 31:31

The good news is that Computer Graphics is fun: fun to look at, fun to use, and when done properly even fun to program and debug. There are also many fun applications of Computer Graphics, ranging from video games, to animated cartoons, to full-length feature movies. If you learn Computer Graphics and Geometric Modeling, you might even get a job in a field where you can have lots of fun. Art and architecture, biomedical imaging, computational photography: whatever you can see, or whatever you imagine you can see, you can design with Geometric Modeling and you can display with Computer Graphics.

Yet for a long time now and for many hapless college students, university courses on Computer Graphics and Geometric Modeling seem to oscillate from tedious to abstruse. Ponderous books and pedantic professors appear to focus endlessly on low-level techniques—line drawing, polygon filling, antialiasing, and clipping—and they elicit little or no connection between Computer Graphics and the rest of Computer Science. Good pedagogy is slighted: levels are mixed; mathematics abused; intuition neglected; elegance avoided; and mysteries ignored. Fun and excitement are drained from the subject. Sadly, I admit, I have taught such courses myself. Collectively, the field must do better. This book is intended as a new testament, a new inspiration for teachers and students alike.

This canon is intentionally short. This book is not an encyclopedia of Computer Graphics, but rather a brief introduction to the subject, essentially what can be taught to advanced undergraduate students and beginning graduate students majoring in Computer Science in a 15 week one semester course. Nevertheless, this book does cover many of the major themes of the discipline.

Broadly, these major themes can be divided into three categories: graphics, modeling, and mathematical foundations. Graphics consists of lighting and shading—reflection and refraction, recursive ray tracing, radiosity, illumination models, polygon shading, and hidden surface procedures. Modeling is the theory of curves, surfaces, and solids—planes and polygons, spheres and quadrics, algebraics and parametrics, constructive solid geometry, boundary files, and octrees, interpolation and approximation, Bezier and B-spline methods, fractal algorithms, and subdivision techniques. The mathematical foundations are mostly linear algebra, but from a somewhat idiosyncratic perspective not typically encountered in standard linear algebra classes—vector geometry and vector algebra, affine spaces and the space of mass-points, affine maps and projective transformations, matrices, and quaternions.

The subject of Computer Graphics is still relatively new. I shall not put new wine in old bottles. Rather I have deliberately sought innovative techniques for presenting this material to Computer Science students. I have borrowed extensively from other authors, but I have rearranged and reordered the topics and I have developed approaches that I believe are elegant to present, appealing to learn, and sensible pedagogically. In contrast to the standard concentration on low-level graphics algorithms, I focus instead on more advanced graphics, modeling, and mathematical methods—ray tracing, polygon shading, radiosity, fractals, freeform curves and surfaces, vector methods, and transformation techniques. Here is how I have organized this material.

*Fractals first.* The course begins with a topic that is visually appealing, intellectually deep, and naturally connected to the rest of Computer Science. Fractal shapes are *recursion made visible.* Computer Science students are used to writing recursive programs; now, perhaps for the first time, these students get to see recursion in visual art as well as in computational science. Fractals are exciting at many levels: visually, intellectually, and computationally. Students can be hooked into learning lots of important technical material in order to generate neat fractals.

I introduce fractals using turtle graphics. LOGO and turtle graphics have been popularized as systems for teaching programming to young children. But LOGO is also a powerful vehicle for studying Computer Graphics and Computational Geometry. This book promotes the turtle as a simple and effective way of introducing many of the fundamental concepts that underlie contemporary Computer Graphics.

In the version of LOGO employed here, the turtle's state is represented by a point $P$ specifying the turtle's location and a direction vector $v$ specifying the turtle's heading in the plane. There are commands such as FORWARD and TURN for altering the turtle's position and orientation. The turtle draws curves by joining with straight lines consecutive points along its path as it moves around in the plane under the control of a turtle programmer.

Much can be learned from this simple turtle. The FORWARD and TURN commands are equivalent to translation and rotation, so students get an early introduction to these fundamental transformations of the graphics pipeline, albeit in two dimensions. (There is also a RESIZE command to change the length of the direction vector, so students get to see scaling along with translation and rotation.) Internally, computations are performed using rectangular coordinates, but turtle programmers have no access to these coordinates. Thus high-level commands are kept distinct from low-level computations, a standard theme of Computer Science that will be stressed repeatedly throughout this book. Students can also write a simple interpreter for LOGO to hone skills learned in other Computer Science courses.

*Points translate, vectors rotate and scale.* Thus, although internally points and vectors in the plane may both be represented by pairs of rectangular coordinates, points and vectors are treated differently in turtle graphics. This distinction between points and vectors persists throughout this book and indeed throughout Computer Graphics; the turtle is a convenient model for introducing this very basic, but often overlooked, distinction. The turtle also anticipates the use later in this book of affine coordinates to distinguish between points and vectors.

Students can generate fractals such as the Sierpinski triangle and the Koch snowflake via simple recursive turtle programs. Thinking about recursive programs to generate concrete visual effects often enhances students' deep understanding of recursion.

Recursive turtle programs to generate fractals are typically easy to write; there is usually an obvious base case, and the body of the recursion consists of recursive calls connected by distinct sequences of the basic turtle commands. Nevertheless, in many such fractal programs a mystery soon appears. Although the choice of the base case seems constrained, changing the base case does not appear to alter the fractal generated in the limit of the recursion. This mystery demands an explanation, and this explanation leads to an important alternative approach to generating fractals—iterated functions systems.

An *iterated function system* is just a finite collection of contractive transformations. Repeatedly applying a fixed set of contractive transformations to a compact set generates a fractal. For particular fractals such as the Sierpinski gasket, one can often easily guess the transformations that generate the fractal. Nevertheless, this general mathematical approach to fractals is much more difficult to motivate to Computer Science students than recursive turtle programs. However, a straightforward analysis of recursive turtle programs reveals that the fractal generated by a recursive turtle program is equivalent to applying the iterated function system generated by the turtle commands in the recursion to the turtle geometry generated by turtle commands in the base case. Thus the analysis of recursive turtle programs motivates the study of iterated function systems.

Moreover, understanding iterated function systems is the key to understanding the mystery students encounter when studying fractals generated by recursive turtle programs—that changing the base case does not appear to alter the fractal generated in the limit of the recursion. The central result is the *trivial fixed point theorem*, which states that iteration of a contractive map on a point in a complete metric space always converges to a unique fixed point. There are some technical mathematical difficulties to overcome in order for students to understand the statement and proof of this fixed point theorem, so strong motivation is essential. The fractal mystery of recursive turtle programs provides this motivation.

Understanding this theorem is well worth the effort because this fixed point theorem will be invoked again later in the course of this book in the chapter on radiosity. Both the Jacobi and the Gauss-Seidel relaxation methods for solving large systems of linear equations are based on this trivial fixed point theorem. Recursive root finding algorithms for certain transcendental equations can also be derived from this theorem. Toward the end of this book, using subdivision, students will discover that Bezier and B-spline curves are also fixed points of iterated function systems. Thus, somewhat surprisingly, smooth polynomial and piecewise polynomial curves are also intimately related to fractals.

The study of fractals from the perspective of turtle graphics and iterated function systems typically takes about four weeks, more that one-quarter of a fifteen-week semester. This time is well spent. In addition to rendering visually exciting graphics, student are introduced to points and vectors, affine coordinates and affine transformations, matrix computations, and an important fixed point theorem. They also learn to distinguish clearly between high-level concepts and low-level computations. With this preparation, students are now ready to move up to three dimensions.

Three dimensions require new mathematical foundations. In two dimensions, students can get by with coordinate geometry, but in three dimensions coordinates can be confusing and often actually get in the way of the analysis. Therefore Part II of this book begins with a thorough review of three-dimensional vector geometry: addition, subtraction, scalar multiplication, dot product, cross product, and determinant. Most Computer Science students have seen vector algebra before either in courses on physics or linear algebra, but their geometric understanding of the vector operations is still tenuous at best. A thorough review of vector methods from a geometric perspective is in order here to prepare the students for modeling and analyzing geometry in three dimensions.

Computer Graphics deals with points as well as with vectors—points, not vectors, are typically displayed on the graphics terminal—so this book discusses affine spaces (spaces of points) and affine transformations along with vector spaces and linear transformations. Affine spaces are new to most students—the restriction to affine combinations may appear artificial at first—but this unfamiliarity is all the more reason to adopt coordinate-free methods. When the levels are not mixed, when theory is kept separate from computation, these concepts are much easier to explain and simpler to understand. Later, students will see that the distinction between points and vectors is computational as well as theoretical: points translate, vector do not.

Vector techniques are applied right away to derive coordinate-free vector formulations for all the affine and projective transformations commonly used in Computer Graphics—translation, rotation, mirror image, scaling, shearing, and orthogonal and perspective projections—before matrix methods are introduced. These vector formulas emphasize the distinction between transformations (high-level concepts) and their matrix representations (low-level computational tools), notions that are too often confused in the minds of the students. When matrices are introduced later to speed up calculations, these vector formulas are invoked to derive matrix representations for each of the corresponding transformations. Because the original vector formulas are coordinate free, these matrix representations are not confined to describing projections into coordinate planes or rotations around coordinate axes, but work for planes and axes in general position.

Vector geometry replaces coordinate computations throughout this book. Coordinates are confined to low-level subroutines for calculating addition, subtraction, scalar multiplication, dot

product, cross product, and determinant. Vector methods are applied ubiquitously to derive metric formulas, surface equations, and intersection algorithms; vector algebra is applied to calculate normal vectors for lighting models, and to generate reflection and refraction vectors for recursive ray tracing.

The proper uses of coordinates are to communicate with a computer and to speed up certain common computations. When coordinates are introduced, the theoretical distinction between points and vectors in affine space leads naturally to the insertion of a fourth coordinate, an *affine coordinate*, to distinguish between the rectangular coordinates for points and vectors: the affine coordinate is one for points and zero for vectors. This affine coordinate leads to the adoption of $4 \infty 4$ matrices to represent affine transformations.

Affine transformations on affine spaces are similar to linear transformations on vector spaces. Affine transformations map points to points, vectors to vectors, and preserve affine combinations. Thus affine transformations are represented by matrices that preserve the fourth, affine coordinate. Translation, rotation, mirror image, scaling, shearing, and orthogonal projection are all affine transformations.

Regrettably, affine spaces and affine transformations are not sufficient to model all the geometry encountered in Computer Graphics. Perspective projection is not an affine transformation. Perspective is not a well-defined transformation on vectors; moreover, points on the plane through the center of projection parallel to the plane of projection are not mapped to affine points but seem rather to be mapped to infinity by perspective projection. Thus a more general ambient space incorporating a more general collection of transformations is required to accommodate perspective projections.

Most books on Computer Graphics adopt projective spaces and projective transformations. Projective transformations include all the affine transformations as well as perspective projection. Nevertheless, projective spaces have several major drawbacks that make them unsuitable either as an algebraic or as a geometric foundation for Computer Graphics.

Projective space contains two types of points: *affine points and points at infinity*. The points at infinity complete the geometry of affine space—parallel lines in affine space meet at points at infinity in projective space—so perspective projection is defined at all points in projective space except the center of projection. But there is a big price to pay for these points at infinity.

The points at infinity in front of an observer are identical to the points at infinity behind the observer. Therefore gazing along an unobstructed direction in projective space, an observer would see the back of their head! Orientation—up and down, left and right, front and back—plays a fundamental role in the visual world, but there is no notion of orientation in projective space. This peculiar geometry of projective spaces is nonintuitive and difficult for most mathematically unsophisticated students of Computer Science to understand.

Worse yet, the points at infinity in projective space supplant the vectors in affine space. To adopt projective geometry, Computer Graphics would have to abandon the vector geometry that provides the foundation for most of the required mathematical analysis. But realistic lighting models depend on vectors normal to surfaces: diffuse and specular illumination, reflection and refraction computations all make use of surface normal vectors.

Finally, projective space is not a linear space. One cannot add points, or even take affine combinations of points, in projective space. Thus projective space is not a suitable model for most computer computations. Matrix multiplication is allowed, so the standard transformations in the graphics pipeline can be accommodated in projective space. But without an algebra for points it is impossible to construct many of classical parametric curves and surfaces, such as Bezier or B-spline curves and surfaces, inside projective space. Similarly, with no notion of addition or scalar multiplication, projective spaces fail to support shading algorithms based on linear interpolation.

All of these basic problems—theoretical and computational, geometric and algebraic—can be overcome by replacing projective space with the space of mass-points.

Mass-points form a vector space. To multiply a mass-point by a scalar, simply multiply the mass by the scalar (masses are allowed to be negative); to add two mass-points, apply Archimedes' law of the lever—place a mass equal to the sum of the two masses at the center of mass of the original two

mass-points. The space of mass-points is a four-dimensional vector space: three dimensions are due to the points, the fourth dimension is induced by the masses. Vectors are incorporated into the space of mass-points as objects with zero mass; affine space is embedded in the space of mass-points by setting the masses of all the points to one.

Replacing projective space by the space of mass-points, replaces a compact, nonorientable, nonlinear three-dimensional manifold by a four-dimensional vector space. Going up in dimension removes the geometric incongruities of projective space and facilitates algebraic computations while modifying only slightly the formal representation.

The space of mass-points has all of the advantages, but none of the disadvantages, of projective space. The space of mass-points incorporates the points and vectors of affine space, so vector algebra and vector geometry make sense in the space of mass-points. The space of mass-points is a linear space, so it is possible to construct Bezier and B-spline curves and surfaces, as well as their rational variants, inside the space of mass-points. Since the space of mass-points is a vector space, the natural transformations on the space of mass-points are linear transformations. The linear transformations that preserve mass are precisely the affine transformations; perspective projections are also linear transformations on the space of mass-points, albeit ones that do not preserve mass.

There are other rewards for working in the space of mass-points. Archimedes' law of the lever can be applied in the space of mass-points to derive the formula for perspective projection from any center of projection into any plane. Also the classical map from the viewing frustum to a rectangular box can be derived simply by adding to the image of an arbitrary point under perspective projection the depth vector from the point to the plane of projection. This addition, however, is the addition of mass-points and vectors in the space of mass-points, not the ordinary addition of points and vectors in affine space. Perspective projection in the space of mass-points introduces additional mass to affine points, and this mass must be taken into account to get the correct formula for the map from the viewing frustum to the rectangular box.

Since the space of mass-points is a four-dimensional vector space, four coordinates are needed to represent mass-points. The fourth coordinate stores the mass; the first three coordinates store the rectangular coordinates of the point multiplied by the mass. Thus the rectangular coordinates of the points can be retrieved from these four coordinates by dividing the first three coordinates by the mass. Notice how coordinates for mass-points extend affine coordinates by permitting the fourth coordinate, the mass, to take on any value.

These coordinates for mass-points resemble homogeneous coordinates for projective points but with this difference: the same affine point with different masses in the space of mass-points represents distinct mass-points; the same affine point with different homogeneous coordinates in projective space represents the same projective point. In fact, projective points are just equivalence classes of mass-points, where two mass-points are equivalent if the masses are located at the same point in affine space.

The space of mass-points is a four-dimensional vector space, so linear transformations on the space of mass-points are represented by $4 \times 4$ matrices. Thus the same matrices are used to represent the same transformations on affine space in both the space of mass-points and projective space. Therefore the familiar computational formalism of homogeneous coordinates and projective transformations remains valid in the space of mass-points.

But there is a big bonus here: we can also multiply vectors in four dimensions. The only real vector spaces that have an associative multiplication with inverses are the real numbers (one dimension), the complex numbers (two dimensions), and the quaternions (four dimensions). Thus quaternion multiplication is multiplication in the space of mass-points. It is not an anomaly that we are working in four dimensions instead of three dimensions; rather we are incredibly lucky, since in four dimensions we can take advantage of this additional multiplicative structure.

Classically, one can use the richness of this quaternion algebra to represent conformal transformations on vectors in three dimensions by sandwiching vectors between quaternions rather than by multiplying vectors by $4 \times 4$ matrices. Thus quaternions provide more compact representations for

conformal transformations than $4 \times 4$ matrices. Moreover, composing conformal transformations by multiplying quaternions is faster than composing conformal transformations by matrix multiplication.

Quaternions are used most effectively in Computer Graphics to avoid distortions during conformal transformations (quaternions are easy to renormalize) and to perform key frame animation (by spherical linear interpolation). There is a good deal of folklore about quaternions scattered throughout the literature, but I have yet to find a good textbook treatment of quaternions for Computer Graphics. Fortuitously, the four-dimensional vector space of mass-points incorporates quaternions in a natural way; we do not need to introduce quaternions as an additional *ad hoc* structure unrelated to projective space.

With the proper mathematical foundations—vector geometry and vector algebra, mass points and quaternion multiplication, affine transformations and perspective projections—the technical tools are now in place to study three-dimensional Computer Graphics.

Realistic rendering can be accomplished in three ways: ray tracing, polygon shading, and radiosity. This book covers all three of these topics. Conceptually, the most straightforward method is recursive ray tracing, so the text treats this method first.

Recursive ray tracing is based on the recursive computation of reflection and refraction rays. Two innovations are introduced here by taking advantage of the transformations studied in the previous chapters: reflection rays can be computed from the law of the mirror using the mirror image map for arbitrary mirror planes; refraction rays can be computed from Snell's law using the rotation transformation around arbitrary axes. In fact, these approaches to reflection and refraction are left as easy exercises for the students. The text adopts an even simpler approach, decomposing reflection and refraction vectors into orthogonal components and analyzing each component directly using the appropriate optical laws and the simple mathematics of dot products and cross products. Thus because the proper mathematical foundations are in place, reflection and refraction rays are easy to compute.

Ray casting is a compelling topic only if students have visually interesting surfaces to render. Therefore this subject provides a natural time and place to introduce some classical surfaces. The study of surfaces is confined here to the investigation of those properties and computations necessary for ray tracing: surface equations, surface normals, and ray–surface intersections. Again vector methods and transformation techniques play a key role in an innovative approach to ray tracing surfaces.

The sphere is the simplest nonplanar surface. Vectors normal to the surface of a sphere are parallel to the vectors from the center to points on the surface of the sphere, and the intersection of a line and a sphere can be reduced to the intersection of a line and a coplanar circle. Once the sphere is mastered, several other quadric surfaces can easily be analyzed using affine and projective transformations.

An ellipsoid is the image of a sphere under nonuniform scaling. Thus surface normals on the ellipsoid can be computed by finding surface normals on the sphere and mapping these normals onto the ellipsoid. The intersections of a ray and an ellipsoid can be calculated in a similar fashion by mapping the ray and the ellipsoid to a ray and a sphere, calculating the intersections of the ray and the sphere, and then mapping the resulting intersection points back to the ellipsoid.

Cylinders and cones are next. A sphere is the locus of points equidistant from a point; a cylinder is the locus of points equidistant from a line. From this definition it is easy using vector techniques to compute normals to the surface of the cylinder. Similar definitions and normal vector computations are developed for the cone. The intersections of a ray and a cylinder can be calculated by applying orthogonal projection onto a plane perpendicular to the cylinder axis to map the ray and the cylinder to a ray and a circle, then invoking the algorithm already developed for ray tracing the sphere to calculate the intersections of the ray and the circle, and finally using the parameters of the intersection points on the line to find the intersection points on the cylinder. The intersections of a ray and a cone can be calculated in a similar fashion, just replace orthogonal projection by perspective projection from the vertex of the cone.

Arbitrary algebraic and general parametric surfaces can also be ray traced. General algorithms are presented for computing surfaces normals and ray–surface intersections for these broad surface types. The torus can be represented both as an algebraic and as a parametric surface. Thus these

general algorithms can be applied to a geometrically familiar, computationally tractable (fourth-degree) surface.

Solid modeling follows right after surface modeling. Three common approaches to solid modeling are investigated: constructive solid geometry (CSG), boundary file representations (B-REP), and octrees. Constructive solid geometry fits in well with the preceding study of ray tracing and surface modeling. The primitives in CSG trees are typically solids bounded by planes, natural quadrics, and tori. CSG trees can be rendered and their mass properties can be calculated by ray casting algorithms. Boundary file representations introduce the notion of topology—connectivity information—alongside geometry—shape information—to facilitate searching the model for important features. Since conversion algorithms from CSG trees to boundary file representations are often quite complicated, boundary files are frequently restricted to polygonal models. To render these polygonal models to appear as smooth curved surfaces requires polygon-shading techniques.

Polygon shading is the second of the three realistic rendering techniques presented in this book. Both Gouraud and Phong shading are covered here. Although these two shading algorithms are low level, scan line procedures, vector methods are still stressed in order to develop clever, fast, incremental implementations. Spherical linear interpolation is introduced for Phong shading, taking advantage of the students' previous encounter with the use of spherical linear interpolation for quaternions representing rotations in key frame animation.

Hidden surface algorithms are required for realistic rendering of polygonal models. Many hidden surface algorithms are available, and the text surveys four representative procedures: $z$-buffer, scan line, depth sort, and BSP-tree. The $z$-buffer algorithm is the easiest to implement, but the scan line algorithm is the hidden surface procedure that integrates best with Gouraud and Phong shading. Only while describing scan line algorithms does this book intentionally descend into low level, coordinate techniques. It is precisely when speed is at a premium that coordinate techniques are appropriate. Students learn that there are indeed times when it is expedient to descend to coordinate-based methods for fast rendering (*Render unto Caesar* . . . ). Depth sort, in contrast, provides some subtle applications of vector techniques, both for measuring relative depth and for finding obstructing planes. Finally, BSP-trees are a well-known and important data structure in Computer Science; BSP-trees are most useful for finding hidden surfaces when the model is fixed, but the viewpoint or the light sources can change position.

Radiosity is the rendering method that presents the most realistic diffuse images. In contrast to recursive ray tracing, radiosity softens shadows and portrays color bleeding. To introduce radiosity, the text begins with the continuous form of the rendering equation and then successively simplifies this integral into a large discrete system of linear equations. Jacobi and Gauss-Seidel relaxation techniques are recommended for solving these linear systems, techniques already familiar to students from the fixed point methods encountered during the investigation of fractals. Gathering and shooting are both investigated here.

Freeform curves and surfaces constitute the final topic presented in this book. So far, students have rendered only a limited number of rigid surfaces, mostly planes and quadrics. But in order to faithfully represent a rich variety of forms including car bodies, ship hulls, airplane wings, toys, shoes, and even many animated cartoon characters, geometric modeling deals mainly with freeform shapes. Therefore this book closes with techniques for representing, analyzing, and rendering freeform curves and surfaces. Typical textbooks in Computer Graphics provide at best only a cursory *ad hoc* introduction to Bezier and B-spline techniques. In contrast, this book delivers a thorough, unified approach to Bezier and B-spline approximation, as well as to subdivision surfaces. Since these topics generally require some mathematical sophistication, several innovations are introduced here to simplify the presentation.

Linear interpolation is already familiar to students from Gouraud and Phong shading. Dynamic programming procedures based on successive linear interpolation are provided for Bezier and B-spline curves and surfaces: for Bezier curves and surfaces this procedure is called the *de Casteljau algorithm*; for B-splines the method is known as the *de Boor algorithm*.

Figures are often easier to understand than formulas. This book uses simple data flow diagrams—pyramid algorithms—to develop a straightforward, common approach to these dynamic programming procedures. Students are encouraged to bypass formulas and reason directly from these diagrams. A common approach permits common derivations for common properties such as affine invariance and nondegeneracy. Explicit expressions and as well as recursion formulas for the blending functions of Bezier curve and surface schemes can also be generated directly from these diagrams.

Blossoming provides the simplest, most direct approach to deriving many of the unique properties of Bezier and B-spline curves and surfaces. Therefore, following the maxim that students should *use the mathematics most appropriate to the problem at hand*, I have tried to present a clear and unintimidating introduction to blossoming. While blossoming will be new to most students, blossoming is not difficult. In addition to introducing the blossom axiomatically, the text also presents the blossom concretely using pyramid algorithms. The pyramid diagram for the blossom is identical to the pyramid diagram for Bezier curves with one slight variation: a different parameter is introduced on each level of the pyramid algorithm. Thus linear interpolation is also at the heart of blossoming. The standard properties of the blossom—symmetry, multiaffinity, the diagonal property, the dual functional property, as well as existence and uniqueness—are all derived here directly from these pyramid diagrams.

Bezier schemes are intimately related to blossoming because the corresponding pyramid algorithms are so closely connected. It follows from the dual functional property that the blossom of a univariate polynomial evaluated at the end points of a parameter interval yields the corresponding Bezier control points. Therefore blossoming provides a general approach to change of basis procedures for Bezier schemes. Conversion between monomial and Bezier form, degree elevation techniques, and subdivision algorithms are all derived here from blossoming. Moreover, the homogenous variant of the blossom is used to derive a differentiation algorithm for Bezier curves based on the de Casteljau algorithm.

B-splines are also closely related to blossoming. A slight modification of the blossoming interpretation of the de Casteljau algorithm for Bezier curves—starting with the blossom evaluated at consecutive knots instead of at the end points of the parameter domain—generates the de Boor evaluation algorithm for B-spline curves. Many introductory books simply take the de Boor algorithm as the definition of the B-splines. But without appropriate motivation, students are at a loss to understand what is so special about this particular recursion formula. Blossoming provides the proper motivation as well as the natural connection between the de Casteljau algorithm and the de Boor algorithm. As with Bezier curves, the blossom of a spline evaluated at the knots yields the corresponding B-spline control points. Therefore blossoming also provides a general approach to change of basis procedures for B-splines, so knot insertion algorithms for B-splines are readily derived here from blossoming. As with Bezier curves, the homogenous variant of the blossom is used to derive a differentiation algorithm for B-splines curves based on the de Boor algorithm, and this differentiation algorithm is used in turn to prove that the polynomial segments of B-spline curves join smoothly at the knots.

Subdivision is the key to rendering Bezier surfaces. The de Casteljau subdivision algorithm generates a polyhedral approximation that converges to the Bezier surface under recursive subdivision. This polyhedral approximation can then be rendered using either ray tracing, or polygon shading, or radiosity. Knot insertion algorithms play the analogous role for B-spline surfaces.

Subdivision can be implemented using matrices to multiply the control points. For Bezier schemes these subdivision matrices form an iterated function system. Hence, rather remarkably, Bezier curves and surfaces can be rendered as fractals by applying this iterated function system to an arbitrary compact set. Thus the book comes full circle, recapitulating close to the very end the fractal methods introduced almost at the very beginning.

Knot insertion for B-splines is the analogue of subdivision for Bezier curves and surfaces. Knot insertion can also be represented by matrix multiplication and these matrices also form an iterated

function system. Thus B-spline curves and surfaces can also be rendered as fractals by applying this iterated function system to an arbitrary compact set. One additional innovation here is that we show how to extend the Lane-Riesenfeld subdivision algorithm for B-splines with uniform knots to a subdivision algorithm due to Scott Schaefer for B-splines with nonuniform knots.

The Lane-Riesenfeld knot insertion algorithm for B-splines with uniformly spaced knots is the principal paradigm for more general subdivision schemas. This book closes with a chapter containing a brief introduction to general subdivision surfaces for rectangular and triangular meshes, including box splines, centroid averaging, Loop subdivision, and Catmull-Clark subdivision, topics still at the frontier of current research. Three simple paradigms are employed to explain each of these methods: split and average (box splines), centroid averaging (meshes of arbitrary topology), and stencils—vertex, edge, and face stencils as well as special stencils for extraordinary vertices (Loop subdivision and Catmull-Clark subdivision).

This concludes the brief survey of the topics covered in this book. Many topics have purposely been omitted from this text, not because they are uninteresting or unimportant, but simply because they are either too advanced—physics-based modeling, scientific visualization, virtual reality—or too specialized—user interfaces, graphics hardware, input devices—for an introductory course.

A good book is a compendium of the abiding past, a snapshot of the transitory present, and a guess at the unknowable future. Writing a good book, much like teaching a compelling course, is about making choices: what to include and what to exclude are dictated by time and taste. I have tried to concentrate on enduring themes and to avoid ephemeral motifs. Graphics hardware and software will certainly soon change, but graphics algorithms based on well-established physical models of light and cogent mathematical methods will last a long time. Therefore I have avoided descriptions of current graphics hardware, and I have omitted from this book special programming languages such as C++ and API's like OpenGL. Low-level graphics algorithms such as line drawing, polygon filling, and clipping are bypassed to give more time and space to high-level graphics techniques such as ray tracing, polygon shading, and radiosity.

I have tried to write a book that is exciting to read without being superficial, rigorous without being pedantic, and innovative without being idiosyncratic. I have kept the manuscript relatively short in the hope that students and lecturers alike will read it in full. Lots of exercises and projects are included to flesh out this book, but there are many topics that I have consciously chosen not to cover. My intention has been to write a *A Guide for the Perplexed*, not a *Summa Theologica*.

Certainly I should thank all those who came before me in the field of Computer Graphics—founders and innovators, scientists and mathematicians, pure academics and practicing engineers, serious students, and conscientious professors. There can be no new testament without an older revelation on which to build. And yet... Siggraph is attended by tens of thousands of people, and hundreds of paper are submitted to Siggraph each year. If even only a small percentage of these people and papers are important, I could not hope to list them all here.

Let me close instead by asking forgiveness from my former students, whom I abused with my lackluster teaching in the past. I hope I have done better by them here. I have drawn inspiration from my forebears, encouragement from my colleagues, and constructive criticism from my graduate students. I trust this constellation is sufficient to the task.

The days of any book are numbered. This book, like its predecessors, will inevitably become obsolete due to innovations in theory and technology. Three-dimensional graphics hardware is now technically feasible. If this hardware becomes popular, the current concentration on two-dimensional projections may someday be outdated, and user interfaces will certainly change. Innovations in mathematics such as Clifford algebras or in physics such as quantum computing may eventually make other parts of this book seem as stodgy and old fashioned as the tomes it is written to supplant. Instructors should keep these issues in mind when choosing a textbook for their classes in the future. Authorities argue, barriers breakdown, canons change, consensus crumble, disciplines decline, epistemologies expire, fashions fade, ideologies implode, laws languish, methods mutate, orthodoxies ossify, paradigms perish; *time and chance happen to them all.*

# *Author*

**Ron Goldman** is a Professor of Computer Science at Rice University in Houston, Texas.

Professor Goldman received his BS in Mathematics from the Massachusetts Institute of Technology in 1968 and his MA and PhD in Mathematics from Johns Hopkins University in 1973. He is an Associate Editor of *Computer Aided Geometric Design*. In 2002 he published a book entitled *Pyramid Algorithms: A Dynamic Programming Approach to Curves and Surfaces for Geometric Modeling*.

Dr. Goldman's current research interests lie in the mathematical representation, manipulation, and analysis of shape using computers. His work includes research in computer-aided geometric design, solid modeling, computer graphics, and splines. He is particularly interested in algorithms for polynomial and piecewise polynomial curves and surfaces, and he is currently investigating applications of algebraic and differential geometry to geometric modeling. He has published over a 100 articles in journals, books, and conference proceedings on these and related topics.

Before returning to academia, Dr. Goldman worked for 10 years in industry solving problems in computer graphics, geometric modeling, and computer-aided design. He served as a Mathematician at Manufacturing Data Systems Inc., where he helped to implement one of the first industrial solid modeling systems. Later he worked as a Senior Design Engineer at Ford Motor Company, enhancing the capabilities of their corporate graphics and computer-aided design software. From Ford he moved on to Control Data Corporation, where he was a Principal Consultant for the development group devoted to computer-aided design and manufacture. His responsibilities included database design, algorithms, education, acquisitions, and research.

Dr. Goldman left Control Data Corporation in 1987 to become an Associate Professor of Computer Science at the University of Waterloo in Ontario, Canada. He joined the faculty at Rice University in Houston, Texas as a Full Professor of Computer Science in July 1990.

# *Contents*