GRAPHICS GEMS

III

Edited by

DAVID KIRK

This is a volume in

The Graphics Gems Series

*A Collection of Practical Techniques
for the Computer Graphics Programmer*

Series Editor

Andrew S. Glassner
Xerox Palo Alto Research Center
Palo Alto, Califomia

GRAPHICS GEMS III

edited by

DAVID KIRK

**California Institute of Technology
Computer Graphics Laboratory
Pasadena, California**

About the Cover

This cover image evolved out of a team effort between The VALIS Group, Andrew Glassner, Academic Press, and the generous cooperation and sponsorship of the folks at Pixar. Special thanks go to Tony Apodaca at Pixar for post-processing the gems in this picture using advanced RenderMan® techniques. The entire cover image was created using RenderMan® from Pixar.

We saw the *Graphics Gems III* cover as both an aesthetic challenge and an opportunity to demonstrate the kind of images that can be rendered with VALIS' products and Pixar's RenderMan®.

Given the time constraints, all of the geometry had to be kept as simple as possible so as not to require any complex and lengthy modeling efforts. Since RenderMan® works best when the geometric entities used in a 3-D scene are described by high order surfaces, most of the objects consisted of surfaces made up of quadric primitives and bicubic patches. Andrew's gem data and the Archimedean solids from the VALIS Prime RIB™ library are-the only polygonal objects in the picture.

Once all of the objects were defined, we used Pixar's Showplace™, a 3-D scene arranging application on the Mac, to position the objects and compose the scene. Showplace also allowed us to position the camera and the standard light sources as well as attach shaders to the objects

In RenderMan®, shaders literally endow everything in the image with their own characteristic appearances, from the lights and the objects to the atmosphere. The shaders themselves are procedural descriptions of a material or other phenomenon written in the RenderMan® Shading Language. We did not use any scanned textures or 2-D paint retouching software to produce this picture.

Where appropriate, we used existing shaders on the surfaces of objects in this picture, taken from our commercially available VG Shaders™ + VG Looks™ libraries. For example,we used Hewn Stone Masonry (Volume 3) to create the temple wall and well; Stone Aggregate (Volume 3) for the jungle-floor, and polished metal (Volume 2) for the gold dish. In addition to these and other existing shaders, several new shaders were created for this image. The custom shaders include those for the banana leaves, the steamy jungle atmosphere, the well vapor, and the forest canopy dappled lighting effect.

Shaders also allowed us to do more with the surfaces than merely effect the way they are colored. In RenderMan®, shaders can transform simple surfaces into more complex forms by moving the surface geometry to add dimension and realistic detail. Using shaders we turned a cylinder into a stone well, spheres into boulders and rocks, and a flat horizontal plane into a jungle floor made up of stones and pebbles.

Similarly, we altered the surface opacity to create holes in surfaces. In this instance, we produced the ragged edges of the banana leaves and the well vapor by applying our custom RenderMan®, shaders to flat pieces of geometry before rendering with PhotoRealistic RenderMan®.

Initially, this image was composed at a screen resolution of 450×600 pixels on a MacIIfx using Showplace. Rendering was done transparently over the network on a Unix® workstation using Pixar's NetRenderMan™. This configuration afforded us the convenience and flexibility of using a Mac for design and a workstation for quick rendering and preview during the picture-making process.

Once the design was complete, the final version of the image was rendered at 2250×3000 pixel resolution. The final rendering of this image was done on a 486 PC/DOS machine with Truevision's RenderPak™ and Horizon860™ card containing 32 MBytes of RAM.

During the rendering process, RenderMan® separates shadows into a temporary file called a shadow map. The $2k \times 2k$ shadow map for this image was rendered in less than an hour. However, using shaders to alter the surface geometry increases rendering time and memory requirements dramatically. As a result, we had to divide the image into 64 separate pieces and render each one individually. The total rendering time for all 64 pieces was 41.7 hours. Once these were computed,

the TIFF tools from Pixar's RenderMan Toolkit™ were used to join the pieces together into a single, 33 MByte image file.

When the image was ready for output to film, we transferred the image file to a removable cartridge and sent it to a local output service. They output the electronic file onto 4×5 Ecktachrome color film and had it developed. The transparency was then sent to Academic Press in Cambridge, Massachusetts where they added the title and other elements to the final (over and had everything scanned and turned into four-color separations which were then supplied to their printer.

We hope you like this image. Producing it was fun and educational. As you might guess, many "graphic gems" were employed in the software used to produce this picture.

Mitch Prater, Senior Partner
Dr. Bill Kolomyjec, Senior Partner
RoseAnn Alspektor, Senior Partner
The VALIS Group
Pt. Richmond, CA
March, 1992
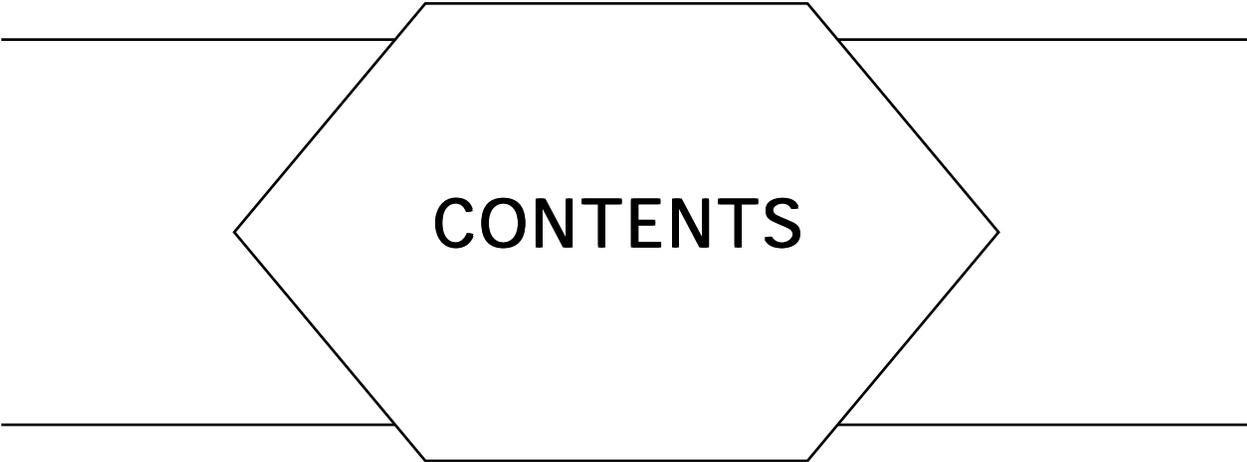
CONTENTS

The Symbol ◇C◇ denotes gems that have accompanying C implentations in the Appendix.

I
IMAGE PROCESSING

II
NUMERICAL AND PROGRAMMING TECHNIQUES

III
MODELING AND TRANSFORMATIONS

IV
2-D GEOMETRY AND ALGORITHMS

V
3-D GEOMETRY AND ALGORITHMS

VI
RAY TRACING AND RADIOSITY

VII
RENDERING

FOREWORD

by Andrew Glassner

Welcome to *Graphics Gems III,* an entirely new collection of tips, techniques, and algorithms for the practicing computer graphics programmer. Many ideas that were once passed on through personal contacts or chance conversations can now be found here, clearly explained and demonstrated. Many are illustrated with accompanying source code.

It is particularly pleasant for me to see new volumes of Gems, since each is something of a surprise. The original volume was meant to be a self-contained collection. At the last moment we included a card with contributor's information in case there might be enough interest to someday prepare a second edition. When the first volume was finished and at the printer's, I returned to my research in rendering, modeling and animation techniques.

As with the first volume, I was surprised by the quantity and quality of the contributions that came flowing in. We realized there was a demand,a need for an entirely new second volume, and *Graphics Gems II* was born. The same cycle has now repeated itself again, and we have happily arrived at the creation of a third collection of useful graphics tools.

Since the first volume of *Graphics Gems* was published, I have spoken to many readers and discovered that these books have helped people learn graphics by starting with working codes and just exploring intuitively. I didn't expect people to play with the codes so freely, but I think I now see why this helps. It is often exciting to start learning a new medium by simply messing around in it, and understanding how it flows. My piano teacher encourages new students to begin by spending time playing freely at the keyboard: keep a few scales or chord progressions in mind, but otherwise explore the spaces of melody, harmony, and rhythm. When I started to learn new mediums in art classes, I often spent time simply playing with the medium: squishing clay into odd shapes or brushing paint on paper in free and unplanned motions. Of course one often moves on to develop control and technique in order to communicate one's message better, hut much creativity springs from such uncontrolled and spirited play.

It is difficult for the novice to play at programming. There is little room for simple expression or error. A simple program does not communicate with the same range and strength as a masterfully simple line drawing or haunting melody. A programmer cannot hit a few wrong notes, or tolerate an undesired ripple in a line. If the syntax isn't right, the program won't compile; if the semantics aren't right, the program won't do anything interesting. There are exceptions to the latter statement, but they are notable because of their rarity. If you're going to write a program to accomplish a task, you've got to do some things completely right, and everything else almost perfectly. That can be an intimidating realization particularly for the beginner: if a newly constructed program doesn't work, the problem could be in a million places, anywhere from the architecture to data structures,

algorithms, or coding errors. The chance to start with something that already works removes the barrier to exploration: your program already works. If you want to change it, you can, and you will discover which new ideas work and which ones don't.

I believe that the success of the *Graphics Gems* series demonstrates something very positive about our research and practice communities. The modern view of science is an aggregate of many factors, but one popular myth depicts the researcher as a dispassionate observer who seeks evidence for some ultimate truth. The classical model is that this objective researcher piles one recorded fact upon another, continuously improving an underlying theoretical basis. This model has been eroded in recent years, but it remains potent in many engineering and scientific texts and curricula. The practical application of computer graphics is sometimes compared to a similarly theoretical commercial industry: trade secrets abound, and anything an engineer learns remains proprietary to the firm for as long as possible, to capitalize on the advantage. I do not believe that either of these attitudes are accurate or fruitful in the long run. Most researchers are biased. They believe something is true, from either experience or intuition, and seek support and verification for that truth. Sometimes there are surprises along the way, but one does not simply juggle symbols at random and hope to find a meaningful equation or program. It is our experience and insight that guide the search for new learning and limited truths, or the clear demonstration of errors of the guiding principle. I hail these prejudices, because they form our core beliefs, and allow us to choose and judge our work. There are an infinite number of interesting problems, and many ways to solve each one. Our biases help us pick useful problems to solve, and to judge the quality and elegance of the solution. By explicitly stating our beliefs, we are better able to understand them, emphasizing some and expunging others, and improve. Programmers of graphics software know that the whole is much more than the sum of the parts. A snippet of geometry can make a complex algorithm simple, or the correct, stable analytical solution can replace an expensive numerical approximation. Like an orchestral arranger, the software engineer weaves together the strengths and weaknesses of the tools available to make a new program that is more powerful than any component

When we share our components, we all benefit. Two products may share some basic algorithms, but that alone hardly makes them comparable. The fact that so many people have contributed to *Gems* shows that we are not afraid to demonstrate our preferences for what is interesting and what is not, what is good and what is bad, and what is appropriate to share with colleagues.

I believe that the *Graphics Gems* series demonstrates some of the best qualities in the traditional models of the researcher and engineer. Gems are written by programmers who work in the field who are motivated by the opportunity to share some interesting or useful technique with their colleagues. Thus we avoid reinventing the wheel, and by sharing this information, we help each other move towards a common goal of amassing a body of useful techniques to be shared throughout the community.

I believe computer graphics has the potential to go beyond its current preoccupation with photorealism and simple surfaces and expand into a new creative medium. The materials from which we will shape this new medium are algorithms. As our mastery of algorithms grows, so will our ability to imagine new applications and make them real, enabling new forms of creative expression. I hope that the algorithms in this book will help each of us move closer to that goal.

PREFACE

This volume attempts to continue along the path blazed by the first two volumes of this series, capturing the spirit of the creative graphics programmer. Each of the Gems represents a carefully crafted technique or idea that has proven useful for the respective author. These contributors have graciously allowed these ideas to be shared with you. The resulting collection of ideas, tricks, techniques, and tools is a rough sketch of the character of the entire graphics field. It represents the diversity of the field, containing a wide variety of approaches to solving problems, large and small. As such, it "takes the pulse" of the graphics community, and presents you with ideas that a wide variety of individuals find interesting, useful, and important. I hope that you will find them so as well.

This book can be used in many ways. It can be used as a reference, to find the solution to a specific problem that confronts you. If you are addressing the same problem as one discussed in a particular Gem, you're more than halfway to a solution, particularly if that Gem provides C or C + + code. Many of the ideas in this volume can also be used as a starting point for new work, providing a fresh point of view. However you choose to use this volume, there are many ideas contained herein.

This volume retains the overall structure and organization, mathematical notation, and style of pseudo-code as in the first and second volumes. Some of the individual chapter names have been changed to allow a partitioning that is more appropriate for the current crop of Gems. Every attempt has been made to group similar Gems in the same chapter. Many of the chapter headings appeared in the first two volumes, although some are new. Ray tracing and radiosity have been combined into one chapter, since many of the gems are applicable to either technique. Also, a chapter more generally titled "Rendering" has been added, which contains many algorithms that are applicable to a variety of techniques for making pictures.

As in the second volume, we have taken some of the important sections from the first volume and included them verbatim. These sections are entitled "Mathematical Notation," "Pseudo-Code," and the listings,

"Graphics Gems C Header File," and "2-D and 3-D Vector Library," the last of which was revised in volume two of Graphics Gems.

At the end of most Gems, there are references to similar Gems whether in this volume or the previous ones, identified by volume and page number. This should be helpful in providing background for the Gems, although most of them stand quite well on their own. The only background assumed in most cases is a general knowledge of computer graphics, plus a small amount of skill in mathematics.

The C programming language has been used for most of the program listings in the Appendix, although several of the Gems have C+ + implementations. Both languages are widely used, and the choice of which language to use was left to the individual authors. As in the first two volumes, all of the C and C + + code in this book is in the public domain, and is yours to study, modify, and use. As of this writing, all of the code listings are available via anonymous ftp transfer from the machines 'weedeater.math.yale.edu' (internet address 128.36.23.17), and 'princeton.edu' (internet address 128.112.128.1). 'princeton.edu' is the preferred site. When you connect to either of these machines using ftp, log in as 'anonymous' giving your full e-mail address as the password. Then use the 'cd' command to move to the directory 'pub/Graphics-Gems' on 'weedeater', or the directory 'pub/Graphics/GraphicsGems' on ' princeton'. Code for *Graphics Gems I, II,* and *III* is kept in directories named 'Gems', 'Gemsll', and 'Gemslll', respectively. Download and read the file called 'README' to learn about where the code is kept, and how to report bugs. In addition to the anonymous ftp site, the source listings of the gems are available on the enclosed diskette in either IBM PC format or Apple Macintosh format.

Finally, I'd like to thank all of the people who have helped along the way to make this volume possible. First and foremost, I'd like to thank Andrew Glassner for seeing the need for this type of book and starting the series, and to Jim Arvo for providing the next link in the chain. I'd also like to thank all of the contributors, who really comprise the heart of the book. Certainly, without their cleverness and initiative, this book would not exist. I owe Jenifer Swetland and her assistant Lynne Gagnon a great deal for their magical abilities applied toward making the whole production process go smoothly. Special thanks go to my diligent and thoughtful reviewers—Terry Lindgren, Jim Arvo, Andrew Glassner, Eric Haines, Douglas Voorhies, Devendra Kalra, Ronen Barzel and John Snyder. Without their carefully rendered opinions, my job would have been a lot harder. Finally, thank you to Craig Kolb for providing a safe place to keep the public domain C code.
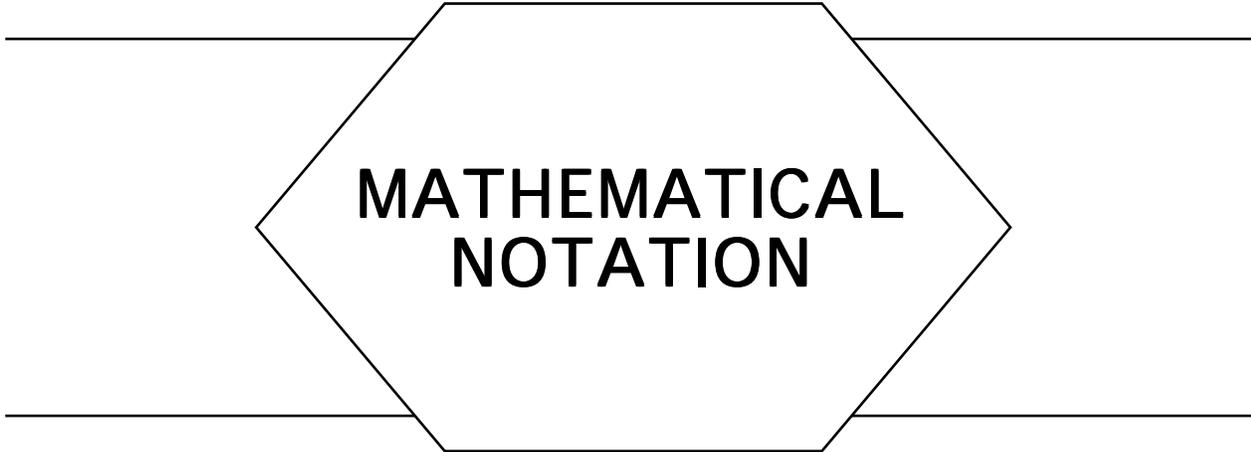
MATHEMATICAL NOTATION

Geometric Objects

0	the number 0, the zero vector, the point $(0, 0)$, the point $(0, 0, 0)$
a, b, c	the real numbers (lower–case italics)
P, Q	points (upper-case italics)
l, m	lines (lower-case bold)
A, B	vectors (upper-case bold)(components A_i)
M	matrix (upper-case bold)
θ, φ	angles (lower-case greek)

Derived Objects

$\mathbf{A}^\perp$	the vector perpendicular to **A** (valid only in 2D, where $\mathbf{A}^\perp = (-A_y, A_x)$
$\mathbf{M}^{-1}$	the inverse of matrix **M**
$\mathbf{M}^T$	the transpose of matrix **M**
$\mathbf{M}^*$	the adjoint of matrix $\mathbf{M}\left(\mathbf{M}^{-1} = \dfrac{\mathbf{M}^*}{\det(\mathbf{M})}\right)$
$\lvert \mathbf{M} \rvert$	determinant of **M**
$\det(\mathbf{M})$	same as above
$\mathbf{M}_{i,j}$	element from row i, column j of matrix M (top-left is $(0, 0)$
$\mathbf{M}_{i,}$	all of row i of matrix **M**

$\mathbf{M}_{\cdot j}$ all of column j of Matrix

$\triangle ABC$ triangle formed by points *A, B, C*

$\angle ABC$ angle formed by points *A, B, C* with vertex at *B*

Basic Operators

$+, -, /, *$ standard math operators

.   the dot (or inner or scalar) product

$\times$ the cross (or outer or vector) product

Basic Expressions and Functions

$\lfloor x \rfloor$ floor of x (largest integer not greater than x)

$\lceil x \rceil$ ceiling of x (smallest integer not smaller than x)

$a \mid b$ modulo arithmetic; remainder of a ÷ b

a mod b   same as above

$B_i^n(t)$ Bernstein polynomial $= \binom{n}{i} t^i (1-t)^{n-i}, \ i = 0 \cdots n$

$\binom{n}{i}$ binomial coefficient $\dfrac{n!}{(n-i)! i!}$
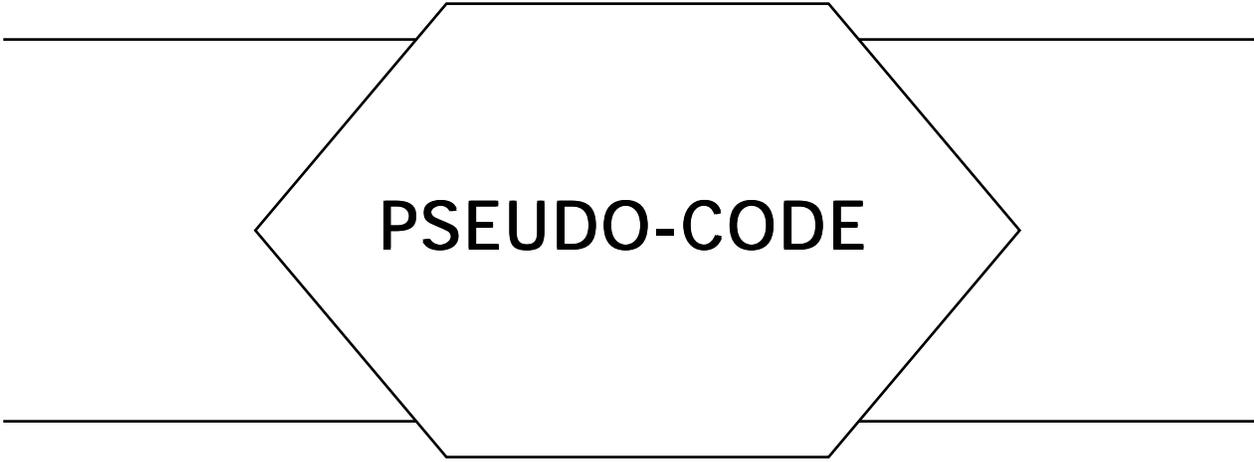
PSEUDO-CODE

Declarations (not required)

name: TYPE ← initialValue;
examples:
π:**real** ← 3.14159;
v: **array** [0..3] **of integer** ← [0, 1, 2, 3];

Primitive Data Types

array [lowerBound..upperBound] **of** TYPE;
boolean
char
integer
real
double
point
vector

matrix3
 equivalent to:
 matrix3: record [array [0..2] of array [0..2] of real;];
 example: m:Matrix3 ← *[[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]];*
 m[2][1] is 8.0
 m[0][2]← 3.3;   assigns 3.3 to upper-right corner of matrix

matrix4

    equivalent to:
    matrix4: record [array [0..3] of array [0..3] of real;];
    example: m: Matrix4 ← [
         [1.0, 2.0, 3.0, 4.0],
         [5.0, 6.0, 7.0, 8.0],
         [9.0, 10.0, 11.0, 12.0],
         [13.0, 14.0, 15.0, 16.0]];
    m[3][1] is 14.0
    m[0][3] ← 3.3; assigns 3.3 to upper-right corner of matrix

Records (Structures)

Record definition:
Box: **record** [
    left, right, top, bottom: **integer**;
    ];

newBox: Box ← **new**[Box];
    dynamically allocate a new instance of Box and return a pointer to it

newBox.left ←10;
    this same notation is appropriate whether newBox is a pointer or structure

Arrays

v: **array** [0..3] **of integer** ← [0, 1, 2, 3]; *v is a four-element array of integers*

v[2] ← 5;   *assign to third element of v*

Comments

A comment may appear anywhere–it is indicated by italics

Blocks

begin
    Statement;
    Statement;
    …
    end;

Conditionals and Selections

if Test
    then Statement;
    [**else** Statement];   *else clause is optional*

result = **select** Item **from**
    instance: Statement;
    endcase: Statement;

Flow Control

for ControlVariable: Type ← InitialExpr, NextExpr **do**
    Statement;
    endloop;

until Test **do**
    Statement;
    endloop;

while Test **do**
    Statement;
    endloop;

loop; *go directly to the next endloop*

exit; *go directly to the first statement after the next endloop*

return[value]   *return value as the result of this function call*

Logical Connectives

or, and, not, xor

Bitwise Operators

bit-or, bit-and, bit-xor

Relations

$=, \neq, >, \geq, <, \leq$

Assignment Symbol

$\leftarrow$

(note: the test for equality is =)

Available Functions

These functions are defined on all data types

min(a, b)	*returns minimum of a and b*
max(a, b)	*returns maximum of a and b*
abs(a)	*returns absolute value of a*
sin(x)	*sin(x)*
cos(x)	*cos(x)*
tan(x)	*tan(x)*
arctan(y)	*arctan(y)*
arctan2(y, x)	*arctan(y/x), defined for all values of x and y*
arcsin(y)	*arcsin(y)*
arccos(y)	*arccos(y)*
rshift(x, b)	*shift x right b bits*
lshift(x, b)	*shift x left b bits*
swap(a, b)	*swap a and b*
lerp(α, l, h)	*linear interpolation: $((1 - \alpha)*l) + (\alpha*h) = l + (\alpha(h - l))$*

clamp(v, l, h)	*return l if v < l, else h if v > h, else v: min(h,max(l,v))*
floor(x) or $\lfloor x \rfloor$	*round x towards 0 to first integer*
ceiling(x) or $\lceil x \rceil$	*round x away from 0 to first integer*
round(x)	*round x to nearest integer, if frac(x) = .5, round towards 0*
frac(x)	*fractional part of x*
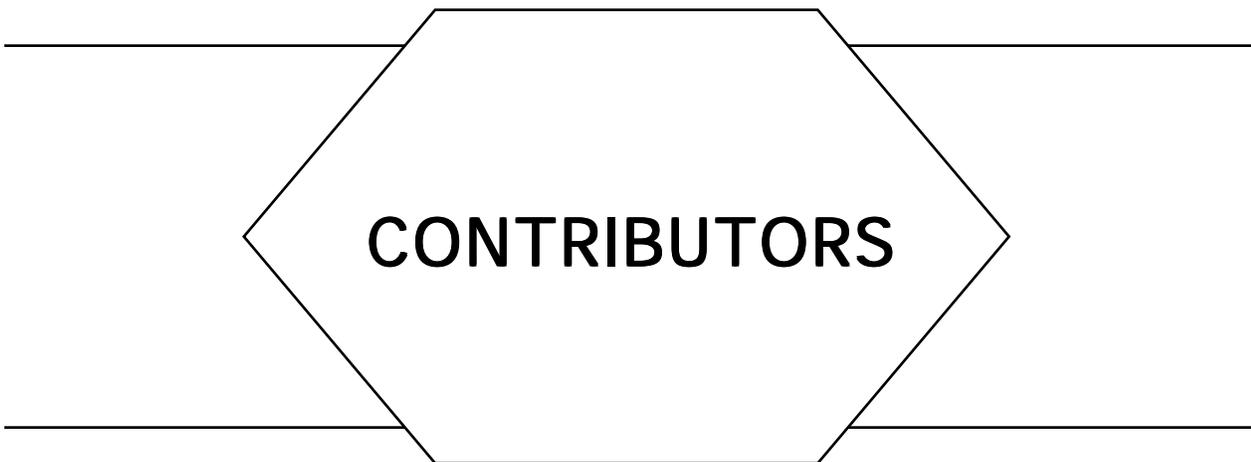
CONTRIBUTORS

Numbers in parentheses indicate pages on which authors' gems begin.

Michael J. Allison (318), *Lawrernce Livermore National Laboratory, P.0. Box 808, Livermore, California 94450*

Franklin Antonio (199), *Qualcomm, Inc., 2765 Cordoba Cove, Del Mar, California 92014*

James Arvo (117, 383), *Program of Computer Graphics, Cornell University, Ithaca, New York 14853*

Didier Badouel (89), *Computer Systems Research Institute, University of Toronto, .10 King's College Road, Toronto, Ontario M5S 1A4, Canada, badouel@dgp.toronto.edu*

A. H. Barr (137), *Computer Graphics Laboratory, California Institute of Technology, Pasadena, California 91125*

Ronen Barzel (34, 374), *Computer Graphics, California Institute of Technology, 350-74, Pasadena, California 91125, ronen@gg.caltech.edu*

Jeffrey C. Beran-Koehn (324), *Department of Computer Science, North Dakota State University, 300 Minard Hall, SU Station, P.0. Box 5075, Fargo, North Dakota 58105-5075, beran-ko@plains,nodak.edu*

Buming Bian (314), *UT System Center for High Performance Computing, 10100 Burnet Rd., Austin, Texas 78758-4497, buming@chpc.utexas.edu*

Dennis Bragg (20), *Graphics Software, Inc., 23428 Deer Run, Bullard, Texas 75757*

Russell C. H. Cheng (343, 349), *School of Mathematics, University of Wales, College of Cardiff, Senghynnydd Road, P.0. Box 915, Cardiff CF2 4AG, United Kingdom, cheng@cardiff.ac.uk*

Norman Chin (219), *Department of Computer Science, Columbia University, New York, New York 10027, nc@cs.columbia.edu*

Joseph M. Cychosz (275, 284), *Purdue University CADLAB, Potter Engineering Center, 520 Evergreen, West Lafayette, Indiana 47907*

Paul Dana (121), *Shadow Graphics, 112 Lily Court, Madison, Alabama 35758, compuserve: 71351,372*

Gershon Elber (133), *Computer Science Department, University of Utah, Salt Lake City, Utah 84112, gershon@gr.utah.edu*

Tim Feldman (29), *Island Graphics Corporation, 4000 Civic Center Drive, San Rafael, California 94903, tim@island.com*

Charles D. Feustel (188), *Department of Mathmatics, Virginia Tech, Blacksburg, Virginia 24061, feustel@mthunx.math.vt.edu*

Kurt Fleischer (362), *California Institute of Technology, Box 350–74, Pasadena, California 91125, kurt@egg.gg.caltech.edu*

Eric Furman (39), *General Dynamics, Electronics Division, San Diego, California 92115*

Mike Fusco (240), *SimGraphic.s Engineering, 1137 Huntington Drive, A1, South Pasadena, California 91030*

Steve Gabriel (374), *10495 W. 85th Place, Arvada, California 80005*

Príamos Georgiades (223, 233), *CrystalGraphics, Inc.; 3110 Patrick Henry Drive, Santa Clara, California 95054*

Andrew Glassner (366, 369), *Xerox PARC, 3333 Coyote Hill Rd., Palo Alto, California 94304*

Ronald N. Goldman (84, 98, 108), *Computer Science Department, Rice University, P.0. Box 1892, Houston, Texas 77251*

Thom Grace (355), *Department of Computer Science, Illinois Institute of Technology, 10 West 31st Street, Chicago, Illinois 60616, grace@iitmax.iit.edu*

Jim Hall (262), *Prime / Computervision, Crosby Dr., Bedford, Massachusetts 01730*

Andrew J. Hanson (51), *Computer Science Department, Indiana University, Bloomington, Indiana 47405, hanson@iuvax.cs.indiana.edu*

Steve Hill (48, 49), *Computing Laboratory, University of Kent, Canterbury, Kent CT2 7NF, United Kingdom, .sah@ukc.ac.uk*

Steve Hollasch (358), *Kubota Pacific Computer, Inc., 2464 El Camino Real, Box #11, Santa Clara, California 95051, hollasch@kpc.com*

Terence Lindgren (262), *Prime / Computervision, Crosby Drive, Bedford, Mas–sachusetts 01730*

Dani Lischinski (256), *Program of Computer Graphic.s, Cornell University, ETC Building, Ithaca, New York 14853, danix@graphics.cornell.edu*

Fernando J. López-López (215), *Physics and Astronomy Deparment, Southwestern College, Chula Vista, California 91810*

Nelson L. Max (318), *Lawrence Livermore National Laboratory, P.O. Box 808, L-301, Livermore, CaIifornia 94550, max2@llnl.gov*

Robert D. Miller (193), *1837 Burrwood Circle, E. Lansing, Michigan 48823*

Tomas Möller (4), *Lund Institute of Technology, sommarstugevägan, 10, 263 71 Höganäs, Sweden, d91tm@efd.lth.se*

Doug Moore (28, 244, 250), *Department of Computer Science, Rice University, P.0. Box 1892, Houston, Texas 77251, dougm@cs.rice.edu*

Jack Morrison (96), *Digital Insight, P.0. Box 2095, Evergreen, Colorado 80439-2095*

F. Kenton Musgrave (288), *Departments of Computer Science and Mathematics, Yale University, Box 2155 Yale Station, New Haven, Connecticut 06520*

Alan W. Paeth (67, 77), *NeuralWare Inc., 4-227 Penn Center W., Pittsburgh, Pennsylvania 15136, awpaeth@alumni.caltech.edu*

Mark J. Pavicic (324), *Department of Computer Science, North Dakota State University, 300 Minard Hall, SU Station, P.0. Box 5075, Fargo, Norlh Dakota 58105-5075, pavicic@plains.nodak.edu*

Jon Rokne (61), *Department of Computer Science The University of Calgary, 2500 University Drive N.W., Calgary, Alberta T2N 1N4, Canada*

Claudio Rosati (173), *IRIS s.r.l., Parco La Selva 151, 03018 PaIiano (FR), Italy*

David Salesin (34, 225, 362), *Program of Computer Graphics, 584 ETC Building Cornell University, Ithaca, New York 14853-3801, dhs@graphics.cornell.edu*

Juan Sanchez (262), *Prime / Computervision, Crosby Dr., Bedford, Massachusetts 01730*

Dale Schumacher (8, 17), *399 Beacon Avenue, St. Paul, Minnesota 55014, dal@syntel.mn.org*

Cary Scofield (36, 383), *Hewlett-Packard Company, 300 Appollo Drive, Chelmsford, Massachusetts 01824, scofield@apollo.hp.com*

Constantin A. Sevici (203), *GTS, 8 Crabapple Lane, Chelmsford, Massachusetts 01824*

Clifford A. Shaffer (188), *Department of Computer Science, Virginia Tech, Blacksburg, Virginia 24061, shaffer@utopus.cs.vt.edu*

Peter Shirley (80, 271), *Department of Computer Science, Indiana University, 101 Lindly Hall, Bloomington, Indiana 47405-4101, shirley@cs.indiana.edu*

Ken Shoemake *(124), 4417 BaItimore Avenue, Philadelphia, Pennsylvania 19104*

Ray Simar (164), *Texas Instruments, P.O. Box 443, Houston, Texas 77251*

John Snyder (374), *California Institute of Technology, Box 350-74, Pasadena, California 91125, snyder@gumby.cs.caltech.edu*

Raman V. Srinivasan (182), *SDRC, 2000 Eastman Drive, Milford, Ohio 45150*

Kelvin Sung (271), *Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 West Springfield Avenue, Urbana, Illinois 61801, ksung@cs.uiuc.edu*

Filippo Tampieri (225, 231, 329), *Program of Computer Graphics, Cornell University, 580 ETCBuilding, Ithaca, New York 14853, fxt@graphics.cornell.edu*

Ben Trumbore (295), *Program of Computer Graphics, Cornell University, 580 Theory Center Building, Ithaca, New York 14853, wbt@graphics.cornell.edu*

Jerry R. Van Aken (164), *Texas Instruments, Mail Stop 712, 12203 Southwest Freeway, Stafford, Texas 77477, jerry@video.sc.ti.com*

Douglas Voorhies (236), *Silicon Graphics Inc., P.0. Box 7311, Mountain View, California 94039-7311, voorhies@sgi.com*

Warren N. Waggenspack, Jr. (275), *IMRLAB, Mechanical Engineering, Louisiana State University, Baton Rouge, Louisiana 70803*

Changyaw Wang (307), *Department of Computer Science, Indiana University, 101 Lindley Hall, Bloomington, Indiana 47405-4101, Wangc@iuvax.cs.indiana.edu*

Len Wanger (240), *SemGraphics Engineering, 1137 Huntington Drive, Al, South Pasadena, California 91030, lrw@egg.gg.caltech.edu*

Joe Warren (23), *Departrnent of Computer Science, Rice University, P.0. Box 1892, Houston Texas 77251, jwarren@cs.rice.edu*

Andrew Woo (338), *Alias Research, Style Division, 110 Richmond Street East, Toronto, Ontario M5C 1P1, Canada*

Xiaolin Wu (301), *Department of Computer Science, Middlesex College, University of Western Ontario, London, Ontario, N6A 5B7 Canada*

Charles A. Wüthrich (89), *Cornputer Systems Research Institute, University of Toronto, 6 King's College Road, Toronto, Ontario M5S 1A1, Canada, wuethri@dgp.toronto.edu*

I
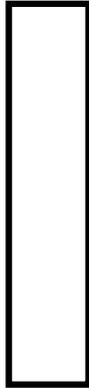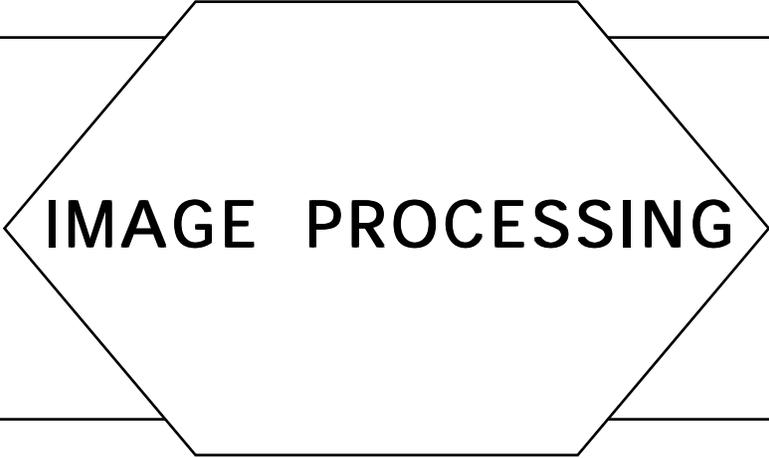
IMAGE
PROCESSING

IMAGE PROCESSING

All of the Gems in this section involve operations performed on images, or two-dimensional arrays of pixels. Often, a graphics programmer may want to change the size, the colors, or other features in an image. The first three Gems describe techniques for stretching or scaling images in various contexts. The first Gem emphasizes speed, while the next two emphasize quality. The fourth Gem describes a method for displaying a full color image using a reduced set of colors.

In some cases, it is useful to combine features from several images. The seventh Gem applies the now-familiar algebra of image composition to black and white bitmaps, or l-bit images. The eighth Gem discusses how to blur two images selectively while combining them in order to simulate camera aperture depth-of-field effects.

Sometimes the desired result is not another image but, in fact, an alternative representation of some of the features in the image. The fifth, sixth, and ninth Gems describe techniques for extracting region boundary information from images.

FAST BITMAP STRETCHING

Tomas Möller
Lund Institute of Technology
Hoganas, Sweden

Introduction

Presented here is an integer algorithm for stretching arbitrary horizontal or vertical lines of a bitmap onto any other arbitrary line. The algorithm can be used in drawing and painting programs where near real-time or real-time performance is required. Examples of application areas are enlarging and diminishing rectangular areas of bitmaps and wrapping rectangular areas onto, for example, circular areas.

The Algorithm

The routine itself is very simple, and most computer-graphics programmers are probably familiar with the Bresenham line drawing algorithm (1965) that it is based upon. In fact, it could be based on any line-drawing algorithm; however, Bresenham was chosen, as it is integer-based and very widespread within the computer graphics community. For those of you who are not familiar with the Bresenham algorithm, pseudo-code follows for line drawing in the first octant.

```
procedure Line(x1, y1, x2, y2)
;draw a line from (x1, y1) to (x2, y2) in first octant
;all variables are integer
begin
    dx ← x2 – x1
    dy ← y2 – y1
    e ← 2*dy – dx
```

```
    for i ← 1, i < = dx, i ← i + 1 do
        WritePixel(x1, y1)              ;display pixel at (x1, y1)
        while e > = 0 do
            y1 ← y1 + 1
            e ← e – 2*dx
        endloop
        x1 ← x1 + 1
        e ← e + 2*dy
    endloop
  end Line;
```

The pseudo-code above works for the second octant as well, but in that case the lines will not be continuous, since *x1* always is incremented by 1. This suits the algorithm very well.

Let us go back to the explanation of the stretching algorithm. Instead of interpreting *x1* and *y1* as a pair of coordinates on a 2-D line, they must be interpreted as 1-D coordinates. *dx* must be interpreted as the length of the destination line, and *dy* as the length of the source line. Using these interpretations, *x1* will be the coordinate on the destination line and *y1* the coordinate on the source line. For each pixel on the destination line, a pixel is selected from the source line. These pixels are selected in a uniform way. See Fig. 1.

If *dx* is greater than *dy,* then the destination line is longer than the source line. Therefore, the source line will be enlarged when plotted on the destination line. On the other hand, if *dy* is greater than *dx,* then the
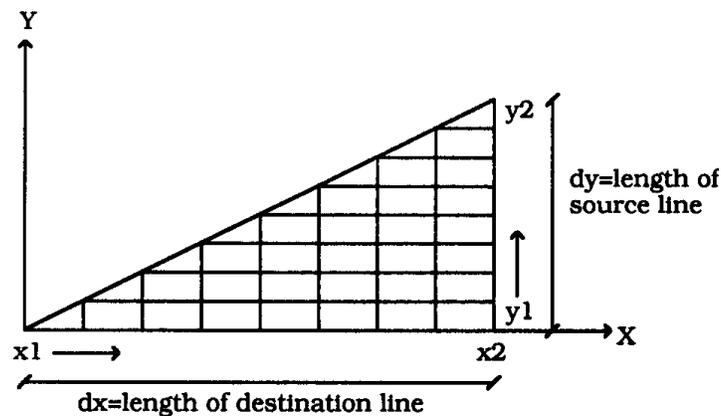
Figure 1.

source line will be diminished. If *dx* is equal to *dy*, the algorithm will yield the same sort of line as the source. Here follows the complete stretcher-algorithm in pseudo-code, rewritten to be able to handle lines where *x2 < x1* and *y2 < y1*.

```
procedure Stretch(x1, y1, x2, y2, yr, yw)
;stretches a source line(y1 to y2) onto the destination line
;(x1 to x2). Both source & destination lines are horizontal
;yr = horizontal line to read pixels from
;yw = horizontal line to write pixels to
;ReadPixel(x, y) returns the color of the pixel at (x, y)
;WritePixel(x, y) writes a pixel at (x, y) with current color
;SetColor(Color) sets the current writing color to Color
begin
     dx ← abs(x2 – x1)
     dy ← abs(y2 – y1)
     sx ← sign(x2 – x1)
     sy ← sign(y2 – y1)
     e ← 2*dy – dx
     dx2 ← 2*dx
     dy ← 2*dy
     for i ← 0, i < = dx, i ← i + 1 do
          color ← ReadPixel(yl, yr)
          SetColor(color)
          WritePixel(x1, yw);
          while e > = 0 do
               y1 ← y1 + sy
               e ← e – dx2
          endloop
          x1 ← x1 + sx
          e ← e + dy
     endloop
end Stretch;
function sign(n):integer;
begin
     if x > 0 return 1
     else return – 1
end sign;
```

The sign function does not need to return zero if *x* is equal to zero, because then either *dx* or *dy* is equal to zero, which means a line with a length equal to 1. Since this algorithm only uses integer arithmetic and does not use multiplications or divisions, it is very efficient and fast.

Another interesting thing about this little program is that it can be used to generate several different shapes of bitmaps. Here follows a list of some things it can be used to render.

Some Projects Using the Bitmap Stretcher

- Rectangular pictures wrapped onto circular or elliptical areas. See source code in appendix for wrapping onto circles.

- Enlarging and diminishing rectangular parts of bitmaps. See source code in appendix.

- Wrapping rectangular parts of bitmaps around parallel trapeziums. For example, a rectangle that is rotated around the *x*- or *y*-axis, and then perspective-transformed, can be used as the destination shape.

Further Work

To improve the algorithm, perhaps an anti-aliasing routine could be added.

See also G1, 147; G1, 166; G3, A.2.

I.2

GENERAL FILTERED IMAGE RESCALING

Dale Schumacher
St. Paul, Minnesota

A raster image can be considered a rectangular grid of samples of a continuous 2-D function $f(x, y)$. These samples are assumed to be the exact value of the continuous function at the given sample point. The ideal procedure for rescaling a raster image involves reconstructing the original continuous function and then resampling that function at a different rate (Pratt, 1991; Foley *et al.,* 1990). Sampling at a higher rate (samples closer together) generates more samples, and thus a larger image. Sampling at a lower rate (samples farther apart) generates fewer samples, and thus a smaller image. Fortunately, we don't really have to reconstruct the entire continuous function, but merely determine the value of the reconstructed function at the points that correspond to the new samples, a much easier task (Smith, 1981). With careful choice of filters, this resampling process can be carried out in two passes, stretching or shrinking the image first horizontally and then vertically (or vice versa) with potentially different scale factors. The two-pass approach has a significantly lower run-time cost, **O**(image_width *image_height*(filter_width + filter_height)), than straightforward 2-D filtering, **O**(image_width*image_height*filter_width*filter_height).

  The process of making an image larger is known by many names, including magnification, stretching, scaling up, interpolation, and upsampling. I will refer to this process as magnification. The process of making an image smaller is also known by many names, including minification, shrinking, scaling down, decimation, and downsampling. I will refer to this process as minification. The processes will be explained in one

$$sinc(x) = \frac{\sin(\pi x)}{\pi x}$$

$$f(x) = 2|x|^3 - 3|x|^2 + 1$$
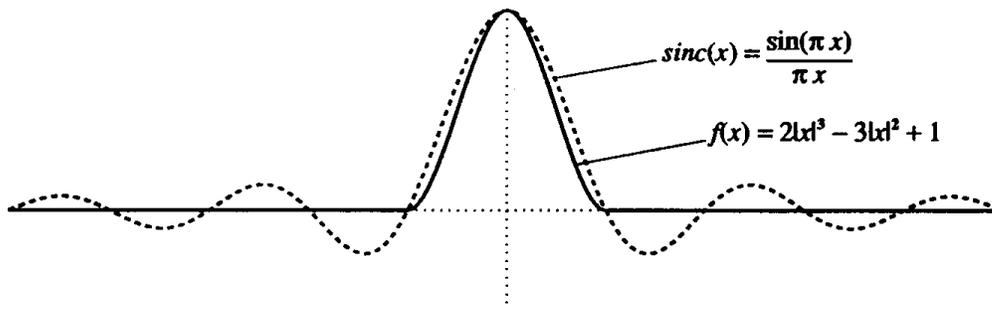
Figure 1.

dimension rather than two, since the scaling is carried out in each axis independently.

In magnification, we determine the contribution each source pixel makes to each destination pixel by application of a filter function. Sampling theory states that the *sinc* function, $f(x) = \sin(\pi x)/\pi x$, is ideal for reconstruction; however, we have a finite sample set and need a filter with a finite support (that is, the area over which the filter is nonzero). The filter I use in this example is a cubic function, $f(x) = 2|x|^3 - 3|x|^2 + 1$, from -1 to $+1$, which covers a unit volume at each sample when applied separably. Figure 1 compares these filter functions. The design of resampling filters is a source of endless debate and is beyond the scope of this gem, but is discussed in many other works (Pratt, 1991; Turkowski, 1990; Mitchell, 1988; Smith, 1982; Oppenheim and Schafer, 1975; Rabiner and Gold, 1975). To apply the filter, we place a copy of our filter function centered around each source pixel, and scaled to the height of that pixel. For each destination pixel, we compute the corresponding location in the source image. We sum the values of the weighted filter functions at this point to determine the value of our destination pixel. Figure 2 illustrates this process.

In minification, the process is similar, but not identical, since we must be concerned with frequency aliasing. Sampling theory defines the Nyquist frequency as the sampling rate that will correctly capture all frequency components in our continuous source signal. The Nyquist frequency is twice the frequency of the highest-frequency component in our source signal. Any frequency component that is higher than half the sampling rate will be sampled incorrectly and will be aliased to a lower frequency.

scaled filter shapes

reconstructed signal

Input Samples

Output Samples
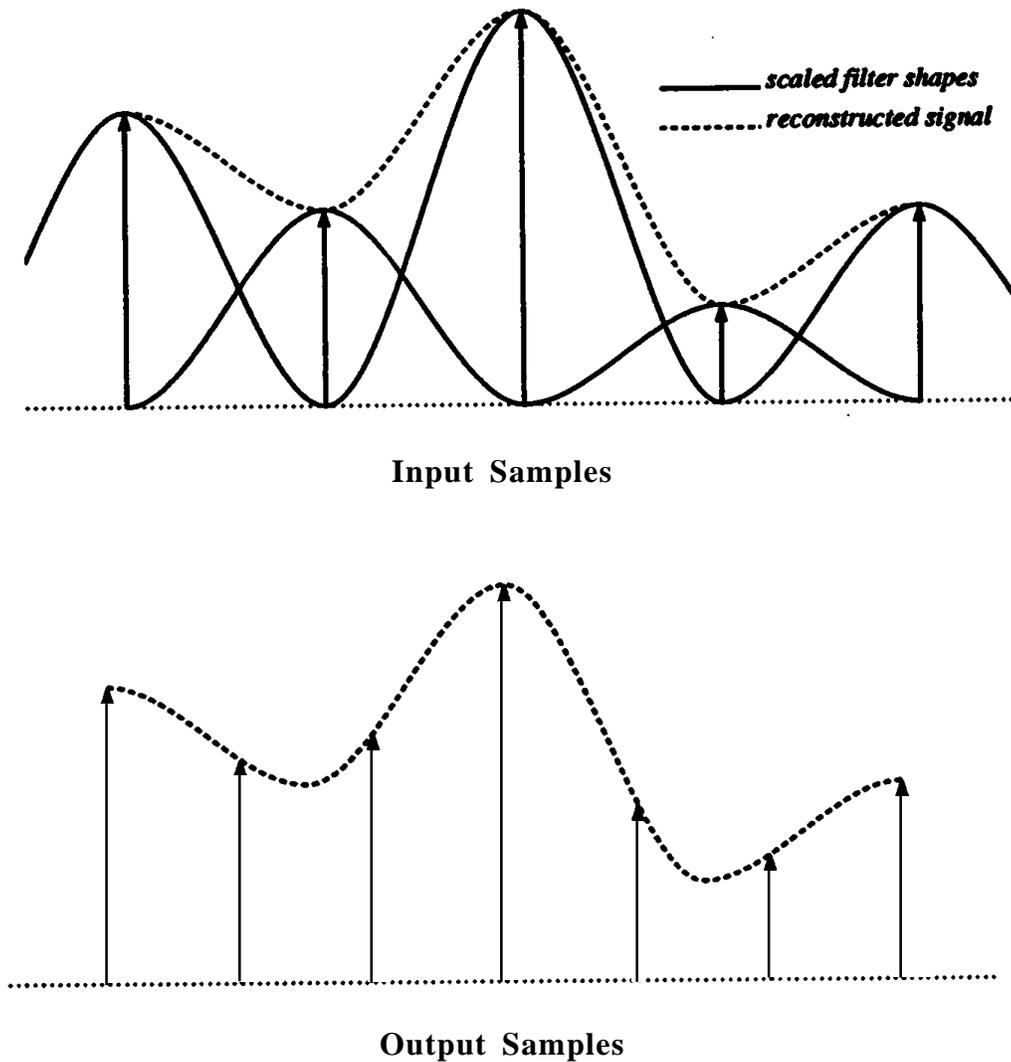
Figure 2.

Therefore, a reconstructed signal will contain only frequency components of half the sampling rate or less. During magnification, we are stretching our reconstructed signal, lowering its component frequencies. However, during minification, we are shrinking our reconstructed signal, raising its component frequencies, and possibly exceeding the Nyquist frequency of our new sampling rate. To create proper samples, we must eliminate all frequency components above the resampling Nyquist frequency. This can be accomplished by stretching the filter function by the image reduction factor. Also, since the filters at each source pixel are wider, the sums will

scaled filter shapes
.... reconstructed signal

Input Samples

Output Samples
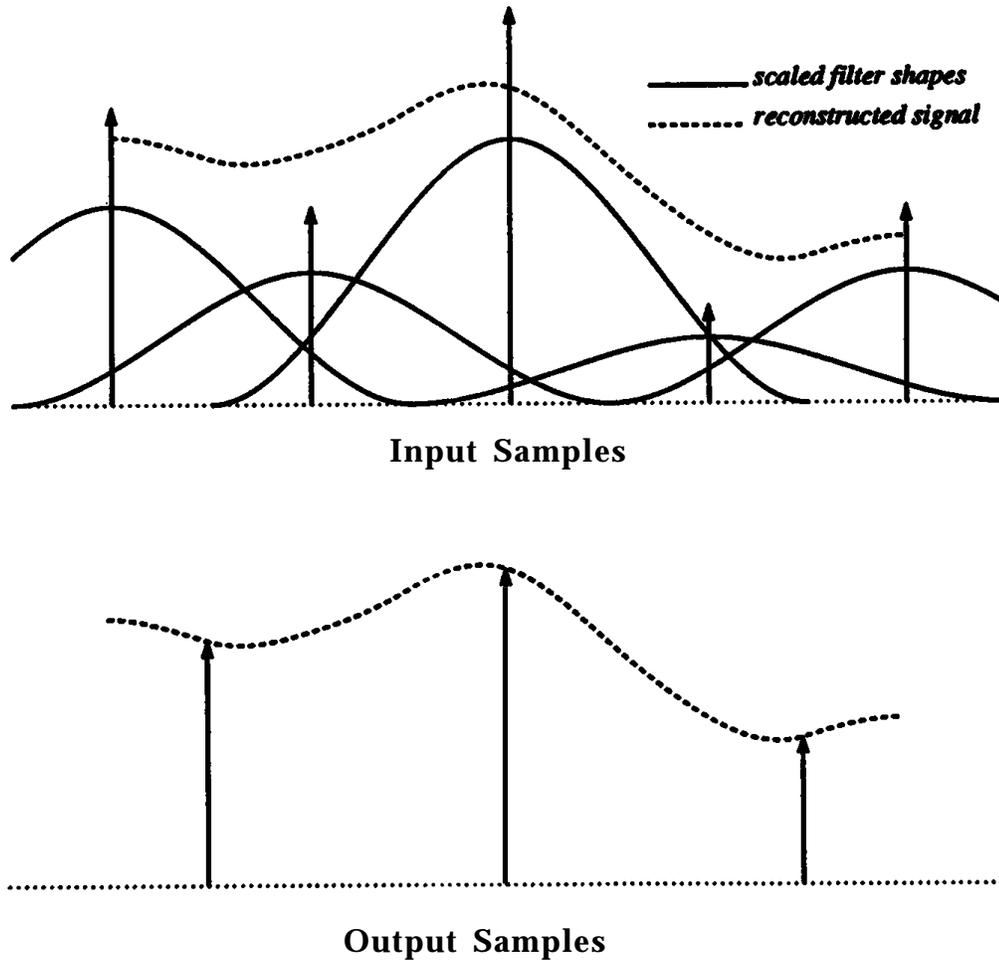
Figure 3.

be proportionally greater and should be divided by the same factor to compensate. Figure 3 illustrates this process.

So far we have only considered the one-dimensional case. We extend this to the two-dimensional case of a typical raster image by scaling first horizontally and then vertically. The further optimization of scaling the smallest destination axis first will not be illustrated here. The filtering operation can lead to a significant number of computations, so we precalculate as much as possible. The scaling process for each row (or column) is identical. The placement and area of the filters is fixed; thus,

we can precalculate the contributors to each destination pixel and the corresponding filter weight. The pseudo-code for calculating the contributors to a destination pixel is as follows:

```
calculate_contributions(destination);
begin
      scale ← dst_size/src_size;
      center ← destination/scale;
      if (scale < 1.0) then begin
          width ← filter_width/scale;
          fscale ← 1.0/scale;
      else
          width ← filter_width;
          fscale ← 1.0;
      end;
      left ← floor(center – width);
      right ← ceiling(center + width);
      for source ← left, source = source + 1, source ≤ right do
          weight ← filter((center – source)/fscale)/fscale;
          add_contributor(destination, source, weight);
          endloop;
      end;
```

After the contributions have been calculated, all the rows (or columns) of the destination image can be processed using the same precalculated filter values. The following pseudo-code shows the application of these values to scale a single destination row.

```
scale_row(destination_row, source_row)
begin
      for i ← 0, i ← i + 1, i < dst_size do
          v ← 0;
          for j ← 0, j ← j + 1, j < contributors[i] do
              s ← contributor[i][j];
              w ← weight_value[i][j];
              v ← v  + (source_row[s]*w);
          endloop;
          destination_row[i] ← v;
      end;
```
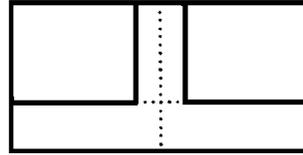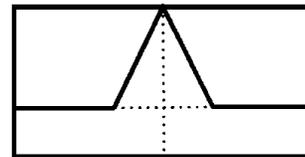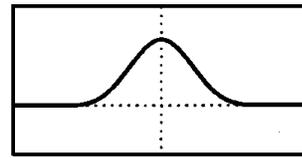
Figure 4.
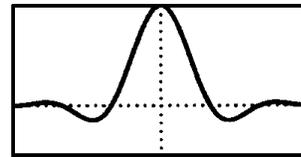
Figure 5.

Figure 6.

Figure 7.

Figure 8.

The same process is then applied to the columns of the image—first precalculating the filter contributions according to the vertical scale factor (which may be different than the horizontal), and then processing columns from the intermediate (horizontally scaled) image to the final destination image.

In the source code provided in the Appendix, a number of filter functions are given, and new functions may be easily added. The zoom() function takes the name of the desired filter and the filter support as parameters. Figures 4 through 8 show the effect of various filters on a sample image, along with the impulse response graphs for each filter function.

The sample images have been scaled up by a factor of 12 in both directions. Figure 4 shows a box filter, which is equivalent to direct replication of pixel values, as it shows considerable tiling or "jaggies." Figure 5 shows the triangle or Bartlett filter, a considerable improvement over the box, still computationally simple, but there are still sharp transition lines. Figure 6 shows a cubic B-spline, which creates no sharp transitions, but its width causes excessive blurring. The triangle and B-spline functions are computed by convolving the box filter with itself

one and three times, respectively. Figure 7 shows the Lanczos$_3$ filter, a *sinc* function damped to zero outside the −3 to +3 range, which shows the excessive "ringing" effect that occurs with the full *sinc* function. Figure 8 shows the Mitchell filter ($B = \frac{1}{3}$, $C = \frac{1}{3}$), a cubic function with no sharp transition, and a good compromise between "ringing" and "blurring" effects.

See also G1, 147; G1, 166; G3, A.1.

OPTIMIZATION OF BITMAP SCALING OPERATIONS

Dale Schumacher
St. Paul Minnesota

This gem describes a series of optimizations for bitmap scaling operations. Instead of giving general scaling algorithms, we take advantage of several application-specific constraints that allow significant reductions in execution time: single-bit per pixel images, known source and destination bitmap sizes, and bit-packed horizontal raster storage and display formats. The example application is the display of fax bitmap images on typical video monitors.

We first assume that we have the source FAX in memory, uncompressed, stored as 8-bit bytes, with the high-order bit of each byte representing the leftmost pixel in a group of eight along a horizontal row. Further, we assume, in choosing our example scale factors, that the resolution of the source FAX is 200 dots-per-inch, in both directions. If the data is in the often-used 200×100 dpi format, we can make it 200×200 dpi by replicating each scanline, a task we can often handle in the decompression phase. Initially we will assume that the data is stored with the bit values for white and black matching those used by the display. A good method of inverting the meaning of 0 and 1 bits will be discussed later. Finally, we assume that the destination bitmap will have the same format as the source.

Since our example image resolution is higher than your typical video monitor, we will only be considering the case of reducing the image, rather than enlarging it. Also, for reasons which will soon be apparent, we work in eighths for scale factors, $\frac{7}{8} = 87.5\%$, $\frac{6}{8} = 75\%$, $\frac{5}{8} = 62.5\%$, $\frac{4}{8} = 50\%$, $\frac{3}{8} = 37.5\%$, $\frac{2}{8} = 25\%$. The general algorithm works as follows. Take a single scanline from the source image. For each byte, use the byte value as an index into a lookup table that gives the reduced bits for a given

input byte. Shift the derived output bits into an accumulator. The number of bits added to the accumulator for each input byte is based on the scale factor (e.g., if we are reducing to $\frac{5}{8}$ scale, we generate five bits output for each eight bits input). When the accumulator has at least eight bits in it, we remove the leftmost eight bits from the accumulator and write them as an output byte into the destination scanline. Any bits remaining at the end of the scanline are shifted into position and flushed out. Many source scanlines can be skipped entirely, again based on the scale factor (e.g., at $\frac{5}{8}$ scale we only process five out of every eight scanlines, skipping three).

Now that the basic algorithm is understood, we can discuss some useful variations and improvements on the process. The heart of the algorithm is the reduction lookup table. If we need to reverse black and white in the final image, one way to do it is to invert the bits stored in the lookup table. Then, instead of 00000000b mapping to 00000b it would map to 11111b. This essentially gives us photometric inversion for free during rescaling. Similarly, we can solve another problem, again for free, by careful creation of the lookup table. if we reduce to $\frac{3}{8}$ scale, we would be seeking three out of each eight bits to output. The simplistic way to do this is shown in Fig. 1a. A better way is to simulate a form of filtering or
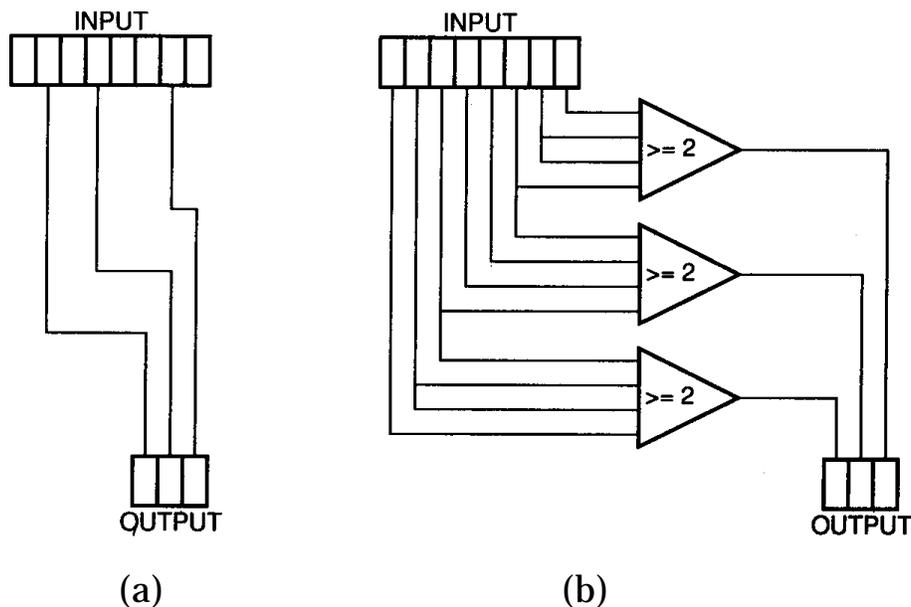
Figure 1.

weighted averaging over the source bits, as shown in Fig. 1b. Since the lookup table can be created at compile-time, the computational cost to create the table using a more complex algorithm is irrelevant to run-time performance. To do proper filtered scaling, we should really be applying the filter across adjacent scanlines, and across byte boundaries as well. Since those operations would carry a high run-time cost, and applying filtering in the limited way that we can show an improvement without additional cost, we do what is cheap. It is better to use filtering, even within these restrictions, than to directly subsample the input as in Fig. 1a.

Any kind of transfer function you'd like can be applied in the same way, within the limitations of an 8-bit span and only black and white as input values. You can even do things like reverse the order of the bits, which can be used in conjunction with a different storage order to either flip the image left-to-right, or rotate it 180 degrees (in case someone fed the image into the scanner upside-down). Expanding the table to 16 bits, which takes 128Kb of memory rather than the 256b used by the 8-bit table, gives even more flexibility. With 16 bits you have a wider span to work with, and can select scale factors in sixteenths rather than eighths, which may allow a better match to your video display size. These techniques, and the sample code given in the appendix, are simply building blocks. Examine the constraints of your own application to find more ways to apply these principles and improve the performance of your code.

See also G1, 147; G1, 166; G2, 57; G2, 84.

A SIMPLE COLOR REDUCTION FILTER

Dennis Bragg
Graphics Software Inc.
Bullard, Texas

Introduction

A simple filter is presented that reduces a 24-bit color raster image to 15 significant bits and eliminates the problem of visible color stepping. The resulting image can be displayed directly on a 16-bit frame buffer or used as the input to a color quantization method for further reduction in the number of colors in the image.

Raster images are often stored as an array of 24-bit pixels with 8 bits allocated to each red, green, or blue (RGB) component. Each RGB component holds one of 256 possible intensity levels. Plate 1 (see color insert) is a 24-bit image that uses 2215 different colors. Note the smooth continuous shading of the colored balls.

Unfortunately, frame buffers that can display 24-bit color images are not always readily available. Color displays that use 8-bit pixels as indexes to a 256-color color map are widely used. A color quantization method (Gervautz and Purgathofer, 1990) is often employed to reduce the number of colors used in a 24-bit image so it can be accurately displayed on an 8-bit device.

Frame buffers that can display 16 bits of color per pixel (five bits per RGB component plus an attribute bit) are also becoming more affordable. The typical solution for displaying a 24-bit image on a 16-bit frame buffer is to mask off the three least significant bits of each RGB component This method reduces the 256 intensity levels available for each color to only 32 levels.

A problem that occurs in color-reducing smooth shaded images is color stepping. A region whose intensity varies continuously from dark to light in the original 24-bit image will often exhibit noticeable intensity level steps when displayed on a 16- or 8-bit frame buffer. In Plate 2 (see color

insert) the image of Plate 1 has been reduced to 256 colors using the color quantization method of Gervautz and Purgathofer. Note the color stepping on the balls due to the limited number of colors available.

This gem solves the color stepping problem by varying the intensity level of each pixel's RGB components by a weighted random amount. The amount of variance is weighted in such a way that the average of any local region of pixels in the resulting image is very close to the actual 24-bit color of the source image.

The resulting image contains 15 significant bits of color per pixel, five bits for each RGB component. The image can be displayed directly on a 16-bit frame buffer or used as the input for a color quantization method to further reduce the number of colors. The resulting image has a somewhat "grainy" appearance, but is much less objectionable than visible color stepping.

The Filter

The filter considers each pixel's RGB component separately. The 256 intensity levels of a component are divided into 32 equal regions. Each region covers eight intensity levels. The first region has an intensity level of zero, the next region has an intensity of eight, and so on.

The intensity of the RGB component will be set to one of these regions. If the component is set to the nearest intensity level, the resulting image would still exhibit color stepping. Instead, the remainder of the intensity divided by 8 (or modulus) is determined. This gives a number ranging from 0 to 7. A random number in the range of 0 to 8 is generated and compared to the remainder. If the remainder is less than or equal to the random number, the component intensity is increased by 8. This has the effect of varying the component in a manner that is random, yet weighted toward the nearest intensity level.

Next, some random noise is added to the component intensity based on a user-supplied noise level. The addition of the noise eliminates any remnants of color stepping that might otherwise be noticeable. Finally, the lower three bits of the component are masked off, reducing the number of significant bits per pixel to 15.

The process produces RGB components that are significantly different from the original 24-bit components. However, the average intensity of the pixel components in any local area of the image is very close to the

average intensity of the original image. In Plate 3 (see color insert), the original 24-bit image was first processed by the filter, then reduced to eight bits per pixel by the same method used in Fig. 2.

Implementation

The filter is implemented with the function rgbvary(). The function requires four arguments: a three-character array of RGB components of the pixel to be processed (rgb), an integer specifying the desired noise level (noise_level), and the x and y location of the pixel (x and y).

The function returns the modified RGB components in the source rgb array. The noise level can vary from zero (no noise) to 8 (loud!). A noise level of 2 has worked well in practice.

The x and y location of the pixel is used by two macros (jitterx and jittery) which generate the random numbers. The jitter macros are based on the jitter function found in GRAPHICS GEMS (Cychosz, 1990). The advantage of using jitter is that it always varies a pixel at a particular x, y location by the same magnitude. This is important when one is color-reducing several frames of an animation. Using a standard random number generator will cause a "snowy" effect as the animation is played. The jitter function eliminates this problem.

The function jitter_init() must be called before any calls to rgbvary() to initialize the look-up tables used by the jitter macros. This procedure uses the standard C function rand() to fill out the tables.

Summary

A filter is presented to reduce a 24-bit image into 15 significant bits per pixel. The procedure eliminates the problem of color stepping at the expense of a slightly grainy appearance. The resulting image can be displayed directly on 16-bit frame buffers or used as input to a color quantization method for further color reduction.

See also G1, 233; G1, 287; G1, 448; G2, 126.

1.4 Plate 1. Original 24-bit color image.

1.4 Plate 2. 256 color image after standard color quantization.

1.4 Plate 3. 256 color image after processing with rgbvary() and standard color quantization.

I.5

COMPACT ISOCONTOURS FROM SAMPLED DATA

Doug Moore and Joe Warren
Rice University
Houston, Texas

Problem

Data in many fields, including medical imaging, seismology and meteorology, arrive as a set of measurements taken over the vertices of a large cubic grid. Techniques for producing a visual representation from a cube of data are important in these fields. Many common visualization techniques treat the data values as sample function values of a continuous function F, and generate, for some c, a piecewise planar approximation to $F(x, y, z) = c$, an isocontour of the function. One of the original *Graphics Gems,* "Defining Surfaces from Sampled Data," surveys several of the best-known techniques for generating isocontours from a data cube (Hall, 1990).

  In this gem, we present an enhancement to all techniques of that type. The enhancement reduces the number of elements of any isocontour approximation and improves the shape of the elements as well. The first improvement typically reduces the size of a representation by about 50%, permitting faster redisplay and reducing memory requirements. The second results in better-quality pictures by avoiding the narrow elements that cause undesirable shading artifacts in many lighting models.

Cube-Based Contouring

Several authors have suggested roughly similar methods that create isocontours for visualization from a cubic data grid. These methods process the data separately on each cube, and use linear interpolation

along the edges of a cube to compute a collection of points lying on the isocontour. In the Marching Cubes algorithm of Lorenson and Cline (Lorenson and Cline, 1987), these intersections are connected to form edges and triangles using a table lookup based on the signs of the values *F(x, y, z) – c* at the vertices of the defining cube.

Unfortunately, that method does not guarantee a continuous contour, since adjacent cubes that share a face with mixed signs may be divided differently (Durst, 1988). Others have suggested an alternative method that disambiguates that case by sampling the function at the center of the ambiguous face (Wyvil *et al.,* 1986). We call methods like these, that compute the vertices of the resulting contour using linear interpolation along edges of the cubic mesh, *edge-based interpolation* methods.

Another problem with edge-based interpolation methods is that the surface meshes they produce can be highly irregular, even for simple trivariate data. These irregularities consist of tiny triangles, produced when the contour passes near a vertex of the cubic mesh, and narrow triangles, produced when the contour passes near an edge of the mesh. In our experience, such triangles can account for up to 50% of the triangles in some surface meshes. These badly shaped elements often degrade the performance of rendering algorithms and finite element analysis applied to the mesh while contributing little to the overall accuracy of the approximation.

Compact Cubes

The contribution of this gem is a general technique for eliminating the problem of nearly degenerate triangles from edge-based interpolation. The idea behind the technique is simple: When a vertex of the mesh lies near the surface, "bend" the mesh a little so that the vertex lies on the surface. The small triangles collapse into points, the narrow ones collapse into edges, and only big, well-shaped triangles are left. The rest of the gem outlines an implementation of this idea; a more detailed explanation is available (Moore and Warren, 1991).

Apply any edge-based interpolation algorithm to the data cube, and in the process, record for each vertex generated along an edge of a cube the point of the cubic grid nearer that vertex. We call that vertex a *satellite* of its nearest gridpoint. If the vertex lies at the midpoint of an edge,
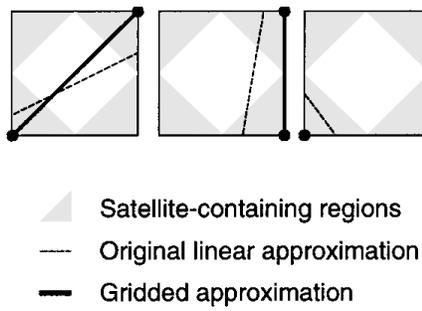
Figure 1. 2D case table for Compact Cubes.
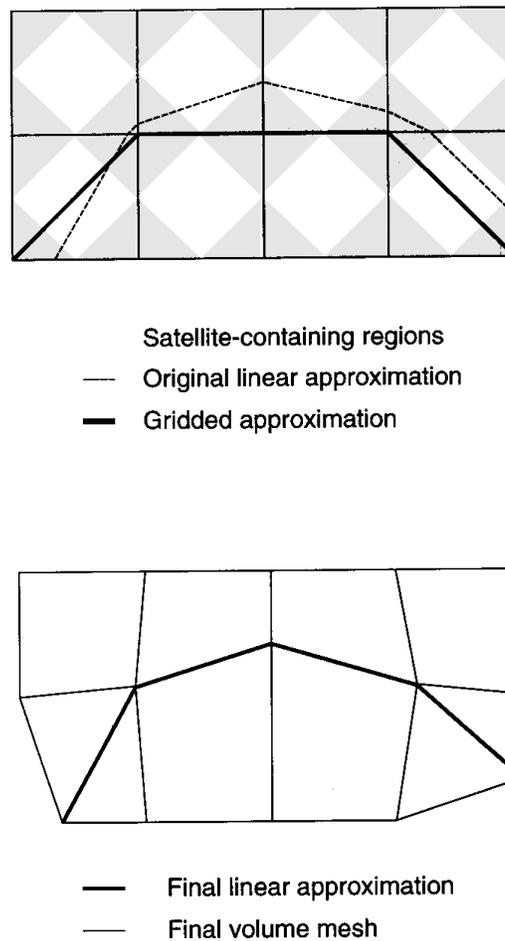
Figure 2. A 2-D example of Compact Cubes.

either endpoint of the edge may be used, as long as all other cubes sharing the edge use the same endpoint. When this phase of the algorithm has completed, you have a triangulation S of the isocontour and a grid point nearest each vertex of the triangulation.

To produce a new, smaller approximation to the isocontour, apply the following procedure:

```
for each triangle T in S do
    if the vertices of T are satellites of distinct gridpoints
        then produce a triangle connecting the gridpoints;
        else T collapses to a vertex or edge so ignore it;
endloop;
for each gridpoint g of the new triangulation do
    displace g to the average position of its satellites;
endloop;
```

The first step of the method defines the topology of a new mesh connecting points of the cubic grid. All the satellites in S of a particular
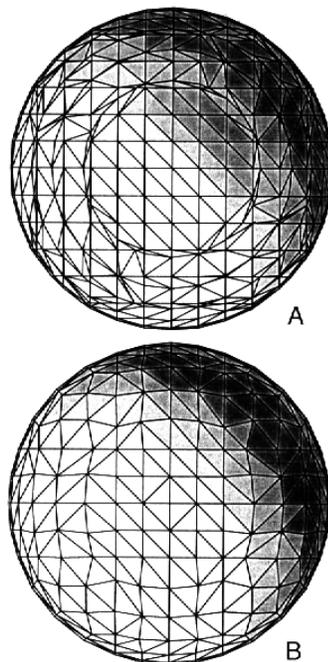
Figure 3. Two approximations to a sphere.

gridpoint are coalesced into a single vertex in the resulting mesh. Thus, small triangles that result when a gridpoint is "chopped off" are collapsed to the gridpoint. Narrow triangles produced when two vertices are very near the same gridpoint are collapsed to make the triangle an edge. Figure 1 illustrates this in two dimensions. This perspective shows that if the original surface mesh is continuous, then the mesh produced in the first step of the algorithm must also be continuous.

In the second step, the vertices of the gridded mesh are displaced to lie on or near the original isocontour. Since each new vertex position is chosen to be at the average position of a small cluster of points lying on the original contour, the new approximation usually diverges only slightly from the original contour.

Figure 2 illustrates this method applied to a two-dimensional mesh. The upper portion illustrates the result of the first step. The lower portion illustrates the output of the second step. The short edges in the upper
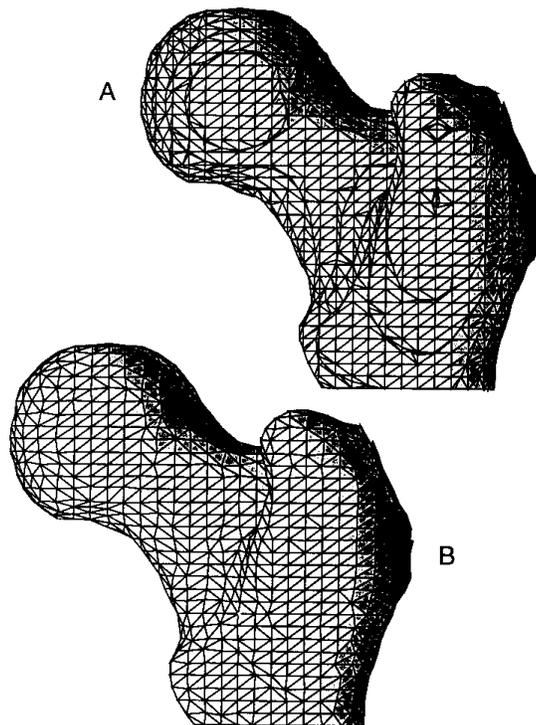
Figure 4. Two approximations to the head of a femur.

portion of the figure have been collapsed to form vertices in the lower portion.

In practice, the method works quite well, reducing the number of triangles by 40% to 60%. Figure 3 shows a sphere generated by Marching Cubes (A) and the same sphere after the application of Compact Cubes (B). Figure 4 shows a human femur, originally presented as CT data, as contoured by Marching Cubes (A) and by Compact Cubes (B). In each example, the number of triangles is reduced by using Compact Cubes, and the shape of the remaining triangles is measurably improved.

As described here, the contours produced by Compact Cubes may have several undesirable features. First, the boundary of the final contour may not lie on the boundary of the defining cubic mesh. Second, two disjoint sheets of the contour passing near a common gridpoint may be fused at that gridpoint. Moore and Warren (1991) describe simple modifications to Compact Cubes that solve each of these problems.

See also G1, 552; G1, 558; G2, 202.

I.6

GENERATING ISOVALUE CONTOURS FROM A PIXMAP

Tim Feldman
Island Graphics Corporation
San Rafael, California

This gem presents an algorithm that follows the edge of a contour in an array of sampled data. It uses Freeman chain encoding to produce a list of vectors that describes the outline of the contour.

Say that you have a terrain map that has been sampled or "digitized" into a rectangular array of gray-scale pixels. Different pixel values correspond to different terrain elevations. This algorithm could be used in producing a "topographic map" that showed the terrain elevations as contour lines. An example program *(contour.c)* that follows one contour is given in the Appendix.

The algorithm is capable of handling contours containing a single sample point, contours surrounding regions of a different elevation, contours that do not form closed curves, and contours that form curves that cross themselves, forming loops. In all cases, it follows the outermost edge of the contour. Given an initial point in an elevation contour in the array, the algorithm finds the edge of the contour. Then it follows the edge in a clockwise direction until it returns to its starting point. Along the way, it describes the path that it follows, using "direction vectors." Each direction vector describes the direction from a pixel on the path to the next pixel on the path. All of the pixels on the path are immediate neighbors. Thus, the vectors may be thought of as the direction part of a traditional two-dimensional vector, with the length part always equal to one pixel. A list of such vectors is known as a "Freeman chain," after Herbert Freeman, its originator (Freeman, 1961). Figure 1 shows the values defining the eight possible directions from a pixel on the path to its neighbors. The pixel array used in this example is in Fig. 2a; Fig. 2b shows the output of the example program. The algorithm found and
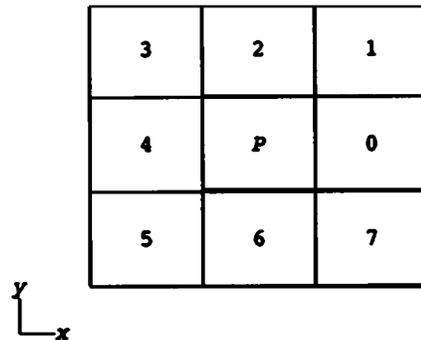
Figure 1. Direction vectors from a sample *P* to its eight neighbors.

followed the edge of the contour with elevation = 200, starting at the sample at *x* = 3, *y* = 2.

The heart of the algorithm lies in knowing how to pick a neighbor in such a way that a clockwise path around the contour is followed. Examine Fig. 2a, and imagine that you are walking around the edge of the contour with elevation = 200, starting from the sample at *x* = 1, *y* = 2. In order to move clockwise around the contour, your first move should be to the sample at *x* = 1, *y* = 3. You are walking on the very edge of your contour; to your left is the dangerous cliff that drops down to a lower elevation. As you follow the contour, note that your heading varies, but that the cliff is always to your left. In choosing your next step, you always try to move ahead and to your left, while not stepping off of the cliff. If you cannot move ahead and to your left, you try a little clockwise of that direction: straight ahead. If that doesn't work, you try a little clockwise of that: ahead and to your right, and so on. If you find yourself on a promontory, your clockwise looking-ahead will cause you to turn around and retrace part of your path. Eventually, you will travel completely around the contour and back to your starting point.

The algorithm works the same way. The *build()* procedure builds a Freeman chain of the directions taken in moving around the contour's edge. *build()* calls the *neighbor()* procedure to get the next neighbor on the path. neighbor() in turn calls *probe()* in looking for that neighbor. The lowest-level procedure is *in_cont(),* which simply tests whether a given sample is in the contour or not. Note that the entire array

7	0	100	100	100	100	100	200	200
6	100	100	100	100	100	200	200	100
5	100	200	200	200	200	200	200	100
4	100	200	255	200	200	100	100	100
3	100	200	250	250	250	200	100	100
2	100	200	200	200	200	100	100	100
1	100	100	200	200	200	200	200	100
0	100	100	100	100	100	100	100	100
y								
└x	0	1	2	3	4	5	6	7

(a)

222000110564577044443

(b)

Figure 2. a) Example sampled elevation data. b) The 22 example vectors outlining the contour with elevation = 200.

of sampled data need not fit into memory at once; *in_cont()* may be modified to access offline storage randomly, if need be.

The *last_dir* variable in *neighbor()* maintains *neighbor()*'s sense of direction. Examine Fig. 3 to see how the *neighbor()* procedure implements the step described above as "try to move ahead and to your left." Say that you arrived at sample P from sample A. Then *last_dir is* 2, and sample C is always outside of the contour, so the first neighbor to probe is D. The direction from P to D is 3; *new_dir* ← *last_dir* + 1.
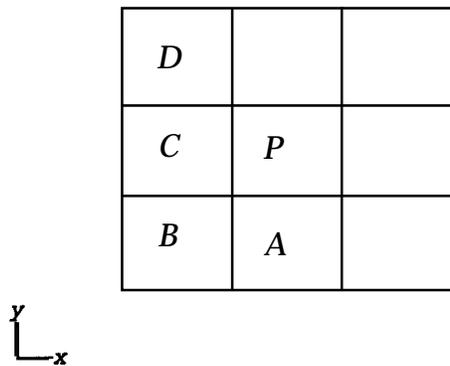
Figure 3. Moving to sample *D* from *P*, via *A* or *B*.

Now say that you arrived at *P* from *B*. *last_dir* is 1, *C* is still outside of the contour, and *D* is still the first neighbor to probe. The direction from *P* to *D* is still 3; *new_dir ← last_dir + 2.*

Note that the cases of arriving at P with *last_dir* equal to 0, 2, 4, or 6 are all congruent (they are simply rotated by 90 degrees). Similarly, the cases for *last_dir* equal to 1, 3, 5, and 7 are all congruent. Therefore, the simple rule that *neighbor()* uses is to set *new_dir* to *last_dir* plus 1 if *last_dir is* even, or plus *2* if *last_dir is* odd. *new_dir* must be kept in the range of 0 through 7, of course, so the addition is modulo 8.

The only remaining subtlety is how to choose the first move properly, so as to start off around the contour in a clockwise direction. This is easily accomplished: The algorithm, when given a start point anywhere in the contour, moves left across the contour until it encounters the edge. This assures that the path begins on the edge. It also assures that the initial arrangement is as shown in Fig. 3: The path begins at a sample P such that the neighbor at *C* is not in the contour and such that the value of *new_dir* should be 3. This implies that the initial value of *last_dir* should be 1. The algorithm sets it up in the *build()* procedure.

The example program was written to demonstrate the idea behind Freeman chains, but it is not storage-efficient. Each member in its linked list takes up an integer and a pointer. But as Freeman pointed out in his original work, only three bits are needed to encode each direction vector. An implementation using three bits would use a large block of memory instead of a linked list, and would have procedures for packing the chain of direction vectors into the block, and for extracting them in sequence.

In order to determine how much memory should be allocated for the block holding all of the direction vectors in a contour, a simplified version of the contour-following algorithm would be used. It would follow the contour and count the number of direction vectors needed to describe the complete path, but it would not store the direction vectors. Once the number of vectors was determined, the memory would be allocated, and the main algorithm would be called to retrace the contour and pack the direction vectors into the memory block.

The preceding approach is more efficient than the example program, but it trades speed for memory space. There is a third approach that still allows the contours to have arbitrary lengths, yet uses memory space efficiently while keeping good speed. It does this by eliminating the pre-scanning step. This is an important consideration for implementations with large data sets, or with data sets that are not held in memory. The approach is to use a simple linked list, as does the example program. However, each member of the list would have a block of allocated memory, instead of an integer. The block would hold many direction vectors, each packed into three bits. Additional blocks of data would be allocated and linked into the list as needed, as the contour was followed. Procedures would be needed for packing and extracting the vectors, and additional housekeeping information would have to be maintained in order to keep everything under control. This technique uses a little space for the pointers in the linked list, but is still much more memory-efficient than the example program. The trade-off with this approach is one that is usually encountered in practical programming: Program complexity would be increased in order to save storage space while maintaining speed.

Finally, some implementations may not need to hold the representation of the contour in memory at all; they may simply write the direction vectors to a sequential-access disk file or to some output device or concurrent process. In that case, the *build*() procedure of the example program would be modified.

See also G3, A.5.

COMPOSITING BLACK-AND-WHITE BITMAPS

David Salesin   Ronen Barzel
Cornell University   *and*   *California Institute of Technology*
Ithaca, New York   *Pasadena, California*

Introduction

A typical bitmap encodes pixels that are black and white. Adding an auxiliary bitmap allows us to represent pixels that are transparent as well. This two-bit representation is useful for black-and-white images that are nonrectangular or that have holes. It also leads to a richer set of operations for combining bitmaps. We encode the three possible pixel values by the Boolean pair (α, β) as follows:

α	β	Meaning
1	0	Black
1	1	White
0	0	Transparent
0	1	Undefined

Compositing Bitmaps

We can combine two pixels $P = (P_\alpha, P_\beta)$ and $Q = (Q_\alpha, Q_\beta)$ into a new pixel $R = (R_\alpha, R_\beta)$ using the *compositing operation* $R \leftarrow P$ **op** Q, as summarized in Table I.

This table is a simplification of the full-color compositing algebra (Porter and Duff, 1984) to a two-bit domain (Salesin and Barzel, 1986). Note that the equations for R_α and R_β in the table are now Boolean formulas: AND is written as multiplication, OR as addition, and XOR as $\oplus$. The Boolean operations can be executed for an entire bitmap at once using a sequence of standard " bitblt" operations. The total number of bitblts required ranges from two to four, depending on the operation.

Table 1. Bitmap Compositing Operations

Operation		Figure	R_α	R_β Description
clear		0	0	Result is transparent
P		P_α	P_β	P only
Q		Q_α	Q_β	Q only
P over Q		$P_\alpha + \overline{Q}_\alpha$	$P_\beta + \overline{P}_\alpha Q_\beta$	P occludes background Q
P in Q		$P_\alpha Q_\alpha$	$Q_\alpha P_\beta$	Q_α acts as a matte for P: P shows only where Q is opaque
P out Q		$P_\alpha \overline{Q}_\alpha$	$\overline{Q}_\alpha P_\beta$	$\overline{Q}_\alpha$ acts as a matte for P: P shows only where Q is transparent
P atop Q		Q_α	$Q_\alpha P_\beta + \overline{P}_\alpha Q_\beta$	$(P$ in $Q) \cup (Q$ out $P)$; Q is both background and matte for P
P xor Q		$P_\alpha \oplus Q_\alpha$	$\overline{Q}_\alpha P_\beta + \overline{P}_\alpha Q_\beta$	$(P$ out $Q) \cup (Q$ out $P)$; P and Q mutually exclude each other

The figures in the table depict the effects of the operations on two sample bitmaps P and Q. In these figures, a grey tone is used to denote areas of transparency in the result.

The **over** operator is useful for placing a nonrectangular black-and-white bitmap on top of an existing image—it is ideal for drawing cursors. The **in** and **out** operators allow one bitmap to act as a matte for another —for example, if P is a " brick" texture and Q a " building," then P **in** Q tiles the building with bricks. The **atop** operator is useful if one bitmap should act as both a matte and background for the other—for example, it allows a small piece of texture to be painted onto an existing bitmap incrementally.

See also G1, 210.

I.8
$2\frac{1}{2}$-D DEPTH-OF-FIELD SIMULATION FOR COMPUTER ANIMATION

Cary Scofield
Hewlett-Packard Company
Chelmsford, Massachusetts

Introduction

Depth of field is defined to be the area enveloping the focal plane in an optical lens system within which objects retain a definitive focal quality. Photographers and cinematographers have long used this aspect of the camera's lens and aperture to direct the viewer's attention to a particular part of the image, away from the areas outside the field of interest. Because of this, it can be advantageous for a computer animation system to include depth-of-field effects in its repertoire.

This gem describes a $2\frac{1}{2}$-D depth-of-field algorithm for simulating change-of-focus in computer-generated animation. This particular algorithm is virtually independent of any hidden-surface removal technique. We stratify a 3-D scene by depth into nonintersecting groups of objects that are rendered independently. The resulting images are filtered to simulate depth of field and then recombined into a single image via a compositing post-process. When gradually changing filters are applied to successive frames of an animation sequence, the effect is to pull the viewer's attention from one depth-plane of the scene to another.

Related Work

Previous attempts at simulating depth-of-field involved the use of a pinhole camera model (Potmesil and Chakravarty, 1982). However, that algorithm did not account for the fact that the surface of a lens provides a

continuum of different views of the environment. Distribution ray tracing (Cook *et al., 1984)* overcomes this deficiency but embeds the technique in the rendering process. Our algorithm can be considered a compromise between these two methods: While we do not integrate shading with depth of field, we do combine surface visibility with depth of field in a restricted way.

The Algorithm

Our approach is essentially a three-stage process:

1. Hidden-surface removal stage: In this first stage, one can either manually or automatically stratify or cluster objects into foreground and background groupings. Each cluster or grouping is rendered separately into its own image with an opacity mask in the alpha channel of each pixel (see Plate 1, color insert). This opacity mask ultimately comes into play during the third stage but it can be modified during the second stage.

2. Filter post-processing stage: This stage uses a convolution mask similar to an exponential low-pass filter to blur the various images to simulate depth-of-field effects. Since our algorithm is unrelated to the focal length of the simulated lens, this allows us to freely manipulate the level of blurriness in an image. However, it is our intent to create a realistic effect, so we choose the level of blurriness carefully. Care also has to be exercised to avoid "vignetting" (Perlin, 1985), a phenomenon that occurs if the filtering algorithm does not compensate for the fact that the areas outside the image boundaries are unknown. This is done by recalculating the weighted filter whenever any portion of the convolution mask is clipped by the image boundary.

3. Compositing stage: Finally, in this stage we follow the algorithms established by Porter and Duff (1984), Duff (1985), and Max and Lerner (1985). The importance of the opacity mask in the first stage comes into play here because it allows us to avoid aliasing artifacts when matting the foreground images onto the background images.

Change-of-Focus Simulation

As stated in the introduction, cinematographers have long used change-of-focus to push or pull the viewer's attention from one part of a scene to another (for example, from the foreground to the background). This "focusing" is a consequence of unfiltered foreground objects (which are "in focus") matted on top of a blurred background (which is "beyond" the focal plane). So, pulling the viewer's attention from the foreground to the background amounts to incrementally translating the "focal plane" of the lens along the optical axis from the foreground to the background over a series of frames. Since we do not use a lens-and-aperture camera model, we must modify the shape of the blur filter from frame to frame. Given that the number of frames over which the change-of-focus simulation will take place is known *a priori*, we can easily perform an interpolation of the filter's "blurriness" ranging from a small value up to a magnitude that gives us the desired maximum degree of image blurriness. This is done for the foreground objects. For the corresponding background images, we use the same series of interpolated values, only in reverse order. When the filtering process is complete, the two separate streams of foreground and background images are then matted together in the last stage of the process to form the final image frames for filming. Plates 2, 3, and 4 (see color insert) are examples of the first, middle, and last frames, respectively, of an animation sequence resulting from this process. As a side note, the results achieved with this process are very similar to results achieved with multiplane camera systems occasionally used in cel animation (Thomas and Johnston, 1981).

Acknowledgment and Historical Note

This gem is a condensation of a much more detailed, but never-published, paper written several years ago in collaboration with James Alvo. As an historical note, the change-of-focus simulation described in this gem was used in the first of the two Apollo Computer ray-traced animation firms (i.e., "Quest").
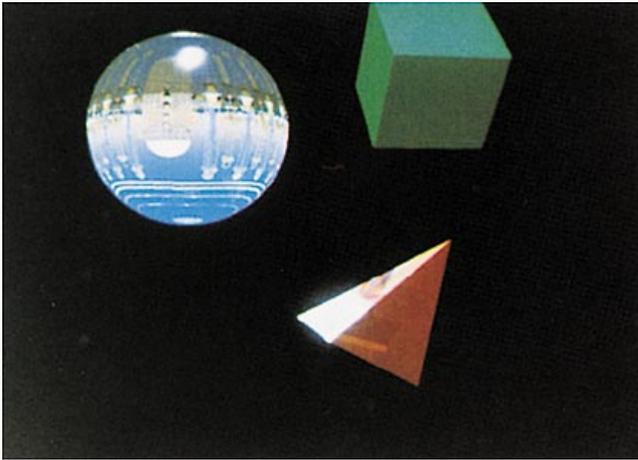
1.8 Plate 1. A cluster of objects from a scene rendered by themselves.

1.8 Plate 2. Composited image of foreground and background objects. Foreground is "in focus."
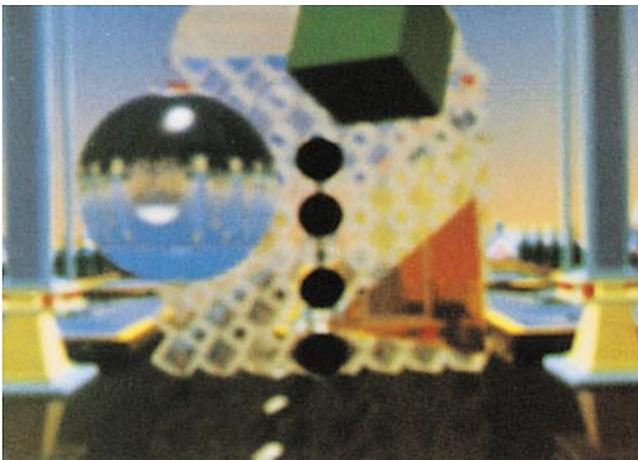
1.8 Plate 3. Composited image. Foreground and background objects are blurred.
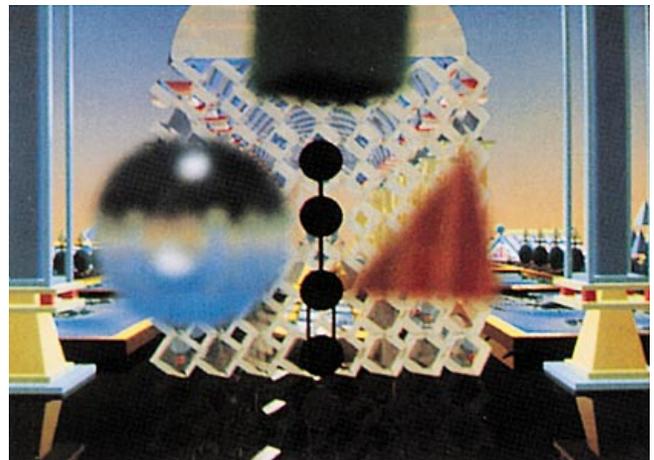
1.8 Plate 4. Composite image. Last frame of animation sequence, with background objects "in focus."

A FAST BOUNDARY
GENERATOR FOR
COMPOSITED REGIONS

Eric Furman
General Dynamics, Electronics Division
San Diego, California

Problem

Finding the outline of multiple areas defined by closed boundaries is a common problem for a number of applications in computer graphics. Two-dimensional coverage for a group of radar sites can be determined by finding the envelope of a set of circles, as shown in Fig. 1. The outline for some set of zoning areas in land use and urban planning is another example. In general, applications of this kind are visualizing unions-of-interest areas. An algorithm is needed to find the composite boundary or envelope of these regions. In other words, we desire to display the outline of a group of two-dimensional regions. Each region is a closed boundary in two dimensions. Common regional primitives are circles, polygons, and ellipses, but any other closed-region outline will work equally well with the technique described. In this gem, we will use the circle as our basic example region. Figures 1 through 3 show the steps of this algorithm. The outline of the circle set has many short connected arcs or scallops.

Other Methods

Several solutions to the multiple-radar-sites problem have taken a direct analytical approach. They work through a list of radar range circles, creating a set of arcs by intersecting each new circle with a set of arcs generated from previous intersections (Bowyer and Woodwark, 1983). Nonintersecting circles must be carried along and intersected with each new circle. Interior/exterior tests are done for each new circle, and the
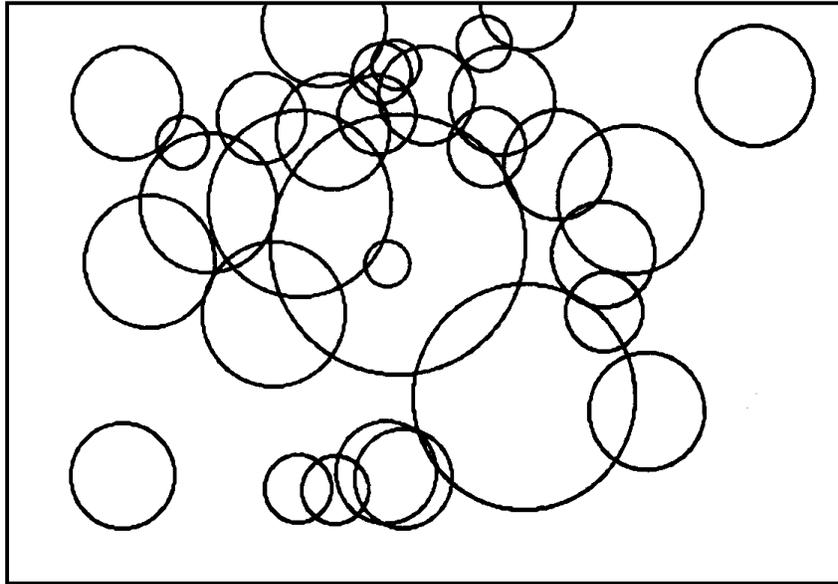
Figure 1. A set of circles shown in a frame buffer.
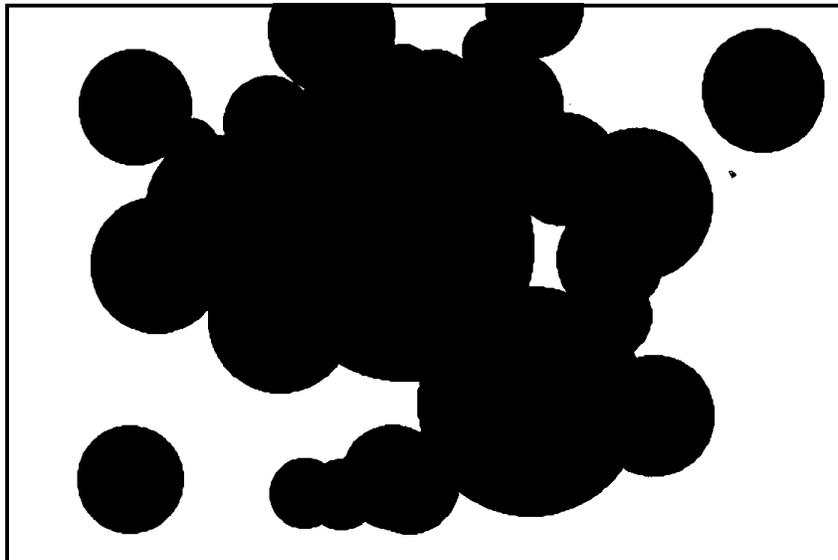
Figure 2. The circles of Fig. 1 after filling.
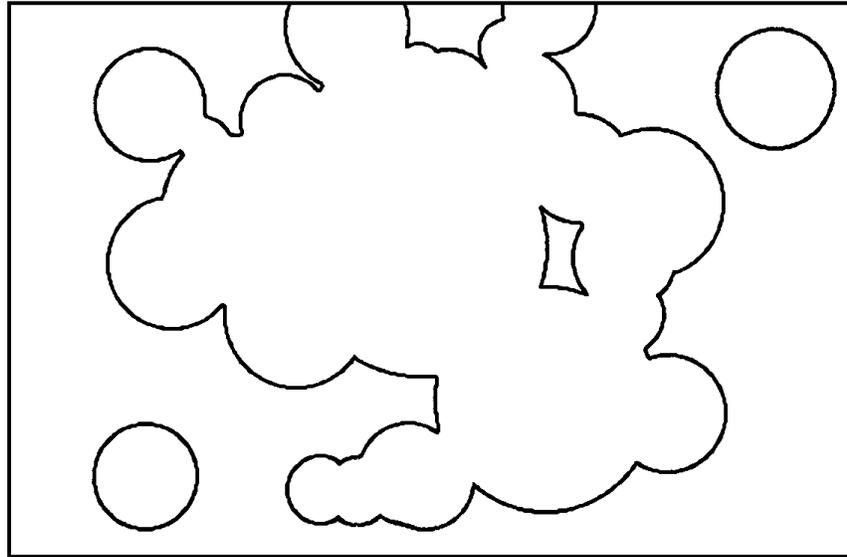
Figure 3. Scallops. The envelope of Fig. 1 circles.

arc set is modified to remove some arc segments and add the new arc. Unfortunately, as the list of circles grows longer, the time to generate the boundary increases with the square of the number of circles. Several improvements have been implemented, such as discarding circles totally contained within other circles and using bounding box tests to limit the number of more costly circle/arc intersection tests.

This direct analytical approach is desirable for creating region-boundary data sets at a resolution much higher than a typical frame buffer can display. However, visualizing the envelope of a set of regions can be done much more quickly.

Fast Boundary Generation

A fast method to generate the boundary for a group of regions blends techniques from computer graphics and image processing. This algorithm consists of two basic steps in the finite resolution of a frame buffer. First, for each closed primitive, draw the region filled and clipped to the frame buffer limits. Even though we wish to display only the envelope of many

closed objects, we both draw and fill each object. Often the drawing and filling can be done in the same step through careful algorithm construction. Second, apply an erosion operator over the full frame buffer to remove all filled interior points. Without filled objects, erosion will not reduce them to their composite boundary. This process is shown in Figs. 1-3, with each circle being independently drawn and filled.

The time for this algorithm's execution is the drawing and filling time for all regions plus the time for a single-pass erosion operation. In our example, this is the time needed to draw/fill all the circles plus the time required to erode them. When one is drawing the individual overlapping objects, it appears some time can be saved by not filling pixels previously filled by another region. Unfortunately, testing to determine if an area has already been filled often takes longer than just writing the data to the frame buffer. The time required grows linearly with the area of all regions filled, for a considerable improvement over the squares growth rate of direct analytical methods.

The example C code in Appendix II to generate filled circles alters the usual midpoint circle-generating algorithm given in computer graphics texts to create filled circles (Foley *et al.,* 1990). Circles are filled by calling the raster_fill function from a direct implementation of the midpoint circle-generating algorithm using second-order partial differences in the fill_circle function.

Erosion of the filled regions to their boundary outline looks at each pixel and its nearest four connected neighbors in the frame buffer. A filled pixel value or color will be erased to the erosion value only when all of its four connected neighbor pixels are also filled with the same value. Erosion to the background value will leave just the boundary pixels drawn in the original fill color. However, eroding to a different color will leave the outline in the original fill value and refill the interior with the new erosion value. Three raster buffers are used for this testing process to avoid replacing pixels in the frame buffer later needed for evaluating other pixels or requiring rendering from one frame buffer to another. In the C code of Appendix II, the raster buffers add a one-pixel border to the frame buffer. Filling this border with the object's original pixel-fill value will leave an open edge where the boundary is clipped at the frame buffer's edge. When the border is set to the background value (zero in the code), a closed edged envelope is drawn where frame-edge clipping

occurs. This process is a fairly simple application of image processing's binary edge detection (Gonzalez and Wintz, 1987) to computer graphics.

Considerations

Although this technique is simple and rapid, like most frame buffer algorithms, it is only accurate to the nearest pixel. The analytical solution's accuracy will be limited by the arithmetic precision used in its calcula-tion.

For circles, a small speed improvement can be made by generating a table of raster span widths for a fixed circle. The diameter should be at least twice the width of the frame buffer's largest dimension to stay within the Nyquist interval. The table can then be used for proportional span-width lookups of any desired circle size. A table lookup eliminates further circle generation, but one must still do the clipping and filling operations. Similar tabling benefits may be gained for any directly scalable and unrotated region.

To assist the user in verifying the functions proper implementation, the C code includes a simple program to test the filling and eroding routines. A tiny pseudo-frame buffer is used with the upper left corner displayed in ASCII using simple "printf" statements.

II
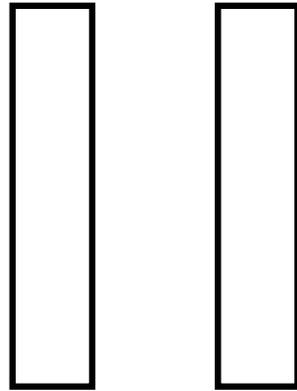
NUMERICAL AND PROGRAMMING TECHNIQUES

II

NUMERICAL AND PROGRAMMING TECHNIQUES

The computer graphics field is mined with complicated mathematics, and graphics programs are often filled with computation-intensive operations. Techniques and tricks for simplified calculations or useful approximations are always welcome. This section contains Gems that add to the bag of tricks for those programmers who like to "sweat the details."

The first Gem describes a fast approximation to the IEEE standard square root operation, and improves upon a technique presented in a previous Gem. The second Gem describes a wrapper to place around the commonly known UNIX(tm) memory allocator "malloc()" in order to improve its usefulness and predictability. The third Gem explains how to think of 3-D rotations being controlled by a track ball, and provides the group theory mathematics behind it. The fourth Gem gives a brief introduction to interval arithmetic and how to use it in computer graphics.

The fifth Gem discusses efficiency issues related to the often-used techniques of cyclic permutations of two, three, or more numbers. The sixth Gem discusses how to select colors for highlighting or selecting of image features, and presents an analogy to the space of Rubik's cube! The seventh Gem deals with producing sets of random points with various distributions, uniform and otherwise. These techniques can be quite useful for distribution ray tracing and other Monte Carlo methods. The last two Gems take some concepts that are often used in two and three dimensions and extend them to higher dimensional spaces.

IEEE FAST SQUARE ROOT

Steve Hill
University of Kent
Canterbury, Kent, United Kingdom

This gem is a reimplementation of the fast square root algorithm pre-sented by Paul Lalonde and Robert Dawson in Graphics Gems I. Refer to this for further details of the operation of the routines.

In my implementation, I have added an extra routine that allows the table of square roots to be dumped as C source. This file can be separately compiled to eliminate the necessity to create the table at run-time.

The new routine uses IEEE double-precision floating-point format. I have included a number of useful #defines to make the program more accessible. Note that on some architectures the order of the words is reversed. The constant MOST_SIG_OFFSET can be set to either one or zero to allow for this fact.

The table size can be adjusted by changing the constant SQRT_TAB_ SIZE. It must be a power of four. The constant MANT_SHIFTS has to be adjusted accordingly—if you quadruple the table size, then subtract two from 3MANT_SHIFTS.

See also G1, 403; G1, 424; G2, 387.

II.2

A SIMPLE FAST MEMORY ALLOCATOR

Steve Hill
University of Kent
Canterbury, Kent, United Kingdom

This Gem describes a simple memory allocation package which can be used in place of the traditional malloc() library function. The package maintains a linked list of memory blocks from which memory is allocated in a sequential fashion. If a block is exhausted, then memory is allocated from the next block. In the case that the next block is NULL, a new block is allocated using malloc().

We call the list of memory blocks a *pool.* A pool may be freed in its entirety, or may be reset. In the former case, the library function free() is used to return all the memory allocated for the pool to the system. In the latter case, no memory is freed, but the high-water mark of the pool is reset. This allows all the data allocated in the pool to be discarded in one operation with virtually no overhead. The memory in the pool is then ready for reuse and will not have to be re-allocated.

The package allows the programmer to create multiple pools, and to switch between them.

Some advantages of this scheme are:

- Memory allocation is fast.

- Data is likely to have greater locality.

- We no longer require a *free* routine for each data structure.

- Resetting the pool is extremely simple. This might replace many calls to the free() library routine.

- Space leaks are less likely.

The principal disadvantage is:

• Individual structures cannot be freed. This might lead to greater program residency.

The package has been used successfully in a ray tracing program. Two pools were used. The first pool holds the permanent data created whilst reading the model file. The second pool is for ephemeral data created during the rendering process. This pool is reset after each pixel has been calculated.

Incorporation of the package had three significant effects. Firstly, the program ran faster. The speed-up was not spectacular, but the program spends most of its time calculating intersections, not allocating memory. Secondly, the code for many operations became simpler. This was due to the elimination of calls to free memory. Finally, all space leaks were eradicated. The program had been worked on by a number of people, and in some cases calls to the appropriate memory de-allocation functions had been forgotten. Using the package eliminated the need for these calls; hence, the space leaks were also eliminated.

II.3

THE ROLLING BALL

Andrew J. Hanson
Indiana University
Bloomington, Indiana

Interactive graphics systems often need techniques that allow the user to rotate graphical objects freely in three-dimensional space using commonly available two-dimensional input devices such as a mouse. Achieving this goal is hampered by the fact that there is no single natural mapping from the two parameters of the input device to the three-parameter space of orientations.

Here we introduce the *rolling-ball* method for mouse-driven three-dimensional orientation control, along with some of its interesting extensions to other scientific visualization problems. This technique exploits a continuous two-dimensional motion (modeled after that of a ball rolling without slipping on a flat table) to reach any arbitrary three-dimensional orientation. Unlike a variety of other methods, the rolling-ball approach has only a single state and is completely *context-free:* One can turn off the mouse cursor and ignore the history or evolving state of the motion, and yet still know *exactly* what the effect of the next incremental mouse motion will be. For applications that benefit from the impression of direct manipulation, this property is very attractive.

It is clear that a mouse can control rotations about two axes (the x and y directions in Fig. 1). Surprisingly, the rolling ball also naturally includes the capability of inducing clockwise and counterclockwise rotations with respect to the screen perpendicular (the z axis in Fig. 1). According to a fundamental but counterintuitive property of the group theory of spatial rotations, moving a rolling-ball controller in small clockwise circles *must* produce small counterclockwise rotations of the ball, and vice versa. This explains why an apparently impossible third degree of rotational
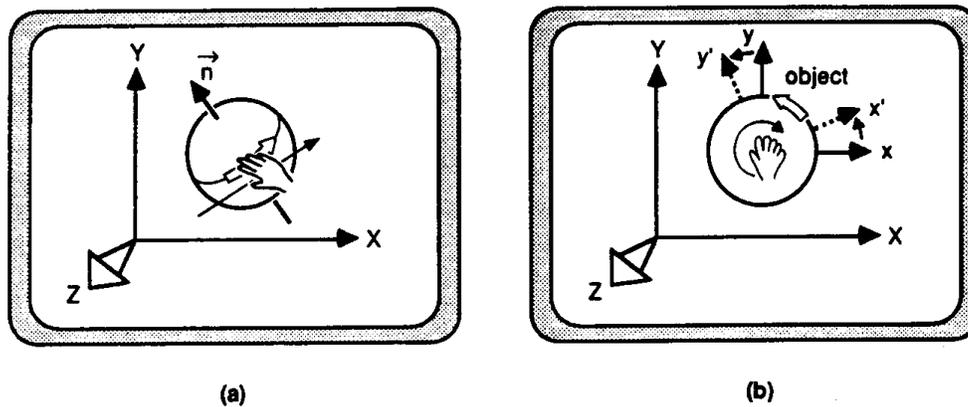
Figure 1. The two basic techniques used in the rolling-ball algorithm to carry out arbitrary spatial rotations of a graphics object. a) Moving the hand with the mouse in the direction indicated by the black arrow causes the object to rotate about the axis $\vec{n}$ lying in the screen plane, i.e., to rotate the equator in the direction of the hollow arrow. b) Moving the hand in small circles causes the object to rotate about the normal to the screen plane in the direction *opposite* to the hand motion, again rotating the equator in the direction of the hollow arrow.

freedom can indeed be generated using a context-free two-degree-of-freedom input device.

The mathematical form of the rolling-ball algorithm given below is in fact included as a part of one of the algorithms studied in the extensive investigation of orientation-control methods by Chen *et al.* (1988); our approach exploits and extends the properties of the algorithm in ways that were not treated by Chen *et al.* (1988). The rolling ball should be understood as a novel, context-free *method* for taking advantage of a known rotation algorithm that is typically used in a context-dependent fashion.

The following treatment consists of two main parts. The first tells how to use the rolling-ball method for three-dimensional orientation control and how to implement it in an interactive graphics system. The second part describes how the rolling ball approach can be extended to other groups of transformations that are extremely important in scientific visualization; the rolling-ball method is then seen to be a fascinating tool in its own right for visualizing the properties of transformation groups.

The Rolling-Ball Algorithm

Using the Method

To understand the basic principle of the method, consider a ball lying on a table beneath the horizontal palm of your hand.

Rotations of the ball about any single axis parallel to the table top are executed by moving the hand horizontally in the direction perpendicular to the axis, thus rolling the ball about that axis. Observe that no single motion of this class produces rotations about the vertical axis, the axis perpendicular to the palm of the hand.

However, if you place your hand flat on top of a ball and move your hand in *small horizontal circles,* the ball will actually rotate about the vertical axis in the *opposite* direction.

The *rolling-ball algorithm* for controlling spatial orientation is implemented simply by treating the orientation of the graphical object to be rotated as the orientation of the ball itself, while using the mouse (or similar two-dimensional input device) to emulate the actions of the palm of the hand.

By executing the indicated motions of the mouse (or hand), one can use the rolling-ball algorithm to achieve the following effects on a displayed graphical object:

- *Rotation about a horizontal screen line,* or the *x*-axis, is carried out by moving the mouse forward or backward relative to the viewer.

- *Rotation about a vertical screen line,* or the *y*-axis, is carried out by moving the mouse to the left or right.

- *Rotation about a diagonal line* lying in the screen plane, whose direction we denote by the vector $\bar{n}$, is carried out by moving the mouse perpendicular to $\bar{n}$, as though the palm were rotating a cylinder or ball about the axis $\bar{n}$.

- *Small clockwise rotations about the perpendicular to the screen,* or the *z*-axis, are carried out by moving the mouse in small, counter-clockwise circles. More pronounced rotations are achieved by using larger circular motions.

- *Small counterclockwise rotations about the perpendicular to the screen* are carried out by moving the mouse in small, clockwise circles.

- *Large rotations about the perpendicular to the screen* are carried out by rotating the object 90° in any direction, rotating the desired amount about the original screen-perpendicular axis (which now lies in the screen plane), and then rotating 90° back to restore the orientation of the original screen-perpendicular axis. This action is essentially a large oblong motion, contrasting with the small circular motions used for small rotations about the screen-perpendicular.

The two most basic actions, rotation about an axis $\bar{n}$ in the screen plane and rotation about the screen-perpendicular axis, are summarized in Fig. 1.

  The position of the input-device cursor is irrelevant for the rolling-ball algorithm, and it is normally made invisible to the user during rotation operations. Only the difference between the previous and current device position is needed by the computation, so it is often desirable to warp the mouse to the center of the screen after each motion to prevent it from leaving the interactive window. Thus the method is truly context-free and is well-suited to user interfaces that emphasize direct manipulation.

Implementation

The rolling-ball algorithm is implemented by taking a given incremental input-device motion to define a vector with components *(dx, dy)* in a right-handed screen coordinate system. The right-handed axis of rotation $\bar{n}$ is then defined as the following unit vector lying in the screen plane and oriented perpendicular to the input-device motion:

$$n_x = \frac{-dy}{dr}, \qquad n_y = \frac{+dx}{dr}, \qquad n_z = 0, \tag{1}$$

where we define the input-device displacement $dr = (dx^2 + dy^2)^{1/2}$.

  Next, we introduce the single free parameter of the algorithm, the effective rolling ball radius R, which determines the sensitivity of the rotation angle to the displacement dr; if dr is a few pixels, a value of R around 100 is appropriate. We choose the rotation angle to be $\theta =$

$\arctan(dr/R) \approx (dr/R)$, so that

$$\cos\theta = \frac{R}{\left(R^2 + dr^2\right)^{1/2}},$$

$$\sin\theta = \frac{dr}{\left(R^2 + dr^2\right)^{1/2}}. \tag{2}$$

The general form of the matrix for a rotation by an angle θ about the axis $\vec{n}$, where $\vec{n} \cdot \vec{n} = 1$, is (see, for example, "Matrix Techniques" by M. Pique in *Graphics Gems I,* p. 446 [Glassner, 1990]):

$$\begin{vmatrix} \cos\theta + \left(n_x\right)^2(1 - \cos\theta) & n_y n_x(1 - \cos\theta) - n_z\sin\theta & n_x n_z(1 - \cos\theta) + n_y\sin\theta \\ n_y n_x(1 - \cos\theta) + n_z\sin\theta & \cos\theta + \left(n_y\right)^2(1 - \cos\theta) & n_y n_z(1 - \cos\theta) - n_x\sin\theta \\ n_z n_x(1 - \cos\theta) - n_y\sin\theta & n_z n_y(1 - \cos\theta) + n_x\sin\theta & \cos\theta + \left(n_z\right)^2(1 - \cos\theta) \end{vmatrix}$$

$$\tag{3}$$

When we substitute into Eq. (3) the values of $\vec{n}$ from Eq. (1), we get the rolling-ball rotation matrix

$$\begin{vmatrix} \cos\theta + \left(dy/dr\right)^2(1 - \cos\theta) & -(dx/dr)(dy/dr)(1 - \cos\theta) & +(dx/dr)\sin\theta \\ -(dx/dr)(dy/dr)(1 - \cos\theta) & \cos\theta + \left(dy/dr\right)^2(1 - \cos\theta) & +(dy/dr)\sin\theta \\ -(dx/dr)\sin\theta & -(dy/dr)\sin\theta & \cos\theta \end{vmatrix},$$

$$\tag{4}$$

where the values of the trigonometric functions are given by Eq. (2).
  We observe the following:

- All vectors must be translated to the desired center of rotation before applying Eq. (4).

- The rotation must be performed in a *single step* as shown in Eq. (4). Carrying out the rotation as a sequence, e.g., first about the *x*-axis and then about the *y*-axis, will give a completely different result (although, for subtle reasons, the difference may be nearly unobservable).

- Changing the overall sign of $\vec{n}$ produces a rotation of the viewpoint around the object instead of a rotation of the object within the view. Small clockwise hand motions will produce small *clockwise* rotations of the viewpoint, but an *object* at the center of the view will continue to rotate *counterclockwise.* This phenomenon derives from a sign difference in the group-theoretical description of body-fixed rotations versus space-fixed rotations (Whittaker, 1944).

Extensions of the Rolling-Ball Method

When we analyze the group-theoretical context of the rolling-ball method, a variety of related applications immediately suggest themselves. Here we summarize the basic group theory involved for ordinary rotations, as well as several extensions that are straightforward to implement. These techniques are useful for a number of scientific visualization applications, including building intuition about groups in general. The reader who has no interest in group theory but just wants to know how to implement and use the algorithm need not read further.

Group Theory of Infinitesimal Rotations

The underlying group theory (Edmonds, 1957) involved in the behavior of the rolling ball can be summarized as follows: If we define L_i, $i = \{x, y, z\}$, to be the infinitesimal generators of the rotation group $O(3)$ with a right-handed convention for positive rotations, then we have the commutation relations

$$[L_y, L_z] = -L_x, \tag{5}$$

$$[L_z, L_x] = -L_y, \tag{6}$$

$$[L_x, L_y] = -L_z, \tag{7}$$

where we used the definition $[A, B] = AB - BA.$ These infinitesimal generators can be represented as matrices or as differential operators of

the form

$$L_x = y\frac{\partial}{\partial z} - z\frac{\partial}{\partial y}$$

and its cyclic permutations. The minus sign in Eqs. (5–7) is not arbitrary, but is determined by our convention that L_i rotate a vector about the ith axis using the right-hand rule. This minus sign is directly responsible for the observed counterrotation and is an inevitable consequence of the properties of the rotation group.

Quaternion rotations, 2 × 2 Matrices, and SU(2) Spinors

The rolling-ball transformation works to define quaternion rotations (see, for example, Shoemake, 1985, or "Using Quaternions" by P.-G. Maillot in *Graphics Gems I,* p. 498 [Glassner, 1990]) even more naturally than ordinary spatial rotations. This follows from the fact that the quaternion formulation is equivalent to the more standard 2×2 matrix notation for the group $SU(2)$, which is the double covering of the usual rotation group $O(3)$ (Edmonds, 1957). (Even though these two groups correspond to entirely different topological spaces, their infinitesimal properties exploited by the rolling ball are identical.)

To carry out $SU(2)$ rotations using the rolling ball, we replace Eq. (3) by

$$U = I_2 \cos \frac{\theta}{2} - i\vec{n} \cdot \vec{\sigma} \sin \frac{\theta}{2}, \tag{8}$$

where I_2 is the 2×2 unit matrix, and $\vec{\sigma}$ denotes the 2×2 matrix basis for $SU(2)$ obeying the cyclic relations $\sigma_x^2 = 1$, $\sigma_x\sigma_y = i\sigma_z$. This is equivalent to a quaternion-based transformation with $(c, u) = (\cos(\theta/2), \vec{n} \sin (\theta/2))$. Note how much more simply Eq. (8) incorporates the fundamental parameters of the rolling ball than the full matrix, Eq. (3).

Changing the overall sign of $\vec{n}$ produces a rotation of the viewpoint around the object instead of a rotation of the object within the view.

The elements of the matrix Eq. (8) may be used to compute an ordinary vector rotation matrix from Eq. (3) as desired, and may also be used directly as 2×2 matrices to rotate *spinors* (Edmonds, 1957), which are the most fundamental objects upon which the rotation group can act.

Four Euclidean Dimensions

In four Euclidean dimensions, there are six degrees of rotational freedom from the group $O(4)$ instead of the three that are present in three-dimensional space due to $O(3)$.

The six $O(4)$ rotation operators $L_{\mu\nu}$, μ, $\nu = \{1, 2, 3, 4\}$, $L_{\mu\nu} = -L_{\mu\nu}$, can be decomposed into $O(3) \times O(3)$ by defining the following combinations:

$$L_i^{\pm} = \frac{1}{2}\left(\frac{1}{2}\epsilon_{ijk}L_{jk} \pm L_{4i}\right). \tag{9}$$

Here ϵ_{ijk} is the totally antisymmetric tensor in three dimensions, and we use the convention that repeated roman indices are summed from 1 to 3. Each of these combinations obeys *independent* $0(3)$ commutation relations,

$$\left[L_i^{\pm},\ L_j^{\pm}\right] = -\epsilon_{ijk}L_k^{\pm}, \quad \left[L_i^{\pm},\ L_j^{\mp}\right] = 0, \tag{10}$$

and therefore can be controlled separately using the $O(3)$ rolling-ball algorithm. The rotation generated in this way can be written as

$$R^{\pm} = I_4 \cos\theta + \vec{n}\cdot\vec{L}^{\pm}\sin\theta, \tag{11}$$

where

$$\vec{n}\cdot\vec{L}^{\pm} = \begin{vmatrix} 0 & -n_z & n_y & \mp n_x \\ n_z & 0 & -n_x & \mp n_y \\ -n_y & n_x & 0 & \mp n_z \\ \pm n_x & \pm n_y & \pm n_z & 0 \end{vmatrix},$$

and the unit vector $\vec{n}$ would normally be defined by Eq. (1). Thus, we can manipulate all the degrees of freedom of *four-dimensional* orientation by using *two copies* of the rolling ball, one for L_i^+ and one for L_i^-.

An alternative technique, which applies also to rotations in N-dimensional Euclidean space, is to break up the group $O(4)$ (or $O(N)$ in N

dimensions) into $O(3)$ subgroups and treat each as an independent rolling-ball transformation.

Lorentz Transformations

Physical systems at very high velocities must be studied using Lorentz transformations of spacetime, rather than Euclidean rotations. Lorentz transformations mix space and time together and preserve a Minkowski-space quadratic form that has one negative component. Pure velocity changes, or "boosts," are similar in form to rotations with hyperbolic functions replacing trigonometric ones. A boost to a frame with velocity $\vec{v} = \hat{v} \tanh \xi$ transforms the vector $(\vec{x}, t)$ by the matrix

$$
\begin{vmatrix}
\delta_{ij} + \hat{v}_i \hat{v}_j (\cosh \xi - 1) & \hat{v}_j \sinh \xi \\
\hat{v}_i \sinh \xi & \cosh \xi
\end{vmatrix}. \tag{12}
$$

To implement $O(2, 1)$ Lorentz transformations, which preserve the form $\mathrm{diag}(1, 1, -1)$, we interpret mouse motions as small velocity changes of a "Lorentz rolling ball" in the direction the mouse is moving. (We could also study the transformation group $O(3, 1)$ of physical spacetime; unfortunately, the analogy of the argument leading to Eq. (10) requires the introduction of complex vectors.)

The infinitesimal generators of $O(2, 1)$ transformations are the boost operators

$$
B_x = t\frac{\partial}{\partial x} + x\frac{\partial}{\partial t}, \quad B_y = t\frac{\partial}{\partial y} + y\frac{\partial}{\partial t},
$$

and the operator

$$
L = x\frac{\partial}{\partial y} - y\frac{\partial}{\partial x}
$$

producing rotations in the x-y plane. The boost operators transform under rotations as ordinary vectors, $[L, B_x] = -B_y$, $[L, B_y] = +B_x$, while their mutual commutation produces a rotation with the *opposite sign* compared to the analogous $O(3)$ operators, $[B_x, B_y] = +L$.

We relate the mouse input to the $O(2, 1)$ transformation Eq. (12) by replacing Eq. (1) by

$$\hat{v}_x = \frac{+dx}{dr}, \qquad \hat{v}_y = \frac{+dy}{dr}, \tag{13}$$

and choosing the boost parameter to be $\xi = \tanh^{-1}(dr/s) \approx (dr/s)$, where's is a suitable scaling factor that ensures $(dr/s) < 1$.

We then find that moving the input device in small clockwise circles produces a rotation of the spatial part of the coordinate frame in the *clockwise* direction, which is the opposite of the result for standard $O(3)$ rotations! This effect, known as the *Thomas Precession,* makes the rolling-ball technique a very natural one for Lorentz transformations.

Summary

In summary, the rolling-ball technique provides an approach to controlling three degrees of rotational freedom in interactive graphics systems with two-dimensional input devices that does not depend on the state, position, or history of the input device. Because of the algorithm's rich group-theoretical origins, a number of related scientific visualization applications naturally present themselves. Becoming fluent with the technique requires some effort on the part of the user. But, once mastered, this method provides context-free, exploratory orientation adjustment that strongly supports the feeling of direct manipulation.

Acknowledgment

II.4

INTERVAL ARITHMETIC

Jon Rokne
The University of Calgary
Calgary, Alberta, Canada

In computer graphics the problem of discretization occurs in two different areas:

- The final output from a computation is a two-dimensional picture that is displayed or printed using devices with finite resolution, causing undesirable effects such as aliasing.

- The computations required to determine positions, intensities, and colors take place either in general-purpose computing devices or in special-purpose graphics computers. In either case the fundamental method for storing real numbers is the so-called floating-point representation. This representation allocates a fixed number of binary digits for storing a floating-point number that can be an input or output quantity or the result of an intermediate calculation.

We will discuss a tool, interval analysis, that uses guaranteed upper and lower bounds for the estimation and control of numerical errors that can occur during numerical calculations in particular those occurring in computer graphics problems. Interval arithmetic is an extensive subject, and we only touch upon some of the basic ideas. We do, however, note that interval arithmetics and analysis has led to the development of new techniques based on inclusion and contraction that are appropriate techniques for some problems in computer graphics.

We first give an example of a problem that may occur in a graphics application. A primitive routine may consist of determining whether two

lines (for simplicity in E^2) are parallel or not (that is, do they have a finite intersection or not). If they intersect, compute the point of intersection. Let the lines be

$$0.000100x + 1.00y = 1.00, \qquad (1)$$

$$1.00x + 1.00y = 2.00, \qquad (2)$$

and assume the arithmetic is a three-digit rounded arithmetic. The true solution rounded to five digits is $x = 1.0001$ and $y = 0.99990$, whereas the three-digit arithmetic gives $y = 1.00$, $x = 0.00$ using the procedure described in Forsythe and Moler (1967). Other erroneous results are obtained in this text as well, using different arrangements of the sequence of calculations. From this example it is seen that such calculations can have large errors such that an intersection that was supposed to be within a region may actually be calculated to be on the outside of the region.

  Another example is region-filling, where the connectedness of the region depends on a calculation that can be fraught with errors in the same manner as the intersection calculation.

  Such errors are difficult to guard against, and eventually they will generate artifacts in computer-generated scenes that are undesirable and difficult to track down and rectify in large programs.

  A number of these errors can be controlled automatically using interval arithmetic. It enables a program to give one of three answers for an item p and a set P.

1. p is definitely in P.

2. p is definitely not in P.

3. Within the computations performed and the precision available it is not possible to tell whether $p \in P$ or $p \notin P$, that is, the result is uncertain.

  In terms of the preceding examples, we would be able to state that lines intersect, that they do not intersect, or that it is uncertain whether they intersect or not. Similarly, we can state that a domain is connected, that it is not connected, or that it is uncertain whether the domain is connected

or not. In each such case decision procedures can be built in to the program to deal with the three cases.

Interval arithmetic has a long history; however, its modern use originated with the publication of the book *Interval Analysis* by Moore (1966). Subsequently, a large number of publications have been devoted to the subject. Bibliographies were published by Garloff (1985, 1987), a number of conferences have been held, and recently a new Soviet journal, *Interval Computations* (Institute for New Technologies, 1991), has been started that is entirely devoted to interval analysis.

Interval arithmetic is defined as follows: Let $I = \{A: A = [a, b], a \cdot b \in R\}$ be the set of real compact intervals and let $A, B \in I$. Then the interval arithmetic operations are defined by

$$A * B = \{\alpha * \beta : \alpha \in A, \beta \in B\},$$

where $* \in \{+, -, \cdot, /\}$ (note that $/$ is undefined when $0 \in B$), that is, the interval result of $A * B$ contains all possible point results $\alpha * \beta$ where α and β are real numbers such that $\alpha \in A$ and $\beta \in B$ and $*$ is one of the basic arithmetic operations.

This definition is motivated by the following argument. We are given two intervals A and B and we know that they contain exact values x and y, respectively. Then the definition guarantees that $x * y \in A * B$ for any of the operations given above *even though we do not know the exact values of x and y.*

This definition is not very convenient in practical calculations. Letting $A = [a, b]$ and $B = [c, d]$, it can be shown that it is equivalent to

$$[a, b] + [c, d] = [a + c, b + d],$$

$$[a, b] - [c, d] = [a - d, b - c],$$

$$[a, b] \cdot [c, d] = [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)],$$

$$[a, b]/[c, d] = [a, b] \, [1/d, 1/c] \text{ if } 0 \sim [c, d], \tag{3}$$

which means that each interval operation $* \in \{+, -, \cdot, /\}$ is reduced to real operations and comparisons.

One very important property of interval arithmetic is that

$$A, \ B, \ C, \ D \in \quad A \subseteq B, \ C \subseteq D, \quad \Rightarrow A * C \subseteq B * D$$

$$\text{for } * \in \{+, \ -, \ \cdot \ , \ /\} \quad (4)$$

if the operations are defined. In other words, if A and C are subsets of B and D, respectively, then $A * C$ is a subset of $B * D$ for any of the basic arithmetic operations. Therefore, errors introduced at any stage of the computations, such as floating-point errors or input errors, can be accounted for. Because of the importance of Eq. (4), it has been called the *inclusion isotony* of interval operations.

One consequence of this is that any programmable real calculation can be embedded in interval calculations using the natural correspondence between operations so that if $x \in X \in I$, then $f(x) \in f(X)$, where $f(X)$ is interpreted as the calculation of $f(.x)$ with x replaced by X and the operations replaced by interval operations.

Another important principle of interval arithmetic is that it can be implemented on a floating-point computer such that the resulting interval contains the result of the real interval computations using Eqs. (3) and directed rounding. Several software systems are available for this purpose, such as PASCAL-SC (Bohlender *et al.,* 1981). The implementation only has to take care that each calculation of interval endpoints is rounded outwards from the interior of the intervals.

Interval arithmetic has some drawbacks as well:

- Subtraction and division are not the inverse operations of addition and multiplication.

- The distributive law does not hold. Only a subdistributive law $A(B + C) \subseteq AB + AC, \ A, \ B, \ C \in I,$ is valid.

- The interval arithmetic operations are more time-consuming than the corresponding real operations roughly by a factor of 3 (although interval arithmetic implementations of some problems may run faster than the corresponding real versions; see Suffern and Fackerell, 1991).

As a simple example of the use of interval computations, we consider the intersecting lines problem given above using three-digit interval

arithmetic. Using Cramer's rule, we get

$$x = \begin{vmatrix} 1.00 & 0.000100 \\ 2.00 & 1.00 \end{vmatrix} \bigg/ \begin{vmatrix} 0.000100 & 1.00 \\ 1.00 & 1.00 \end{vmatrix}$$

and

$$y = \begin{vmatrix} 0.000100 & 1.00 \\ 1.00 & 2.00 \end{vmatrix} \bigg/ \begin{vmatrix} 0.000100 & 1.00 \\ 1.00 & 1.00 \end{vmatrix}.$$

Using interval arithmetic we obtain

$$x \in X = [0.980, \ 1.03]$$

and

$$y \in Y = [0.989, \ 1.02],$$

which in each case contains the exact solution. This calculation corresponds to a more stable calculation in real arithmetic than the one quoted from Forsythe and Moler (1967). If that particular sequence of calculations were performed in interval arithmetic, then the interval would be larger, but in all cases we have that *the exact result* is contained in the resulting intervals.

One interesting feature of interval arithmetic is that it can be used to develop new algorithms that are not simply extensions of algorithms in real arithmetic. One example of this is the interval Newton method first developed by Moore (1966). Let $F(x)$ be given and suppose that we want to find the points ξ where $F(\xi) = 0$ in a given interval X_0. Then the interval Newton method is defined to be the iteration

$$X_{n+1} = m(X_n) - F(X_n)/F'(m(X_n)), \qquad n = 0, \ 1, \dots,$$

where $m([a; b]) = ((a + b)/2$ and $F'(X)$ is the interval evaluation of the derivative of F. The method has some interesting properties.

1. If a zero, ξ, of F exists in X_0, then $\xi \in X_n$ for all n; see Moore (1966). This means that all the zeros in the initial X_0 are retained in subsequent intervals.

2. If X_n, is empty for some n, then F has no zeros in X (Moore, 1966).

Further properties of interval iterations for the solution of equations can be found, for example, in Neumaier (1990).

This method can be applied in computer graphics in ray-tracing. The intersection calculation between an implicit surface and a ray results in a problem of finding either one (the smallest) root or all the roots of a function $F(x) = 0$ (see Hanrahan, 1989). With the use of interval arithmetic techniques, the result can be guaranteed to be contained in the resulting intervals, avoiding anomalies in the rendering process (see Kalra and Barr, 1989, for a discussion of the problem).

Further discussions on the use of interval arithmetic for implicit surface rendering, in contouring algorithms, and in planarity estimation is found in Mudur and Koparkar (1984) and Suffern and Fackerell (1991), where it is combined with subdivision techniques in order to improve the results.

See also G2, 394.

FAST GENERATION OF CYCLIC SEQUENCES

Alan W. Paeth
NeuralWare Incorporated
Pittsburgh, Pennsylvania

Free-running inner-loops often require a sequence of values or conditions that repeat every N steps. For instance, a technique for high-speed Z-buffer drawing (Booth *et al.,* 1986) must perform buffer swapping and housekeeping in cycles of three. When N is not a power of two, direct examination of a register's low order bits may not be used to form a count modulo N. Similarly, a fast 2-D N-gon generator requires the cyclic production of a sequence of N values, with vertex N identical to vertex zero. This Gem considers compact methods for $N < 8$ that use neither more than three machine instructions nor three registers. No conditional logic is employed, making the techniques well-suited to hand coding.

$N = 2$ (Review)

The familiar two-fold "toggle" alternates between **true** and **false**:

> condition := **not**(condition); *True in alternating cases*
> **if** (condition) . . . (2.1)

Similarly, a two-fold "cycle" of values is a simple alternation. When both values are predetermined, one instruction and one integer register suffice:

> **register** a := v1; *initialize*
> **constant** c := v1 + v2;
>
> **repeat** *cycle*
> a := a − c; a:[v1 v2 · · ·] (2.2)

For instance, Wirth (1973) speeds a prime number sieve by using (2.2) to generate the sequence [2 4 2 4···] of distances between successive odd integers not divisible by three (Knuth, 1981). Rewriting the arithmetic of (2.2) with logical **xor** operations yields a well-known, patented method for inverting a frame-buffer's pixels in alternating fashion. In arithmetic form, a pixel inversion scheme well-suited to greyscale frame buffers is rederived (Newman and Sproull, 1979).

   Most generally, the cyclic sequence is specified only at run time. For $N = 2$, cycling is swapping, easily accomplished in three arithmetic or logical operations without resort to a third holding register, as described in previous Gems by Paeth (1990) and Wyvill (1990), respectively.

N = 3 (Extension)

The pairwise swapping technique does not extend gracefully: Cyclic permutation of the sequence [a, b, c] by exchanging (for example) elements at locations (1, 2) and (2, 3) costs six instructions and three registers. A first-principles cyclic brigade method requires $N + 1$ registers and $N + 1$ assignments. Though straightforward, the latter still exceeds both the instruction and register limits set forth in the preface:

```
r1 := r1 xor r2;      r2 := r2 xor r1;                  rx := r1;   r1 := r2;
r1 := r1 xor r2;      r2 := r2 xor r3;   <versus>       r2 := r3;   r3 := rx;
r3 := r3 xor r2;      r2 := r2 xor r3                                          (3.1)
```

Often, as in (2.1), values are required only to trigger a 1-in-N event. For $N = 3$, two registers and two lines suffice. Each register instruction is of the compact, two-op form << rx = rx op ry >>:

```
register r1 := 0;           Three fold trigger
register r2 := 1;
repeat                      cycle:
    r1 := r1 xor r2;         r1: [1 1 0 ···]
    r2 := r2 xor r1;         r2: [0 1 1 ···]
```

This produces the tertiary ($r1$, $r2$) column set shown. The trigger occurs when a register is zero. Testing on $r2$ streamlines the operation under the

assumption that hardware condition codes are set by the preceeding logical operations. The phase of the test may be adjusted by substituting a column other than the third in the initialization of ($r1$, $r2$). Triggering 2-in-3 times defines the complementary set: a test for nonzero is substituted. The three phase-distinct 1-in-3 tests may be done concurrently, forming a cyclic switch:

$$
\begin{array}{lll}
\textbf{if } r \ = 0 \ \textbf{then} \ . \ . \ . & \quad \textit{1-in-3, phase} =0 \\
\textbf{if } rl \ = r2 \ \textbf{then} \ . \ . \ . & \quad \textit{1-in-3, phase} =1 \\
\textbf{if } rl \ = 0 \ \textbf{then} \ \ . \ . \ . & \quad \textit{1-in-3, phase} =2 \qquad (3.2\text{b})
\end{array}
$$

The conditions and related blocks may be embedded among the **xor** operations to take advantage of implicit condition code sensing. That is, the two **xor** lines that implicitly define the modulo three counter need not be adjacent.

Cyclic permutation of three variables in three registers can be done in the minimum number of instructions (three). The derivation is not obvious and relates the threefold arithmetic case described later.

```
register int a = c1;           Three fold cycle
register int b = c1 xor c2;
constant int c = c1 xor c2 xor c3;

repeat                         cycle:
    a = a xor b;               a: [c1  c2  c3···]
    b = b xor c;
    b = b xor a;                                       (3.3)
```

The use of logical **xor** is valuable as the elements may be a mixed sequence of integers, pointers and floats; arithmetic operations would not permit this. Note that the last two lines both update the value in b, while register c is never written. This suggests the alternate line:

```
    b := b xor (a xor c);                              (3.3b)
```

in which c is equated to a predetermined compile time constant. However, the value must typically occupy a third register at run time. (See the C-code for a two register variant which produces the cycle [1, 2, 3].)

When mere triggering is required, fixing $c = 0$ elides the middle line of the cycle, reconstructing the $N = 3$ triggering case. Alternately, the two lines may be regarded as the first two of three lines of the familiar **xor** swap code. Because the final line of the latter matches the first, three passes through the two-line **xor** code produce the same sequential action on the two registers as do two passes through the three-line code (this is suggested in Eq.(3.1a) by the grouping of instructions). Both define a restoring double swap: the identity operation on two elements in no less than six steps. Thus, the two-line code forms cycles of length three while generating the sequence $[r1, r1 \oplus r2, r2]$.

N =3, 4, 6

Remarkably, cycles of length up to $N = 6$ require only two registers. Clearly, there is insufficient storage for swapping of all elements, else a cyclic brigade of $N + 1$ registers and $N + 1$ assignments would suffice. Instead, the goal is to derive a set of values (on one or both registers) in which all generated values are distinct. Thus, the registers must "count," and rival first-principles code such as a quick hexagon-drawing routine:

```
Xval =X_Value_Table[(i := (i + 1 mod 6))];
Yval = Y_Value_Table[i];
```

Here the modulus is a major expense: its cost is on par with integer division. The other first-principles method uses conditional logic to restart a decrementing counter, giving a large branch penalty on modern pipelined hardware, made worse by small N.
 As will be seen in (6.2), vertex production of 2-D hexagons may use this sixfold cycle:

```
register x = y = 1;
repeat
   Xval = X_coord[x := x  +  y];
   Yval = Y_coord[y := y + not(x)];
```

where **not**(x) is bit inversion, i.e., **not**(x) = x **xor** (-1) under two

complement hardware. Arrays of length seven are required having suitable offsets, as seen in the companion C code.

N = 6 Derivation

Nonuniform rotation may be modeled by functional composition. That is, $F(F. \, . \, .(F([x])). \, . \, .) = [x]$ in no less than N steps. For instance, the linear fractional function $F(x) = [2x - l]/[\, x \, + \, 1]$ yields $F^6(x) = x$ (Yale, 1975). Such forms may be equated directly to the algebra of 2×2 matrices (Birkhoff and MacLane, 1965); the former are treated preferentially for ease of derivation.

 The values of two registers may be represented by a point [x, y] on an integer lattice, one coordinate per register. Treated as a (column) vector v, the function F is a 2×2 matrix of which premultiplies v. For a given N, F must be determined such that $F^N v = I$. When F is a shear matrix rotation may be achieved in three shears (Paeth, 1986), requiring only one assignment statement per shear (p.182). When the off-diagonal matrix element is $\{\pm 1\}$, no multiplication occurs and one machine instruction suffices. All-rational forms also yield rotation, but the sets of circumferential points are not roots of unity (vertices of an N-gon inscribed in a unit circle on the complex Cartesian plane). The one solution is for fourfold rotation. That decomposition is:

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}.$$

Regrouping of the first and third matrix (p.192) allows a two-line rotation, useful on machines that provide an implicit multiplication by two:

```
x := x + y;              x := x + (2*y);
y := y-x;      <or>   y := y-x;
x := x + y;                                          (4.1)
```

This sequential form is slightly more expensive than the compact $\{x = -y, \; y = x\}$ form made possible when simultaneous reassignment of register values is possible, as in hardware. Rotations of three and six may

be formed by finding the eigenvalues of the product of an X and Y shear in symbolic form and equating them with the roots of unity, thus determining the values of the two off-axis elements. In its most compact form, this yields the quadratic equation $z = \frac{1}{2}\left(m \pm \sqrt{m^2 - 4}\right)$, which can represent roots of unity $z^N = 1$ for $N = \{1, 2, 3, 4, 6\}$, with the respective values $m = \{2, -2, -1, 0, 1\}$. Solution using MAPLE on the matrix equation $(XY)^N = I$ with symbolic matrix elements does not reveal real-valued, integral forms markedly distinct from this general solution:

$$\left(\begin{bmatrix} 1 & 0 \\ m \mp 2 & \pm 1 \end{bmatrix}\begin{bmatrix} \pm 1 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \pm 1 & 1 \\ \pm m - 2 & m \mp 1 \end{bmatrix}\right)^N = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},$$

$$(m, N) = \{(-1, 3), (0, 4), (1, 6)\}$$

Both three-and sixfold rotation using unit elements are thus possible. These are unexpected, given the irrationality of cos 60°. The distortion of the simplified two-shear rotational form has become a virtue in fixing vertices to integral locations. Note that the three nontrivial solutions for $N = (3, 4, 6)$ enumerate the set of N-gons that tile the plane (Figs. 1a, 1b).

An automated examination of all three-instruction, three-register shears having small multipliers was conducted. No solutions for new N were found, and most forms were not markedly distinct. The two-register
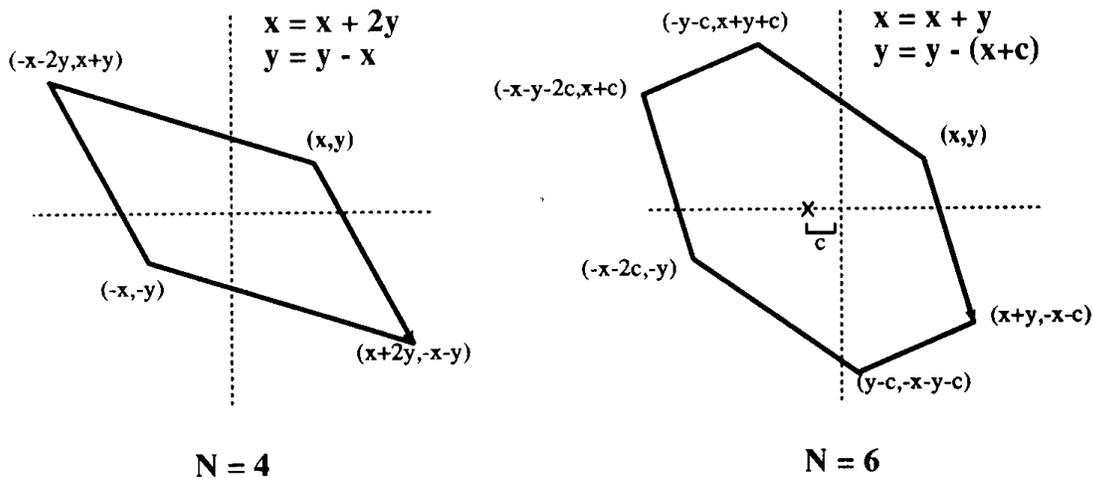
Figure 1.

$N = 3$ form was rewritten using **xor**, leading to (3.3). The $N = 6$ two-register form was seen to accommodate one additional constant, which offsets the hexagon in *x*, illustrated in Fig. 1b and presented algebraically below:

```
a := a+b;       a:   [a  a+ b  b+c           −a+2c  2c  −a −b c −b  ]
b:= b −a+c;  b:   [b  c   −a   c −a −b  −b      a −c     a+b  −c]
```
(6.1)

When $c = 0$, the values in register a lag those in *b* by two steps, suggestive of a (cos *t*, sin *t*) generator, except the 90° quadrature becomes a 120° phase offset. With $c \neq 0$ the generation of a sequence of distinct values is achieved, meeting the original goal. Setting $c = -1$ allows the implicit formation of the $-(a+1)$ term using the logical ones complement (Paeth and Schilling, 1991), giving:

```
a=b= 1;                 six-fold fixed-ualue cycle
repeat
   a := a + b;          a:[1   2   0 -3 -4 -2]
   b := b + not(a);   b:[1 -2 -3 -1   2   3]
```
(6.2)

in which the *c* offset displaces the hexagon's center laterally, removing symmetry of central inversion. This helps achieve distinct values. With $a > 0$, $b > 0$ and $2c > a + b$, the sequence in *a* is always positive.

N = 6 (Triggering)

Arbitrary triggering is possible using the sixfold form. The distinct values of the preceding algorithm allow concurrent 1-in-N triggers having any phase offset. However, 2-in-N and 3-in-N forms with nonadjacent triggers present greater difficulty: They may not be created by replacing equality test with an inequality. This is a consequence of the figures' convexity: In geometric terms, a test such as $y > 4$ represents a horizontal half-space of values. Intersection of the polygon by the plane splits it into two distinct boundary sets of conterminous vertices. The goal is a simple trigger that does not resort to intra-register bit testing as in the companion Gem cited above.

 Six states allow 64 trigger patterns, in which a "*" (".") in the *i*th place represents a (dis)arming of the trigger for the *i*th state. Elimination

of complementary patterns halves this number. Patterns containing repetitive triggers such as "***…" and "*.*.*." may be decomposed into super or subcycles and are eliminated. Left are three prototypical patterns, having one, two, or three bits set. Testing uses the sixfold rotation variant in (6.2) with implicit ($c = -1$) and starting values $a = b = 0$:

$$
\begin{array}{rcccccc}
\text{a:} & 0 & 0 & -1 & -2 & -2 & -1 \\
\text{b:} & 0 & -1 & -1 & 0 & +1 & +1 \\
\hline
\end{array}
$$

$$
\begin{array}{rcccccc}
a = 0 \text{ AND } b = 0: & * & \cdot & \cdot & \cdot & \cdot & \cdot \\
a = 0: & * & \cdot & * & \cdot & \cdot & \cdot \\
a = 0 \text{ OR } b = 0: & * & * & \cdot & * & \cdot & \cdot
\end{array}
\tag{6.3}
$$

The widespread use of **xor** suggests methods similar to pseudo-random number (RPN) generation on the field of integers **mod 1** (see Morton, 1990). The traditional shift and carry-test logic hardware may be "wired" directly into three **xor** register instructions having a permuting form, giving

repeat   *cycle by seven*
    b := b **xor** a;
    c := c **xor** b;
    a := a **xor** c;

This yields the table of values listed below.

A	B	C
a	b	c
$a \oplus b$	$b \oplus c$	$a \oplus b \oplus c$
$a \oplus c$	a	b
c	$a \oplus b$	$b \oplus c$
$a \oplus b \oplus c$	$a \oplus c$	a
b	c	$a \oplus b$
$b \oplus c$	$a \oplus b \oplus c$	$a \oplus c$

(7.1)

Here column A leads B by two steps, likewise B ahead of C, but C leads A by three steps. Each column takes on all $N-1$ possible arrangements of **xor** among the three variables, omitting the forbidden zero state. This does not restrict the periodic production of zero elements, formed either by setting any (but not all) of {a, b, c} to zero, or by equating initial values in two registers, since *M* **xor** *M = 0.*

Use of four registers ($r=4$) suggests $2^4-1 = 15$ states. Since *r* is even, *N* is composite with factors $(2^2+1)(2^2-1)$. This reveals the subcycle for $N=5$, rounding out the table for small *N.* However, this method shows only a marginal gain over the brigade method (five variables, one temporary register, six assignments) and was not explored. For those inclined to large *N,* factors may be used to compose larger cycles: concurrent loops of relatively prime length resynchronize after a number of steps equal to the product (the GCM) of their lengths.

For the last single-digit value, $N=9$ remains difficult as it is neither prime nor a square-free composite. The next primes at (11, 13) are not of the 2^m-1 Mersenne form. By Fermat's theorem, they (and any prime p) are factors of 2^{p-1}, here $2^{10}-1$ and $2^{12}-1$. Since this implies that the number of registers grows at least linearly with the cycle length for **xor** methods, the brigade method wins by virtue of simplicity. Although the practical limit of all methods explored thus far is $N < 8$, more exotic and convoluted methods are possible and may be examined through brute-force means. One is presented below.

N = 24

As a last example, the code

```
register a = 4;
register b = 3;
repeat
    a := a − b;
    a := a bit-and 15; explicit limit on register a
    b := b xor a;                                        (24.1)
```

offers a method of cycling modulo 24. Limiting the domain of register *a* to 16 values necessarily introduces value multiplicity. The initial values

chosen confine both *a* and *b* to the domain [1..15] and further insures that they are never simultaneously equal.

 This code's value is in forming parallel 24:1, 12:1, 8:1, and 6:1 rate division using the conditional tests *(b* = 1), (*a* = 4), (*b* = 7) and (*b* = 12), respectively. These tests are chosen so at most one is true at any step, allowing rate multiplication (up to 10-in-24) by combining the {1, 2, 3, 4} -in-24 tests by **or**ing of the triggering bits. Note that only the 3-in-24 rate shows slight nonuniformity:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
a:	1	15	2	3	7	12	5	3	2	15	3	4	9	7	2	11	15	12	13	1	12	7	11	4
b:	2	13	15	12	11	7	2	1	3	12	15	11	2	5	7	12	3	15	2	9	11	12	7	3
b=1:	*																							
a=4:												*												*
b=7:						*									*								*	
b=12:				*						*						*					*			

Summary

Methods for cyclic production of both arbitrary values and of Boolean states has been presented. Cases *N* = {2, 3, 4, 6, 7} were treated in detail. The extensive C-code variants provided in the appendices make a useful set of additions to the graphics programmer's bag of tricks.

..

II.6

A GENERIC PIXEL SELECTION MECHANISM

Alan W. Paeth
NeuralWare Inc.,
Pittsburgh, Pennsylvania

Reversing the colors of a frame buffer's pixels is a common way to highlight a region. A useful reversal function provides color pairs that are visually distinct. On newer hardware, lookup tables (which map a pixel's appearance) are keyed by window, introducing spatial dependence. This burdens the design of a "best" function. This gem offers a simple *a priori* solution that guarantees visually distinct color pairs, though their eventual appearance remains unknown to the algorithm. Typical use is in creating screen-wide, window-invariant tools, such as system cursors or selection rectangles for display "snapshots."

A useful reversing function F on pixel p satisfies two algebraic criteria: $F(F(p)) = p$ and $F(p)\ \mu\ p$. The first assures that the function is its own inverse. The second is crucial in guaranteeing that the two elements in any color pair are "not nearly equal," leaving them visually distinct. For one-bit pixels, complementation (bit toggle) is the obvious solution. At higher precision, the (ones) complement of all bits becomes an arithmetic operation: $F(p) = \textbf{not}(p) = -1 - p$ under two's complement arithmetic (Paeth, 1991). This has been generalized (Newman and Sproull, 1979) for $o \le c < 1$ as $F_c(p) = frac(c - p)$. This fails the second criterion: For parameter c a pixel of value $c/2$ maps onto itself. Geometrically, the unit interval has been displayed (by c) and mirrored onto the original interval, thereby introducing a stationary point.

The solution used in the Palette system (Higgins and Booth, 1986) returns to logical operations. Given a binary integer that defines discrete positions along an interval, bit complementation of merely the uppermost bit swaps the interval's lower and upper halves without any mirroring. The pixels in any color pair are now displaced by half the interval

distance, guaranteeing distinct colors. In the case of color-mapped pixels (which serve as indices), elements in a pair are far removed in the domain of the mapping function, yielding colors likewise removed in the range should the color map define a monotonic function—a common case. Certain nonlinear non-Cartesian color maps (Paeth, 1991) also work well under this function and support a simple geometric interpretation.

The generic function may now be constructed by making simple assumptions. The pixel precisions of monochromatic channels on typical framebuffers are one, four, or eight bits. The operation

$$\textbf{macro } \text{bwpixflip}(x) \qquad x := x \textbf{ bit-xor } 133 \qquad \textit{hex 85}$$

complements the topmost bit in all cases without knowledge of the precision in use. When the underlying pixel is of lower precision, toggling the higher-order bits is of no consequence or is squelched by action of a hardware write mask. Conversely, operation upon a high precision, pixel will complement additional low-precision bits, but these are sufficiently removed to be of much consequence.

For RGB pixels, three copies of hexadecimal 85 assures operation on three adjacent channels. This also introduces a toggle at bit 12, a further benefit on hardware providing extended monochromatic precision or color table indexing. The generic color reverse function is

$$\textbf{macro } \text{pixelflip}(x) \qquad x := x \textbf{ bit-xor } 8750469 \qquad \textit{hex 858585}$$

Threefold use of the operation swaps halves of the unit interval along each color axis. Geometrically, this represents a shuffling of eight sub-cubes within the unit color cube about the central point $(\frac{1}{2}\frac{1}{2}\frac{1}{2})$ of midlevel gray. In non-Rubik fashion, the orientation of each cubelet is preserved (fig. 1a). In the first-principles "**xor – 1**" case (not shown) an additional central inversion of the eight cubelets inverts the entire solid and the undesirable stationary point is reintroduced at the mid-gray position.

Finally, it is often advantageous to leave the blue channel uncomplemented. When blue occupies the uppermost pixel bits (as on the Adage/Ikonas or SGI/Iris), complementation of the lower 16 bits defining the red and green channels still occurs; all monochromatic and lookup cases (in which pixel precision never exceeds 16 bits) are also
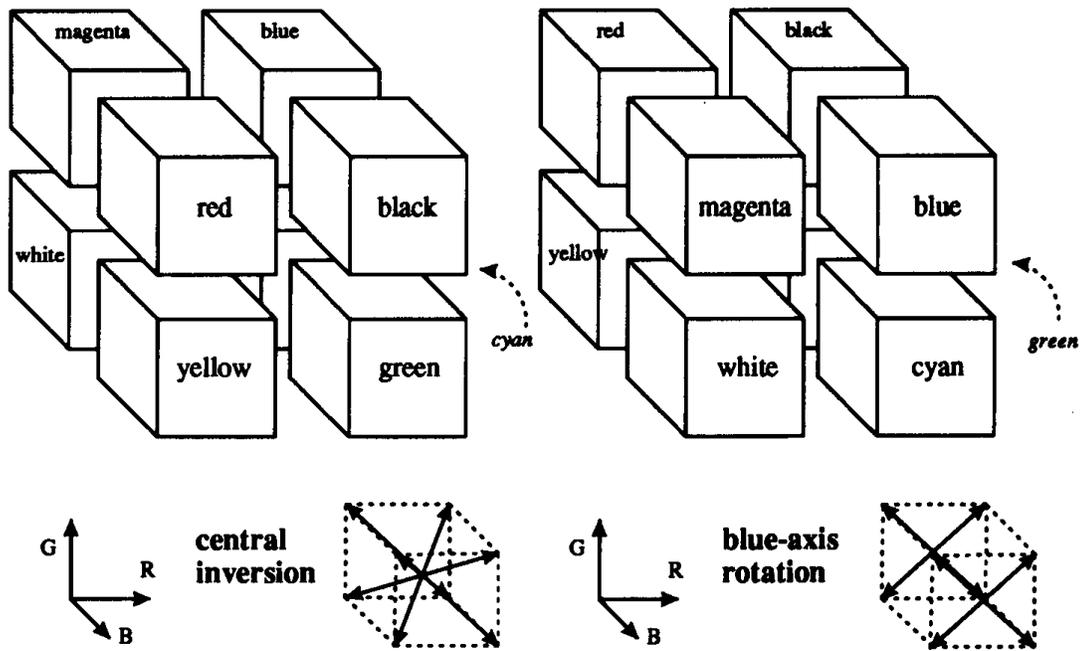
Figure 1

covered implicitly. The alternate generic macro is

macro pixelflip2(x)   x := x **bit-xor** 34181   *hex 8585*

For 24-bit color, preservation of blue means that subcubes no longer swap through central inversion (Fig. la), but are instead rotated a half-turn about the blue axis in "Ferris-wheel" fashion (lb). This creates a pair of opponent colors (red, green) for which the human visual system is highly responsive, plus pairs (blue, white), (cyan, magenta) and (yellow, black). The alternate macro supports the use of short, 16-bit integers in the reversal.

See also G1, 215; G1, 219; G1, 233; G1, 249.

NONUNIFORM RANDOM POINT SETS VIA WARPING

Peter Shirley
Indiana University
Bloomington, Indiana

We often want to generate sets of random or pseudorandom points on the unit square for applications such as distribution ray tracing. There are several methods for doing this, such as jittering and Poisson disk sampling. These methods give us a set of N reasonably equidistributed points on the unit square: (u_1, v_1) through (u_N, v_N).

Sometimes, our sampling space may not be square (e.g., a circular lens) or may not be uniform (e.g., a filter function centered on a pixel). It would be nice if we could write a mathematical transformation that would take our equidistributed points (u_i, v_i) as input, and output a set of points in our desired sampling space with our desired density. For example, to sample a camera lens, the transformation would take (u_i, v_i) and output (r_i, θ_i) such that the new points were approximately equidistributed on the disk of the lens.

It turns out that such transformation functions are well known in the field of Monte Carlo integration. A table of several transformations useful for computer graphics is given in Table I. The method for generating such transformations is discussed for the rest of this article. Note that several of these transformations can be simplified for simple densities. For example, to generate directions with a cosine distribution, use the Phong density with $n = 1$. To generate points on the unit hemisphere, use the sector on the unit sphere density with $\theta_1 = 0$, $\theta_2 = \pi/2$, $\phi_1 = 0$, and $\phi_2 = \pi$.

For Monte Carlo methods we must often generate random points according to some probability density function, or random rays according to a directional probability density. In this section a method for one and two dimensional random variables is described. The discussion closely follows that of Shreider (1966).

Table 1. Some Useful Transformation.[a].

Target Space	Density	Domain	Transformation
Radius R disk	$p(r, \theta) = \dfrac{1}{\pi R^2}$	$\theta \in [0, 2\pi]$ $r \in [0, R]$	$\theta = 2\pi u$ $r = R\sqrt{v}$
Sector of radius R disk	$p(r, \theta) = \dfrac{2}{(\theta_2 - \theta_1)(r_2^2 - r_1^2)}$	$\theta \in [\theta_1, \theta_2]$ $r \in [r_1, r_2]$	$\theta = \theta_1 + u(\theta_2 - \theta_1)$ $r = \sqrt{r_1^2 + v(r_2^2 - r_1^2)}$
Phong density exponent n	$p(\theta, \phi) = \dfrac{n+1}{2\pi} \cos^n \theta$	$\theta \in \left[0, \dfrac{\pi}{2}\right]$ $\phi \in [0, 2\pi]$	$\theta = \arccos\left((1 - u)^{1/(n+1)}\right)$ $\phi = 2\pi v$
Separated triangle filter	$p(x, y)(1 - \lvert x\rvert)(1 - \lvert y\rvert)$	$x \in [-1, 1]$ $y \in [-1, 1]$	$x = \begin{cases} 1 - \sqrt{2(1-u)} & \text{if } u \geq 0.5 \\ -1 + \sqrt{2u} & \text{if } u < 0.5 \end{cases}$ $y = \begin{cases} 1 - \sqrt{2(1-v)} & \text{if } v \geq 0.5 \\ -1 + \sqrt{2v} & \text{if } v < 0.5 \end{cases}$
Triangle with vertices a_0, a_1, a_2	$p(a) = \dfrac{1}{\text{area}}$	$s \in [0, 1]$ $t \in [0, 1-s]$	$s = 1 - \sqrt{1 - u}$ $t = (1 - s)v$ $a = a_0 + s(a_1 - a_0) + t(a_2 - a_0)$
Surface of unit sphere	$p(\theta, \phi) = \dfrac{1}{4\pi}$	$\theta \in [0, \pi]$ $\phi \in [0, 2\pi]$	$\theta = \arccos(1 - 2u)$ $\phi = 2\pi v$
Sector on surface of unit sphere	$p(\theta, \phi)$ $= \dfrac{1}{(\phi_2 - \phi_1)(\cos\theta_1 - \cos\theta_2)}$	$\theta \in [\theta_1, \theta_2]$ $\phi \in [\phi_1, \phi_2]$	$\theta = \arccos[\cos\theta_1 + u(\cos\theta_2 - \cos\theta_1)]$ $\phi = \phi_1 + v(\phi_2 - \phi_1)$
Interior of radius R sphere	$p = \dfrac{3}{4\pi R^3}$	$\theta \in [0, \pi]$ $\phi \in [0, 2\pi]$ $R \in [0, R]$	$\theta = \arccos(1 - 2u)$ $\phi = 2\pi v$ $r = w^{1/3} R$

[a]The symbols u, v, and w represent instances of uniformly distributed random variables ranging over [0, 1].

If the density is a one-dimensional $f(x)$ defined over the interval $x \in [a, b]$, then we can generate random numbers α_i that have density f from a set of uniform random numbers ξ_i, where $\xi_i \in [0, 1]$. To do this we need the probability distribution function $F(x)$:

$$F(x) = \int_a^x f(x')\,d\mu(x'). \tag{1}$$

To get α_i we simply transform ξ_i:

$$\alpha_i = F^{-1}(x_i),$$

where F^{-1} is the inverse of F. If F is not analytically invertible, then numerical methods will suffice because an inverse exists for all valid probability distribution functions.

  If we have a two-dimensional density (x, y) defined on $[a, b: c, d]$, then we need the two-dimensional distribution function:

$$F(x, y) = \int_c^y \int_a^x f(x', y') \, d\mu(x', y'). \tag{3}$$

We first choose an x_i using the marginal distribution $F(x, d)$, and then choose y_i according to $F(x_i, y)/F(x_i, d)$. If $F(x, y)$ is separable (expressable as $g(x)h(y)$), then the one-dimensional techniques can be used on each dimension.

  As an example, to choose reflected ray directions for zonal calculations or distributed ray tracing, we can think of the problem as choosing points on the unit sphere or hemisphere (since each ray direction can be expressed as a point on the sphere). For example, suppose that we want to choose rays according to the density

$$p(\theta, \phi) = \frac{n + 1}{2\pi} \cos^n \theta, \tag{4}$$

where n is a Phong-like exponent; θ is the angle from the surface normal and $\theta \in [0, \pi/2]$ (is on the upper hemisphere); and ϕ is the azimuthal angle ($\phi \in [0, 2\pi]$). The distribution function is

$$P(\theta, \phi) = \int_0^\phi \int_0^\theta p(\theta', \phi')\sin\theta' \, d\theta' \, d\phi'. \tag{5}$$

The $\sin \theta'$ term arises because $d\omega = \sin \theta \, d\theta \, d\phi$ on the sphere. When the marginal densities are found, p (as expected) is separable, and we find that a (r_1, r_2) pair of uniform random numbers can be transformed to a direction by

$$(\theta, \phi) = (\arccos((1 - r_1)^{1/(n+1)}), 2\pi r_2). \tag{6}$$

Typically, we want a directional (θ, ϕ) pair to be with respect to some unit vector y (as opposed to the z axis). To do this we can first convert the angles to a unit vector a:

$$a = (\cos \phi \sin \theta , \sin \phi \sin \theta, \cos \theta).$$

We can then transform a to be an a' with respect to ψ by multiplying a by a rotation matrix R ($a' = Ra$). This rotation matrix is simple to write down:

$$R = \begin{bmatrix} u_x & v_x & w_x \\ u_y & v_y & w_y \\ u_z & v_z & w_z \end{bmatrix},$$

where $u = (u_x, u_y, u_z)$, $v = (v_x, v_y, v_z)$, $w = (w_x, w_y, w_z)$, form a basis (an orthonormal set of unit vectors where $u = v \times w$, $v = w \times u$, and $w = u \times v$) with the constraint that w is aligned with ψ:

$$w = \frac{\psi}{|\psi|}.$$

To get u and v, we need to find a vector t that is not colinear with w. To do this simply set t equal to w and change the smallest magnitude component of t to one. The u and v follow easily:

$$u = \frac{t \times w}{|t \times w|},$$

$$v = w \times u.$$

This family of techniques is very useful for many sampling applications. Unfortunately, some sampling spaces (e.g., the surface of a dodecahedron) are not naturally dealt with using the methods in this gem. Special purpose or, as a last resort, rejection techniques are then called for.

See also G1, 438.

II.8

CROSS PRODUCT IN FOUR DIMENSIONS AND BEYOND

Ronald N. Goldman
Rice University
Houston, Texas

Introduction

Cross product is one of the gods' great gifts to mankind. It has many applications in mathematics, physics, engineering, and, of course, computer graphics. Normal vectors, rotations, curl, angular momentum, torque, and magnetic fields all make use of the cross product.

Given two linearly independent vectors u and v in three dimensions, their cross product is the vector $u \times v$ perpendicular to the plane of u and v, oriented according to the right-hand rule, with length equal to $|u||v| \sin\Theta$, where Θ is the angle between u and v. In rectangular coordinates, the cross product can be computed from the simple determinant formula

$$u \times v = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \end{vmatrix}.$$

Equivalently,

$$u \times v = (u_2 v_3 - u_3 v_2, u_3 v_1 - u_1 v_3, u_1 v_2, - u_2 v_1).$$

At first glance, cross product seems to be an artifact of three dimensions. In three dimensions the normal direction to the plane determined by two vectors is unique up to sign, but in four dimensions there are a whole plane of vectors normal to any given plane. Thus, it is unclear how to define the cross product of two vectors in four dimensions. What then

is the analogue of the cross product in four dimensions and beyond? The goal of this gem is to answer this question.

Tensor Product

There is a way to look at the cross product that is more instructive than the standard definition and that generalizes readily to four dimensions and beyond. To understand this approach, we need to begin with the notion of the tensor product of two vectors u, v.

The tensor product $u \otimes v$ is defined to be the square matrix

$$u \otimes v = u^t * v,$$

where the superscript t denotes transpose and $*$ denotes matrix multiplication. Equivalently,

$$(u \otimes v)_{ij} = (u_i, v_j).$$

Notice that for any vector w,

$$w \, (u \otimes v) = (w \cdot u)v.$$

Thus, the tensor product is closely related to the dot product.
Like dot product, the tensor product makes sense for two vectors of arbitrary dimension. Indeed, the tensor product shares many of the algebraic properties of the dot product. However, unlike the dot product, the tensor product is not communative. That is, in general,

$$u \otimes v \neq v \otimes u \qquad \text{because } u_i \, v_j \neq u_j \, v_i \,.$$

Applications of the tensor product of two vectors to computer graphics are given in Goldman (1990, 1991).

Wedge Product

The wedge product of two vectors u and v measures the noncommutativity of their tensor product. Thus, the wedge product $u \wedge v$ is the square

matrix defined by

$$u \wedge v = u \otimes v - v \otimes u.$$

Equivalently,

$$(u \wedge v)_{ij} = (u_i v_j - u_j v_i).$$

Like the tensor product, the wedge product is defined for two vectors of arbitrary dimension. Notice, too, that the wedge product shares many properties with the cross product. For example, it is easy to verify directly from the definition of the wedge product as the difference of two tensor products that:

$u \wedge u = 0,$

$u \wedge v = - v \wedge u$ (anticommutative),

$u * (v \wedge w) \neq (u \wedge v) * w^t$ (nonassociative),

$u \wedge cv = c(u \wedge v) = (cu) \wedge v,$

$u \wedge (v + w) = u \wedge v + u \wedge w$ (distributive),

$u * (v \wedge w) + v * (w \wedge u) + w * (u \wedge v) = 0$ (Jacobi identity),

$r * (u \wedge v) * s^t = (r \cdot u)(s \cdot v) - (r \cdot v)(s \cdot u)$ (Lagrange identity).

The wedge product also shares some other important properties with the cross product. The defining characteristics of the cross product are captured by the formulas

$$u \cdot (u \times v) = v \cdot (u \times v) = 0,$$

$$|u \times v| = |u|^2 |v|^2 \sin^2 \Theta.$$

By the Lagrange identity, the wedge product satisfies the analogous

identities:

$$u * (u \wedge v) * u^t = v * (u \wedge v) * v^t = 0,$$

$$u * (u \wedge v) * v^t = (u \cdot u)(v \cdot v) - (u \cdot v)^2 = |u|^2|v|^2 \sin^2 \Theta.$$

A variant of the last identity can be generated by defining the norm of a matrix M to be

$$|M|^2 = \tfrac{1}{2}\left\{\sum_{ij}(M_{ij})^2\right\}.$$

Then by direct computation it is easy to verify that

$$|u \wedge v|^2 = (u \cdot u)(v \cdot v) - (u \cdot v)^2 = |u|^2|v|^2 \sin^2 \Theta.$$

In addition, the cross product identity

$$(u \times v) \times w = (w \cdot u)v - (w \cdot v)u$$

has the wedge product analogue

$$w \cdot (u \wedge v) = (w \cdot u)v - (w \cdot v)u. \tag{1}$$

The cross product can be used to test for vectors perpendicular to the plane of u and v because

$$w \times (u \times v) = 0 \iff w \perp u, v.$$

Similarly, the wedge product recognizes vectors perpendicular to the plane determined by u and v because by (1),

$$w * (u \wedge v) = 0 \iff (w \cdot u) = (w \cdot v) = 0 \iff w \perp u, v.$$

Moreover, in three dimensions,

$$u \wedge v = \begin{vmatrix} 0 & u_1 v_2 - u_2 v_1 & u_1 v_3 - u_3 v_1 \\ u_2 v_1 - u_1 v_2 & 0 & u_2 v_3 - u_3 v_2 \\ u_3 v_1 - u_1 v_3 & u_3 v_2 - u_2 v_3 & 0 \end{vmatrix}$$

Thus, in three dimensions the entries of the wedge product matrix $u \wedge v$ are, up to sign, the same as the components of the cross product vector $u \times v$. This observation explains why wedge product and cross product share so many algebraic properties.

In three dimensions we are really very lucky. The matrix $u \wedge v$ is antisymmetric so, up to sign, it has only three unique entries. This property allows us to identify the matrix $u \wedge v$ with the vector $u \times v$. Nevertheless, there is something very peculiar about the vector $u \times v$. If u and v are orthogonal unit vectors, then the vectors $u, v, u \times v$ form a right-handed coordinate system. But if M is the linear transformation that mirrors vectors in the u, v plane, then $\{u \cdot M, v \cdot M, (u \times v) \cdot M\} = \{u, v, -u \times v\}$ forms a left-handed coordinate system. Thus, $(u \cdot M) \times (v \cdot M) \neq (u \times v) \cdot M$, so $u \times v$ does not really transform as a vector. This anomaly should alert us to the fact that cross product is not really a true vector. In fact, cross product transforms more like a tensor than a vector.

In higher dimensions we are not nearly so lucky. For example, in four dimensions the antisymmetric matrix $u \wedge v$ has, up to sign, six, not four, distinct entries. Thus, the matrix $u \wedge v$ cannot be identified with a four-dimensional vector. In n dimensions, the antisymmetric matrix $u \wedge v$ has $n(n-1)/2$ unique entries. But $n(n-1)/2 \neq n$ unless $n = 0,3$. Thus, only in three dimensions can we identify the wedge product of two vectors with a vector of the same dimension. In general, the wedge product is an antisymmetric 2-tensor. This antisymmetric tensor shares many of the important algebraic properties of the cross product, and thus it is a natural generalization of the cross product to four dimensions and beyond.

Acknowledgment

FACE-CONNECTED LINE SEGMENT GENERATION IN AN n-DIMENSIONAL SPACE

Didier Badouel and Charles A. Wüthrich
University of Toronto
Toronto, Ontario, Canada

In the early days of Computer Graphics, straight line rasterization was developed to render segments onto the raster plane. Later, three-dimensional segment discretization had to be developed to keep track of the path of a ray in the object space. These algorithms generate a connected sequence that represents the segment in the discrete space; moreover, they define a path in which the directions are uniformly distributed. An extension to higher-dimensional spaces is suited for applications ranging from line generation in a time–space coordinate system to the incremental generation of a discrete simultaneous linear interpolation of any number of variables.

This gem presents an algorithm that generates a face-connected line segment in discrete n-dimensional spaces. In two dimensions, the algorithm introduced below coincides with any classical 4-connected straight line drawing algorithm. Among all discrete segments joining two points, this algorithm produces one in which the directions are uniformly distributed. A definition of uniform distribution is given below.

Consider an n-dimensional lattice, or *hyperlattice,* i.e., the set of all points $P = (p_0, p_1, \ldots, p_{n-1})$ of $\mathbf{Z}^n$: Neighbourhood relations can be defined between the Voronoi *hypervoxel* associated with each point of the hyperlattice. In fact, only voxels having a *hyperface* in common, i.e., corresponding to hyperlattice points having $n-1$ coordinates in common, will be considered here as neighbours. In a two-dimensional lattice, such neighbourhood relation is the well-known 4-connection, while in the three-dimensional space it leads to 6-connection. The neighbourhood relations among the hyperlattice points introduce a rasterization proce-

dure for curves into the hyperlattice: A rasterization of a curve is in fact a path of neighbouring lattice points.

Consider two hyperlattice points $P = (p_0, p_1, \ldots, p_{n-1})$ and $Q = (q_0, q_1, \ldots, q_{n-1})$. Let $n_i = |q_i - p_i|$ Then a face-connected shortest path between P and Q requires $m = \Sigma n_i$ steps. The hyperline points are the points of coordinates $x_i = (q_i - p_i)t + p_i$, with $t \in [0, 1]$. The parameter t introduces an ordering on the points of the straight line. Consider the straight line points P_{i, h_i} obtained for $t = h_i/n_i$, where $h_j = 1, \ldots, n_i$, and order them in increasing order of their corresponding parameter value. Whenever an ambiguity occurs, and two parameter values h_j/n_i and h_j/n_j coincide, an arbitrary order has to be chosen. In other words, the segment PQ is subdivided into n_i parts for each dimension i, and the points obtained on the straight line segment are ordered by increasing values of the parameter t. When two subdivision points coincide, the one corresponding to the smaller dimension is considered to precede the other one.

The resulting set is a finite ordered set of the segment points P_{i, h_i}, which can be renamed as $A_0, A_1, \ldots, A_{m-1}$. Consider the finite path built by taking the sequence of directions $\{a_k\}_{k=0, \ldots, m-1}$, such that each direction a_k corresponds to the point $A_k = P_{a_k, l}$, for some l. Such a path is said to be *uniformly distributed* with respect to the directions that constitute it. It is clear that in such a path the occurrences of the different directions that have to appear in it are as evenly spaced as possible in the chain. Moreover, if we follow the previously defined path from the point P, the point Q shall be reached.

Whenever the hyperface-connected rasterization onto the n-dimensional hyperlattice of a straight line segment joining two hyperlattice points is computed, the result is a hyperface-connected path joining the two points. This path is uniformly distributed among all directions. A simplified version of the routing algorithm can be therefore summarized as follows. For each direction i, an integer counter d_i *is* used. In order to generate the straight line between the two points P and Q, the values of n_i are computed. Their least common multiple $l = \mathrm{LCM}(n_i)$ is evaluated,[1] and the values of $n''_i = l/n_i$ are computed. To obtain only integer

---

[1]In fact, either a low-complexity method in $O(n \log k)$ based on a table lookup or a simple common multiple can be used here.

computations, the values of $n'_{i} = 2n''_{i}$ are used. The cells d_i are initialized to the value $n'_i/2$. This initialization has to be made, otherwise the path generated corresponds to another rasterization scheme. At each step, n'_i is added to the di with the smallest value, and the ith signed direction is generated. The generation procedure is repeated until all d_i have reached the value $2l + n''_i$, which is equivalent to $\forall i, d_i \geq 2l$.

III

MODELING AND TRANSFORMATIONS

III
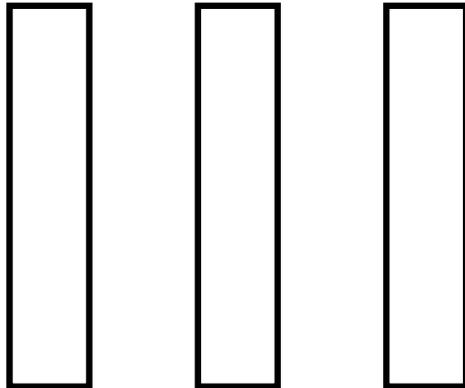
MODELING AND TRANSFORMATIONS

Most of the Gems in this section are concerned with transformations: how to compose, decompose, and manipulate them. The first Gem describes how to interpolate between two orientations using the quaternion representation, and adds the wrinkle of extra spins at the end of the interpolation. The fifth Gem is a useful companion to the first, in that it discusses issues relating to the choice of transformations upon which the interpolation occurs. The seventh Gem describes an alternative technique for interpolation, using Bézier curves.

The second and third Gems discuss how to decompose complex transformations into simpler components. The fourth and sixth Gems in this section describe two complementary techniques for producing rotations that are random in some sense, and are uniformly distributed in direction. These Gems are an improvement on a previous Gem.

The final Gem in this section is a solicited contribution that provides useful information for those interested in physically based modeling. This Gem provides closed form expressions for volume and mass properties for superquadric ellipsoids and toroids, and discusses the calculation and use of the inertia tensor.

QUATERNION INTERPOLATION WITH EXTRA SPINS

Jack Morrison
Digital Insight
Evergreen, Colorado

Quaternions are handy for representing the orientation or rotation of a 3-D object (Shoemake, 1985). The "Slerp" operation (spherical linear int*erp*olation) interpolates between two quaternions at a constant speed, using the most direct rotational path between the orientations. An animator may, however, want the interpolation to provide extra spins along the same path (complete revolutions about the same axis—see Fig. 1). This Gem gives a simple formula for doing this, derived with the help of Steven Gabriel.

Given two unit quaternions $\mathbf{A}$ and $\mathbf{B}$, and an interpolation parameter α ranging from 0 to 1, the basic Slerp equation is

$$\text{Slerp}(\mathbf{A},\ \mathbf{B};\ \alpha)\ =\ \mathbf{A}(\mathbf{A}^{-1}\ \mathbf{B})^{\alpha}.$$

An easier-to-implement equation uses the angle θ between the quaternions:

$$\theta\ =\ \text{acos}(\mathbf{A}\ \cdot\ \mathbf{B}),$$

$$\text{Slerp}(\mathbf{A},\ \mathbf{B};\ \alpha)\ =\ \frac{\sin\ (\theta\ -\ \alpha\theta)}{\sin\ \theta}\mathbf{A}\ +\ \frac{\sin\ (\alpha\theta)}{\sin\ \theta}\mathbf{B}.$$

To include k additional spins, determine

$$\phi\ =\ \theta\ +\ k\pi,$$
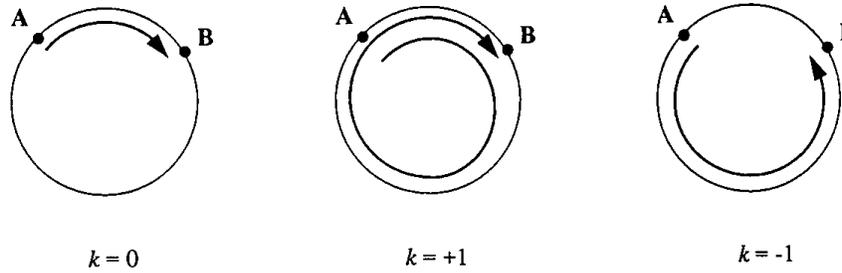
Figure 1.   Slerp (**A**, **B**) with *k* additional spins.

then

$$\text{Slerp}(\mathbf{A},\mathbf{B},k;\alpha) \ = \ \frac{\sin (\theta - \alpha\phi)}{\sin \theta}\mathbf{A} \ + \ \frac{\sin (\alpha\phi)}{\sin \theta}\mathbf{B}.$$

Figure 1 shows the effect of *k,* viewing orientations **A** and **B** along the rotation axis. For *k* = 0, *f* equals *q,* the equation reduces to the original form, and the interpolation follows the shortest circular path from **A** to **B**. For *k* = + 1, one full additional rotation is generated as a goes from 0 to 1. For *k* = – 1, the interpolation goes the "long" way around.

   The C implementation first checks if **A** and **B** are on opposite hemispheres (cos θ < 0). If so, the interpolation is modified to use the negation of **B** (which represents the same orientation as **B** itself), to ensure that the shortest rotation path is taken. It also checks whether **A** and **B** are the same, so that sin θ = 0, and no axis is defined for spinning. If **A** and **B** represent orientations 180 degrees apart, all rotation paths are the same length, but the quaternions still define an axis for spinning. Note that for a given **A**, **B**, and *k,* the quantities θ, ϕ, and sin θ could be computed once outside the interpolation loop for efficiency.

   See also G1, 498.

III.2

DECOMPOSING PROJECTIVE TRANSFORMATIONS

Ronald N Goldman
Rice University
Houston, Texas

Introduction

In another Gem (III.3) we show how to decompose linear and affine transformations of three-dimensional space into the products of scale, shears, rotations, translations, and projections. The goal of this gem is to demonstrate how to decompose an arbitrary projective transformation of three-dimensional space into simple, geometrically meaningful factors. For an alternative approach, see Thomas (1991).

Let us begin by recalling the difference between linear, affine, and projective transformations. Consider a point P with coordinates (x, y, z), and let P_{new} with coordinates $(x_{new}, y_{new}, z_{new})$ denote the new transformed point. Formulas that express the new coordinates $x_{new}, y_{new}, z_{new}$ in terms of the original coordinates x, y, z are given by the following equations:

Linear:	$x_{new} = ax + by + cz,$
Affine:	$x_{new} = ax + by + cz + d,$
Projective:	$x_{new} = (ax + by + cz + d)/(\alpha x + \beta y + \gamma z + \delta),$

with analogous formulas for y_{new} and z_{new}. Remember, too, that for projective transformations, the denominators of $x_{new}, y_{new}, z_{new}$ are identical.

Linear transformations are usually represented by 3 X 3 matrices L. A point $P = (x, y, z)$ transforms according to the formula

$$P_{new} = P * L,$$

where $*$ denotes matrix multiplication. Similarly, affine transformations are represented by 4×3 matrices

$$A = \begin{vmatrix} L \\ T \end{vmatrix}.$$

Here the upper 3×3 submatrix L represents a linear transformation and the last row T represents a translation. Let $P^* = (P, 1)$. Then a point P transforms according to the formula

$$P_{new} = P^* * A = P * L + T$$

Projective transformations can be represented by 4×4 matrices

$$M = \begin{vmatrix} L & N^t \\ T & d \end{vmatrix},$$

where the upper 3×3 submatrix L represents a linear transformation, the vector T represents a translation, and the column $(N\ d)^t$ *(the* superscript t denotes transpose) represents the projection. A homogeneous point $P^* = (P, 1)$ transforms according to the formula

$$P^*_{new} = P^* * M = (P^* * A)/(P \cdot N + d) = (P * L + T)/(P \cdot N + d),$$

where by convention $(x, y, z, w) = (x/w, y/w, z/w)$. That is, by convention, after performing the matrix multiplication, we divide by the homogeneous coordinate w_{new} to recover the actual point. Projective transformations are very important in computer graphics because perspective is a projective transformation (Goldman, 1990).

First Decomposition Algorithm—Perspective in Four Dimensions

The standard way to factor any projective transformation of three-dimensional space is first to embed three-space into four-space, next to apply the 4×4 matrix M as a linear transformation in four-space, then to

apply a perspective projection of four-space with the eye point at the origin and the perspective hyperplane at $w = 1$, and finally to project from four-space back to three-space by identifying the four-dimensional hyperplane $w = 1$ with standard three-dimensional space. That is, we map

$$R^3 \to R^4 \to R^4 \to R^3$$

by sending

$$P \to P^* = (P, 1) \to Q^* = (Q, 1) = (P, 1) * M * G \to Q,$$

where G is the perspective transformation of R^4 from the origin into the hyperplane $w = 1$.

From the point of view of computer graphics and geometric modeling, this decomposition is not very satisfactory. To complete this decomposition, we would need to factor an arbitrary transformation M of four-dimensional space into simple, geometrically meaningful, four-dimensional transformations. While it is possible to do so, this violates the spirit of what we are trying to accomplish. In computer graphics and geometric modeling, we generally apply modeling transformations to position an object in space, and then apply a perspective projection to position the object on the screen for viewing. All these transformations occur in three dimensions so we should never really need to discuss transformations of higher-dimensional space. It is these three-dimensional transformations that we would like to recapture in our decomposition of an arbitrary projective transformation.

Second Decomposition Algorithm
Affine * Projective

To obtain a better decomposition, we can factor any projective transformation by using the product formula

$$\begin{vmatrix} L & N^{t} \\ T & d \end{vmatrix} = \begin{vmatrix} L & 0 \\ T & 1 \end{vmatrix} * \begin{vmatrix} I & \Omega^{t} \\ 0 & d - T * \Omega^{t} \end{vmatrix},$$

where $L * \Omega^t = N^t$ This factoring certainly works when L *is* nonsingular, since then we can simply take $\Omega = N*(L^{-1})^t$, but it also works even if L is singular provided only that N is in the image of L^t. Now the first factor is simply the affine transformation given by the linear transformation L followed by the translation T, and the second factor is a pure projective transformation. We know from Gem III.3 how to factor any affine transformation of three-dimensional space into simple, geometrically meaningful factors. Thus, if N is in the image of L^t, we can use this product formula to decompose a projective transformation into simple, geometrically meaningful factors, followed by a pure projective transformation.

What is the effect of the pure projective transformation? Consider the pure projective transformation matrix

$$M* = \begin{vmatrix} I & \Omega^t \\ 0 & d* \end{vmatrix}.$$

By convention, $M*$ transforms a point according to the formula

$$P_{new} = P/(P \cdot \Omega + d*).$$

Thus, any point P satisfying the equation

$$P \cdot \Omega + d* = 1$$

is left invariant by the transformation $M*$. But this equation is linear, so it defines a plane. Thus, the projective transformation $M*$ leaves a plane of points unmoved. Moreover, if S is the plane defined by the linear equation

$$P \cdot \Omega + d* = 0,$$

then $M*$ maps S to infinity. Finally, for an arbitrary point P

$$(P \cdot \Omega + d *) = \text{signed distance to the plane } S.$$

Thus, the pure projective transformation $M*$ maps planes at a fixed

distance from S into other planes at a new fixed distance from S. Note, too, that the plane at infinity ($w = 0$) is mapped to a finite plane.

From the perspective of computer graphics and geometric modeling, this decomposition of an arbitrary projective transformation into the product of a purely affine and a purely projective transformation is still not entirely satisfactory. In computer graphics, affine modeling transformations are usually followed by perspective projections. Thus, we would like, whenever possible, to factor a projective transformation into the product of an affine transformation and a perspective projection.

In one special case we do actually achieve this goal. Observe that if $d^* = 0$, then $P_{new} \cdot \Omega = 1$. Thus, P_{new} always lies on the invariant plane of M^*. In this case, M^* *is* actually a perspective projection with the eyepoint at the origin and the perspective plane given by the linear equation $P \cdot \Omega = 1$. Thus, if $N^t = L * \Omega^t$ (that is, if N is in the image of L^t, e.g., L *is* nonsingular) and $d = * \Omega^t$, then our product formula actually give us the decomposition that we desire, since the second factor is a perspective projection. More generally, we need to decide when a projective transformation M has a perspective factor M^*. One hint we can apply here is that if $d^* = 0$, then $\text{Det}(M^*) = 0$.

Third Decomposition Algorithm— Perspective * Affine

If a projective transformation has a perspective factor, then it must be a singular matrix. This is easy to see because every perspective transformation M has an eyepoint E that is mapped to a singularity—that is, to the point with homogeneous coordinates $(0, 0, 0, 0)$. Thus,

$$E * M = 0,$$

so the eyepoint E is an eigenvector of the matrix M corresponding to the eigenvalue 0. Thus, M must be singular. We shall show that if L is nonsingular, then the converse is also true. That is, if M is a singular 4×4 matrix whose upper 3×3 submatrix L is nonsingular, then M can be factored into the product of a perspective projection and an affine transformation.

Suppose, then, that we have a singular 4×4 matrix

$$M = \begin{vmatrix} L & N^t \\ T & d \end{vmatrix}$$

representing a projective transformation, and that the linear transformation L is nonsingular. Since M is singular, $\mathrm{Det}(M) = 0$. Therefore, one of its eigenvalues is 0. Let E be a nonzero eigenvector corresponding to the eigenvalue 0. Since L is nonsingular, E cannot lie at infinity—that is, $E \neq (e_1, e_2, e_3, 0)$—otherwise, L would also have a nonzero eigenvector corresponding to the eigenvalue 0. We will use E as the eyepoint of the perspective projection.

To complete the definition of the perspective transformation, we also need the perspective plane. Recall that by convention a point P is mapped by the projective transformation M to the point

$$P_{new} = (P * L + T)/(P \cdot N + d).$$

Thus, points on the plane S defined by the linear equation

$$P \cdot N + d = 1$$

are not affected by the projective part of the transformation. Let R be the perspective projection defined by the eyepoint E and the perspective plane S, and let A be the affine transformation defined by the linear transformation L and the translation T. Then we shall show that

$$M = R * A.$$

We can verify that this equation is valid by checking that it holds for all points. If P is in S, then $P * R = P$ and $P \cdot N + d = 1$, so

$$P * M = (P * L + T) = P * A = P * (R * A).$$

If P is not in S, then the line from the eyepoint E to the point P intersects the plane S in a unique point Q so

$$P = \lambda Q + (1 - \lambda)E.$$

Therefore, because E is an eigenvector of M corresponding to the eigenvalue 0,

$$P^* * M = \{\lambda Q^* + (1 - \lambda)E^*\} * M$$

$$= \lambda(Q^* * M)$$

$$= \lambda(Q * L + T)/\lambda(Q \cdot N + d)$$

$$= (Q * L + T)$$

$$= Q^* * A$$

$$= P^* * (R * A)$$

The last equality holds because P lies on the line joining Q and E. Therefore, the perspective projection R maps P^* to Q^*.

   Thus, we have succeeded in factoring a singular projective transformation M into the product of a perspective transformation R and an affine transformation A. The matrix for the perspective transformation R can be found explicitly from the eyepoint E and the plane S by the methods described in Goldman (1990), and the affine transformation A can be factored further into simple, geometrically meaningful, factors by the techniques described in Gem 3.3. Thus, we have succeeded in decomposing a singular projective transformation into simple, geometrically meaningful factors.

   Still, this factoring is not quite satisfactory, since in geometric modeling the perspective transformation comes last rather than first. Therefore, let us try to reverse the order of our factors.

Fourth Decomposition Algorithm— Affine * Perspective

Consider again a singular 4×4 matrix

$$M = \begin{vmatrix} L & N^{\mathrm{t}} \\ T & d \end{vmatrix}$$

representing a projective transformation where the linear transformation L is nonsingular. Again, let E be a nonzero eigenvector of M corresponding to the eigenvalue 0, and let S be the plane defined by the linear equation

$$P \cdot N + d = 1.$$

Furthermore, let A be the affine transformation defined by the linear transformation L and the translation T, and let R be the perspective projection defined by the eyepoint $A(E)$ and the perspective plane $A(S)$. We shall show that

$$M = A * R.$$

Before we proceed, notice that for the perspective transformation R to be well defined, the perspective plane $A(S)$ cannot collapse to a line or a point and the eyepoint $A(E)$ cannot lie in the perspective plane $A(S)$. This will certainly be true if A, or equivalently L, is nonsingular, as is generally the case in computer graphics and geometric modeling applications. Recall, too, that we need this assumption anyway to insure that the eyepoint $A(E)$ does not lie at infinity.

We can verify that this new factoring of M *is* valid by again checking that it holds for all points. If P is in S, then $P \cdot N + d = 1$, so

$$P * M = (P * L + T) = P * A = P * A * R,$$

where the last equality holds because R is invariant on $A(S)$. On the other hand, if P is not in S, then the line joining the point E to the point P intersects the plane S in a unique point Q so

$$P = \lambda Q + (1 - \lambda)E.$$

Therefore, because E is an eigenvector of M corresponding to the eigenvalue 0,

$$
\begin{aligned}
P^* * M &= \{\lambda Q^* + (1 - \lambda)E^*\} * M \\
&= \lambda(Q^* * M) \\
&= \lambda(Q * L + T)/\lambda(Q \cdot N + d) \\
&= (Q * L + T) \\
&= Q^* * A \\
&= P^* * (A * R).
\end{aligned}
$$

The last equality holds because *P* lies on the line joining *Q* and *E, so* A(*P**) lies on the line joining A(*Q**) and A(*E**). Therefore, the perspective projection *R* maps A(*P**) to A(*Q**).

Summary

To summarize: We have four ways of decomposing a projective transformation

$$M = \begin{vmatrix} L & N^t \\ T & d \end{vmatrix}.$$

1. $M = M * G$: as the product of a linear transformation *M* of four-dimensional space and a perspective transformation *G* from the origin into the hyperplane $w = 1$.

2.
$$\begin{vmatrix} L & N^t \\ T & d \end{vmatrix} = \begin{vmatrix} L & 0 \\ T & 1 \end{vmatrix} * \begin{vmatrix} I & \Omega^t \\ 0 & d - T \cdot \Omega^t \end{vmatrix},$$

   where $L * \Omega^t = N^t$: as the product of an affine transformation and a pure projective transformation. This works provided *N* is in the image of L^t. In particular, this decomposition is valid when *L* is nonsingular. Moreover, if $d = 1 * \Omega^t$, then the second factor is the perspective projection from the origin to the plane $P \cdot \Omega = l$.

3. $M = R * A$: as the product of a perspective projection *R* followed by an affine transformation *A*. Here *R* is the perspective projection defined by the eyepoint *E*, where *E* is a nonzero eigenvector of *M* corresponding to the eigenvalue 0, and the perspective plane *S* consisting of points *P* that satisfy the linear equation $P \cdot N + d = 1$, and *A* is the affine transformation defined by the linear transformation *L* and the translation *T*. This decomposition is valid whenever the matrix *M* is singular and the matrix *L is* nonsingular—that is, whenever $Det(M) = 0$ and $Det(L) \neq 0$.

4. $M = A * R$: as the product of an affine transformation *A* followed by a perspective projection *R*. Here *A* is the affine transformation defined by the linear transformation *L* and the translation *T*, and *R*

is the perspective transformation defined by the eyepoint $A(E)$, where E *is* a nonzero eigenvector of M corresponding to the eigenvalue 0, and the perspective plane $A(S)$, where S is the plane of points P that satisfy the linear equation $P \cdot N + d = 1$. This decomposition is valid whenever the matrix M is singular, the plane $A(S)$ does not collapse to a line or a point, and the point $A(E)$ does not lie in the plane $A(S)$ or at infinity. In particular, this works whenever $Det(M) = 0$ and $Det(L) \neq 0$.

This last case is the standard case in computer graphics and geometric modeling. Thus, in the standard case we can decompose a projective transformation into the product of a non-singular affine transformation followed by a perspective projection. By Gem III.3, we can further factor the affine transformation into the product of three scales, two shears, one rotation, and one translation.

Although we have succeeded in factoring an arbitrary, singular, projective transformation, notice again that this decomposition is not unique since the factoring of the affine transformation is itself not unique. Nevertheless these factoring procedures are still useful because they allow us to decompose singular projective transformations into simple, geometrically meaningful factors.

See also G2, 319; G2, 320; G3, C.3.

DECOMPOSING LINEAR AND AFFINE TRANSFORMATIONS

Ronald N. Goldman
Rice University
Houston, Texas

Goal

Every nonsingular linear transformation of three-dimensional space is the product of three scales, two shears, and one rotation. The goal of this Gem is to show how to decompose any arbitrary, singular or nonsingular, linear or affine transformation of three-dimensional space into simple, geometrically meaningful, factors. For an alternative approach to similar problems (see Thomas, 1991).

Nonsingular Linear Transformations

Linear transformations of three-dimensional space are generally represented by 3×3 matrices. To decompose an arbitrary nonsingular linear transformation, consider, then, an arbitrary nonsingular 3×3 matrix L. We shall show that L can be factored into the product of three scales, two shears, and one rotation matrix.

Let the rows of L be given by the vectors u, v, w. Since the matrix L is nonsingular, the vectors u, v, w are linearly independent. Therefore, using the Gram-Schmidt orthogonalization procedure, we can generate 3 orthonormal vectors u^*, v^*, w^* by setting

$$u^* = u/|u|,$$

$$v^* = (v - (v \cdot u^*)u^*)/|v - (v \cdot u^*)u^*|,$$

$$w^* = (w - (w \cdot u^*)u^* - (w \cdot v^*)v^*)/|w - (w \cdot u^*)u^* - (w \cdot v^*)v^*|.$$

This orthogonalization procedure can be used to decompose the matrix L into the desired factors.

Begin with the rotation. By construction, the matrix R whose rows are u^*, v^*, w^* is an orthogonal matrix. If $\text{Det}(R) = -1$, replace w^* by $-w^*$. Then R is the rotation matrix we seek. Using the results in Goldman (1991a), we can, if we like, retrieve the rotation axis and angle from the matrix R.

The three scaling transformations are also easy to find. Let

$$s_1 = |u|,$$

$$s_2 = |v - (v \cdot u^*)u^*|,$$

$$s_3 = |w - (w \cdot u^*)u^* - (w \cdot v^*)v^*|.$$

That is, s_1, s_2, s_3 are the lengths of u^*, v^*, w^* before they are normalized. Now let S be the matrix with s_1, s_2, s_3 along the diagonal and with zeroes everywhere else. The matrix S represents the three independent scaling transformations that scale by s_1 along the x-axis, s_2 along the y-axis, and s_3 along the z-axis. (If $\text{Det}(R)$ was originally -1, then replace s_3 by $-s_3$. In effect, this mirrors points in the xy-plane.)

Before we can introduce the two shears, we need to recall notation for the identity matrix and the tensor product of two vectors.

Identity:

$$I = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}.$$

Tensor Product:

$$v \otimes \omega = \begin{vmatrix} v_1\omega_1 & v_1\omega_2 & v_1\omega_3 \\ v_2\omega_1 & v_2\omega_2 & v_2\omega_3 \\ v_3\omega_1 & v_3\omega_2 & v_3\omega_3 \end{vmatrix} = \begin{vmatrix} v_1 \\ v_2 \\ v_3 \end{vmatrix} * |\omega_1\omega_2\omega_3| = v^t * \omega.$$

Here $*$ denotes matrix multiplication and the superscript t denotes

transpose. Observe that for all vectors μ

$$\mu \cdot I = \mu,$$

$$\mu * (v \otimes \omega) = (\mu \cdot v)\omega.$$

Now we are ready to define the two shears. Recall from Goldman (199lb) that a shear H is defined in terms of a unit vector v normal to a plane Q, a unit vector ω in the plane Q, and an angle ϕ by setting

$$H = I + \tan\phi(v \otimes \omega).$$

Let H_1 be the shear defined by unit normal v^*, the unit vector u^*, and the angle θ by setting

$$H_1 = I + \tan\theta(v^* \otimes \omega^*),$$

$$\tan\theta = v \cdot u^*/s_2.$$

Similarly, let H_2 be the shear defined by unit normal w^*, the unit vector r^*, and the angle ψ by setting

$$H_2 = I + \tan\psi(w^* \otimes r^*),$$

$$\tan\psi = SQRT\{(w \cdot u^*)^2 + (w \cdot v^*)^2\}/s_3,$$

$$r^* = \{(w \cdot u^*)u^* + (w \cdot v^*)v^*\}/s_3\tan\psi$$

Then it is easy to verify that

$$u^* * H_1 = u^*, \quad v^* * H_1 = v^* + (v \cdot u^*/s_2)u^*, \quad w^* * H_1 = w^*,$$

$$u^* * H_2 = u^*, \quad v^* * H_2 = v^*,$$

$$w^* * H_2 = w^* + \{(w \cdot u^*)u^* + (w \cdot v^*)v^*\}/s_3.$$

Finally, we shall show that

$$L = S * R * H_1 * H_2.$$

Since the transformation L is linear, we need only check that both sides give the same result on the canonical basis $\mathbf{i}$, $\mathbf{j}$, $\mathbf{k}$. By construction we know that

$$\mathbf{i} * L = u, \qquad \mathbf{j} * L = v, \qquad \mathbf{k} * L = w,$$

so we need to verify that we get the same results for the right-hand side. Let us check.

First,

$$\mathbf{i} * S * R * H_1 * H_2 = (s_1)\mathbf{i} * R * H_1 * H_2$$

$$= (s_1)u^* * H_1 * H_2$$

$$= u.$$

since by construction the two shears H_1 and H_2 do not affect u^*.

Next,

$$\mathbf{j} * S * R * H_1 * H_2 = (s_2)\mathbf{j} * R * H_1 * H_2$$

$$= (s_2)v^* * H_1 * H_2$$

$$= \{s_2 v^* + (v \cdot u^*)u^*\} * H_2$$

$$= s_2 v^* + (v \cdot u^*)u^*$$

$$= v.$$

Finally,

$$\mathbf{k} * S * R * H_1 * H_2 = (s_3)\mathbf{k} * R * H_1 * H_2$$

$$= (s_3)w^* * H_1 * H_2$$

$$= (s_3)w^* * H_2$$

$$= s_3 w^* + (w \cdot u^*)u^* + (w \cdot v^*)v^*$$

$$= w.$$

Although we have succeeded in factoring an arbitrary nonsingular linear transformation, notice that this decomposition is not unique. Indeed, the Gram-Schmidt orthogonalization procedure depends upon the ordering of the vectors to which it is applied. We could, for example, have applied the Gram-Schmidt procedure to the vectors in the order *w, u, v* instead of *u, v, w*. We would then have retrieved a different decomposition of the same matrix. Nevertheless, this procedure is still of some value since it allows us to decompose an arbitrary nonsingular linear transformation into simple, geometrically meaningful factors.

Singular Linear Transformations

Now let *L* be an arbitrary singular 3×3 matrix. There are three cases to consider, depending on the rank of *L*. If rank(*L*) = 0, there is nothing to do since *L* simply maps all vectors into the zero vector. The case rank(*L*) = 1 is also essentially trivial, since all vectors are simply appropriately scaled and then projected onto a single fixed vector. Therefore, we shall concentrate on the case where rank(*L*) = 2.

We will show that when rank(*L*) = 2, we still need one rotation, but we require only two scales, one shear, and one parallel projection. Thus, the number of scales is reduced by one and a shear is replaced by a parallel projection. Moreover, we shall shovv that the parallel projection can be replaced by a shear followed by an orthogonal projection.

Again, let the rows of *L* be given by the vectors *u, v, w*. Since the matrix *L* is singular, the row vectors *u, v, w* are linearly dependent, but since rank(*L*) = 2, two rows of *L* are linearly independent. For simplicity and without loss of generality, we will assume that *u* and *v* are linearly independent.

Modifying the Gram-Schmidt orthogonalization procedure, we can generate three orthonormal vectors *u*, v*, w** by setting

$$u^* = u/|u|,$$

$$v^* = (v - (v \cdot u^*)u^*)/|v - (v \cdot u^*)u^*|,$$

$$w^* = u^* \times v^*.$$

This orthogonalization procedure can again be used to decompose the matrix L into the desired factors.

By construction, the matrix R whose rows are u^*, v^*, w^* is an orthogonal matrix and $\text{Det}(R) = 1$. The matrix R is the rotation matrix that we seek. We can recover the axis and angle of rotation from the matrix R using the techniques described in Goldman (1991a).

The two scaling transformations are also easy to find. Let

$$s_1 = |u|,$$

$$s_2 = |v - (v \cdot u^*)u^*|.$$

Note that s_1, s_2 are, respectively, the lengths of u^*, v^* before they are normalized. Now let S be the matrix with s_1, s_2, 1 along the diagonal and with zeroes everywhere else. The matrix S represents the two independent scaling transformations that scale by s_1 along the x-axis and s_2 along the y-axis.

The shear H is the same as the first of the two shears that we used to decompose a nonsingular linear transformation. Using the notation for the identity matrix and the tensor product of two vectors that we established above,

$$H = I + \tan\theta(v^* \otimes u^*),$$

$$\tan\theta = v \cdot u^*/s_2.$$

Thus, H is the shear defined by the unit normal vector v^*, the unit vector u^*, and the angle θ. Again, it is easy to verify that

$$u^* * H = u^*, \qquad v^* * H = v^* + (v \cdot u^*/s_2)u^*, \qquad w^* * H = w^*.$$

Last, we define the parallel projection P to be projection into the u^*v^*-plane parallel to the vector$(w^* - w)$. According to Goldman(1990), the matrix P is given by

$$P = I - w^* \otimes (w^* - w).$$

Notice that if $w = 0$, this parallel projection reduces to orthogonal projection into the u^*v^*-plane (Goldman, 1990). In any event, it is easy

to verify that

$$u^* * P = u^*, \qquad v^* * P = v^*, \qquad w^* * P = w.$$

Finally, let us show that

$$L = S * R * H * P$$

by checking that both sides give the same result on the canonical basis $\mathbf{i}$, $\mathbf{j}$, $\mathbf{k}$. By construction we know that

$$\mathbf{i} * L = u, \qquad \mathbf{j} * L = v, \qquad \mathbf{k} * L = w,$$

so we need to verify that we get the same results for the right-hand side. Let us check.
    First,

$$\mathbf{i} * S * R * H * P = (s_1)\mathbf{i} * R * H * P$$
$$= (s_1)u^* * H * P$$
$$= s_1 u^*$$
$$= u,$$

since by construction the two linear transformations H and P do not affect u^*.
    Next,

$$\mathbf{j} * S * R * H * P = (s_2)\mathbf{j} * R * H * P$$
$$= (s_2)v^* * H * P$$
$$= \{s_2 v^* + (v \cdot u^*)u^*\} * P$$
$$= s_2 v^* + (v \cdot u^*)u^*$$
$$= v.$$

Finally,

$$\mathbf{k} * S * R * H * P = \mathbf{k} * R * H * P$$
$$= w^* * H * P$$
$$= w^* * P$$
$$= w.$$

By the way, every parallel projection can be written as the product of a shear followed by an orthogonal projection. To see this, recall that a projection P parallel to the vector ω into the plane with normal vector n is given by Goldman (1990):

$$P = I - (n \otimes \omega)/n \cdot \omega.$$

Consider the orthogonal projection O into the plane perpendicular to the unit vector n (Goldman, 1990),

$$O = I - (n \otimes n),$$

and the shear K defined by the unit normal vector n, the unit vector $v = (n - \omega/\omega \cdot n)/|n - \omega/\omega \cdot n|$, and the angle θ given by $\tan\theta = |n - \omega/\omega \cdot n|$:

$$K = I + \tan\theta(n \otimes v).$$

Since $v \cdot n = 0$, it follows that $(n \otimes v) * (n \otimes n) = (v \cdot n)(n \otimes n) = 0$. Therefore,

$$I - (n \otimes \omega)/n \cdot \omega = \{I + \tan\theta(n \otimes v)\} * \{I - (n \otimes n)\},$$

or, equivalently,

$$P = K * O$$

In our case $\omega = w^* - w$, $n = w^*$, and $v = w$. Thus, we have shown that when rank(L) = 2, we can factor L into the product of two scales one rotation, two shears, and one orthogonal projection. This decomposition is the same as in the nonsingular case, except that the number of scales is reduced by one and the standard nonsingular factors—scales, rotation, and shears—are followed by an orthogonal projection.

Notice again that this decomposition is not unique. Indeed, the modi-fied Gram-Schmidt orthogonalization procedure also depends upon the ordering of the vectors to which it is applied. We could, for example, have applied the Gram-Schmidt procedures to the vectors in the order *v, u* instead of *u, v*. We would then have retrieved a different decomposi-tion of the same matrix. Nevertheless, this procedure is still of some value since it allows us to decompose an arbitrary singular linear transforma-tion into simple, geometrically meaningful factors.

Affine Transformations

Finally, recall that every affine transformation *A* is simply a linear transformation *L* followed by a translation *T*. If the affine transformation is represented by a 4×3 matrix

$$A = \left| \begin{matrix} L \\ T \end{matrix} \right|,$$

then the upper 3×3 submatrix *L* represents the linear transformation and the fourth row *T* represents the translation vector. Thus, to decom-pose an arbitrary affine transformation into simpler, geometrically mean-ingful factors, simply factor the associated linear transformation *L* and append the translation *T*. Thus, every nonsingular affine transformation of three-dimensional space can be factored into the product of three scales, two shears, one rotation, and one translation. Similarly, every singular affine transformation of three-dimensional space can be factored into the product of two scales, two shears, one rotation, one orthogonal projection, and one translation. Again, these decompositions are not unique, since the decompositions of the associated linear transformations are not unique. Nevertheless, these procedures are still of some value since they allow us to decompose arbitrary affine transformations into simple, geometrically meaningful factors.

See also G2, 319; G2, 320; G3, C.2.

III.4

FAST RANDOM ROTATION MATRICES

James Arvo
Cornell University
Ithaca, New York

In a previous Gem (Arvo, 1991), I described a method for generating random rotation matrices based on random unit quaternions. As Ken Shoemake points out in his Gem (III.6) that algorithm was flawed in that it did not generate *uniformly distributed* rotation matrices. For a method based on quaternions that corrects this defect see his algorithm. In this Gem I describe a completely different approach to solving the same problem that has the additional benefit of being slightly faster than the previous method. The approach is based on the following fact:

> *To generate uniformly distributed random rotations of a unit sphere, first perform a random rotation about the vertical axis, then rotate the north pole to a random position.*

The first step of this prescription is trivial. Given a random number, x_1, between 0 and 1. the matrix R does the trick:

$$R = \begin{bmatrix} \cos(2\pi x_1) & \sin(2\pi x_1) & 0 \\ -\sin(2\pi x_1) & \cos(2\pi x_1) & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{1}$$

Here we are assuming that the z-axis is the vertical axis, so the "north pole" will be the point $z = (0, 0, 1)$. The second operation is not quite so obvious, but fortunately it can be carried out quite efficiently. Observe that we can take the point z to any other point p on the sphere via a

reflection through the plane orthogonal to the line $\overline{zp}$ and containing the origin. Such a reflection is given by the *Householder matrix*

$$H = I - 2vv^{\mathrm{T}}, \tag{2}$$

where v is a unit vector parallel to $\overline{zp}$ (see, for instance, Golub and Van Loan, 1985). To turn this into a rotation we need only apply one more reflection (making the determinant positive). A convenient reflection for this purpose is reflection through the origin—that is, scaling by –1. Thus, the final rotation matrix can be expressed as the product

$$M = -HR, \tag{3}$$

where R is the simple rotation in Eq. (1). The rotation matrix M will be uniformly distributed within SO(3), the set of all rotations in three-space, if H takes the north pole to every point on the sphere with equal probability density. This will hold if the image of z under the random reflection is such that both its azimuth angle and the cosine of its elevation angle are uniformly distributed. The matrix H in Eq. (2) will satisfy these requirements if we let

$$v = \begin{bmatrix} \cos(2\pi x_2)\sqrt{x_3} \\ \sin(2\pi x_2)\sqrt{x_3} \\ \sqrt{1 - x_3} \end{bmatrix}, \tag{4}$$

where x_2 and x_3 are two independent uniform random variables in [0,1]. To show this we need only compute $p = Hz$ and verify that p is distributed appropriately. Using the preceding definition of v, we have

$$p = z - 2vv^{\mathrm{T}}z = \begin{bmatrix} -2\cos(2\pi x_2)\sqrt{x_3(1 - x_3)} \\ -2\sin(2\pi x_2)\sqrt{x_3(1 - x_3)} \\ 2x_3 - 1 \end{bmatrix}. \tag{5}$$

Because the third component of p is the cosine of its elevation angle, we see immediately that it is uniformly distributed over [–1, 1], as required.

random_rotation(x_1, x_2, x_2, M)

    x_1, x_2, x_3 : **real**;   *Three random variables.*

    M : **matrix3**;   *The resulting matrix.*

begin

    $\theta \leftarrow 2\pi x_1$;   *Pick a rotation about the pole.*

    $\phi \leftarrow 2\pi x_2$;   *Pick a direction to deflect the pole.*

    $z \leftarrow x_3$;   *Pick the amount of pole deflection.*

    Construct a vector for performing the reflection.

$$V \leftarrow \begin{bmatrix} \cos\phi\sqrt{z} \\ \sin\phi\sqrt{z} \\ \sqrt{1-z} \end{bmatrix}$$

    Construct the rotation matrix by combining two
    simple rotations: first rotate about the Z axis,
    then rotate the Z axis to a random orientation.

$$M \leftarrow (2VV^\mathsf{T} - I) \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

    end

Figure 1. An efficient procedure for creating random 3×3 rotation matrices.

Similarly, from its first two components we see that the azimuth angle of p is $2\pi x_2$, which is uniformly distributed over $[0, 2\pi]$.

The complete algorithm combining the reflection and simple rotation is shown in Fig. 1, and an optimized version in C appears in the appendix. Procedure "random_rotation" requires three uniformly distributed random numbers between 0 and 1. Supplying these values as arguments has several advantages. First, the procedure can be used in conjunction with your favorite pseudorandom number generator, and there are a great many to choose from. Secondly, if we obtain the three random numbers by *stratified* or *jittered* sampling of the unit cube, the resulting rotation
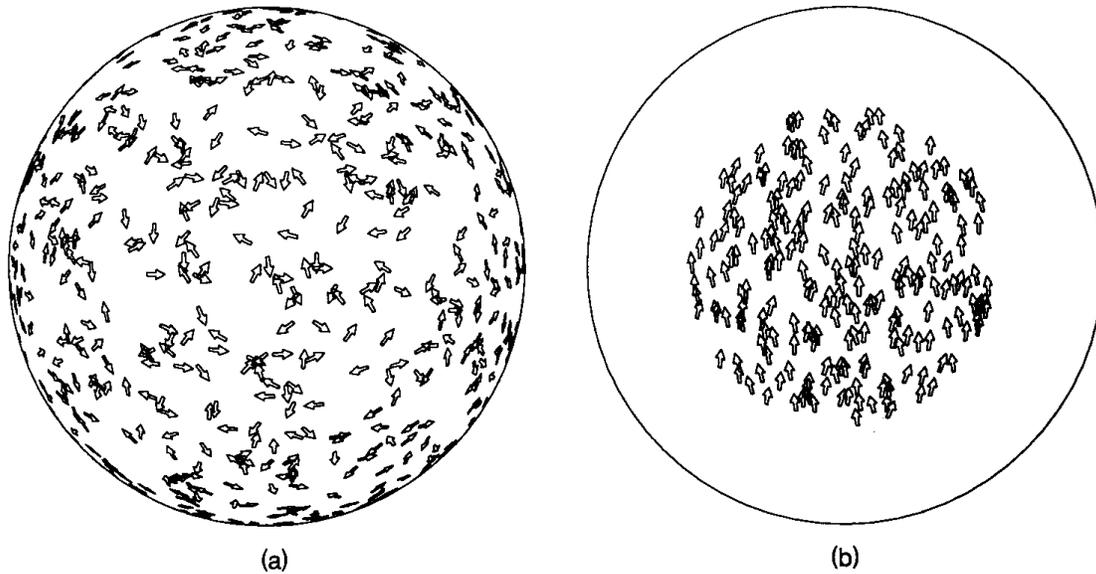
Figure 2.

matrices will inherit the benefits—namely, less clumping. Finally, if we restrict the range of the random input variables (while maintaining their uniformity), we can generate uniformly distributed perturbations or "wobbles" within given limits.

Figure 2a shows the result of applying 1,000 random rotations to a sphere with an arrow painted at one pole. The resulting pattern looks much the same from any vantage point, providing visual confirmation of uniformity. Figure 2b was generated by restricting x_1 and x_3 to the range [0, 0.1].

See also G2, 355; G3, C.6.

III.5
ISSUES AND TECHNIQUES FOR KEYFRAMING TRANSFORMATIONS

Paul Dana
Shadow Graphics
Madison, Alabama

Introduction

Spencer, (1991) explains how to express a matrix as a set of parameters, called an *unmatrix.* In Spencer's unmatrix, each parameter is a single value that can be interpolated using linear interpolation or piecewise splining techniques. When simple interpolation is used to calculate transformations between samples, unnatural motion can occur.

This Gem shows how to interpolate motion naturally between two sample matrices for use in motion control or key-framing applications. This technique was used to create the key frame animation system described in Dana (1991).

Interpolating in Logarithmic Space

Unnatural or accelerated motion occurs when scaling parameters are interpolated in a straightforward way. A simple interpolation, halfway between a scaling parameter of 0.50 and 2.0, produces a value of 1.25 when the visually *expected* value would be 1.0.

A solution to this problem is to interpolate the logarithm of the scaling parameter.

Relative Motion

The most natural motion from one sample transformation to another is the one that takes the *shortest* apparent path. When rotation parameters are interpolated, sometimes the shortest path may cross the zero degree mark. For example, the shortest path between 10 degrees and 350 degrees is –20 degrees, not +340 degrees.

A second rotation problem occurs when you want an object to appear to rotate relative to its own oriental ion instead of relative to its current orientation in the 3-D universe.

To get the desired transformation and to solve both the zero mark problem and the axis problem:

1. Express the motion between two samples *relative to the first sample,* prior to interpolation.

2. Calculate the interpolation using an identity transformation and the difference between the two samples.

3. Concatenate the interpolated transformation to the first sample.

This solves the zero mark problem because the rotational values of an identity transformation are all zero. It also solves the axis problem by expressing a segment of motion relative to the segment's first sample.

Linear vs. Splined Interpolation

Although it might seem best to use a splining technique to interpolate all the parameters of an unmatrix, experience has shown that ordinary linear interpolation is best for the scaling, shearing, rotation, and perspective parameters, and splined interpolation is best for the translation parameters.

Subdividing Motion

For a motion to have constant speed between two samples, the motion must be subdivided into intervals of equal length in space, not just duration in time. When using splined interpolation to interpolate translation parameters, the spline type must allow for even subdivision along the length of a piece. A spline, such as a Cardinal spline, should be used.

See also G3, C.7.

UNIFORM RANDOM ROTATIONS

Ken Shoemake
Otter Enterprises
Palo Alto, California

Background

A previous Graphics Gem (Arvo, 1991) presented an algorithm for generating random rotations, in both quaternion and matrix form. It takes as input three uniform deviates and efficiently computes a random rotation with a uniformly distributed axis and a uniformly distributed angle. The purpose of the present gem is to demonstrate that, surprisingly, that algorithm does *not* generate a uniformly distributed rotation, and to give two simple algorithms that do.

How can the distribution of the axis and angle be uniform, and yet the distribution of the rotations not be? To answer that question requires, first of all, a definition of uniformity. Since rotations form a group, an approach that is both standard and intuitively satisfying uses "Haar measure" as follows: If X is a random rotation with uniform distribution, then for any fixed but arbitrary rotation R, RX and XR have the same distribution as X. A rough physical analogy is the testing of a bicycle wheel for balance by spinning it and looking for wobbles, or dragging a flat edge across freshly poured cement to smooth it.

Planar Rotations

Before examining the implications of this definition for spatial rotations, let us first examine its application to the simpler case of planar rotations, where we can use both our eyes and our intuition. A planar rotation can be represented in several ways, for example as an angle between 0 and

2π, as a unit complex number $x + iy = \cos\theta + i\sin\theta$, and as a matrix

$$\begin{bmatrix} x & y \\ -y & x \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

Planar rotations combine by summing their angles modulo 2π, so one way to generate a uniform planar rotation is to generate a uniform angle. Combination of X with R merely slides the angle around. Since this leaves the distribution unchanged, it is uniform. A more geometrical interpretation of this problem is that we want to generate points uniformly distributed on the unit circle $x^2 + y^2 = 1$, with probability proportional to arc length. Note that the average magnitude of x will be $1/\pi$ times the integral of $2\cos\theta$ from 0 to $\pi/2$, namely $2/\pi \approx 0.6366$. This computation is merely "summing"—integrating—the x values of all the points on the right half of the circle and dividing by the "number of points" used—the arc length of that half of the circle. Here and subsequently we take advantage of circle (later, sphere) and sinusoid symmetries when computing magnitudes. Now suppose a rotation is generated by choosing a uniformly distributed x between -1 and $+1$, with y
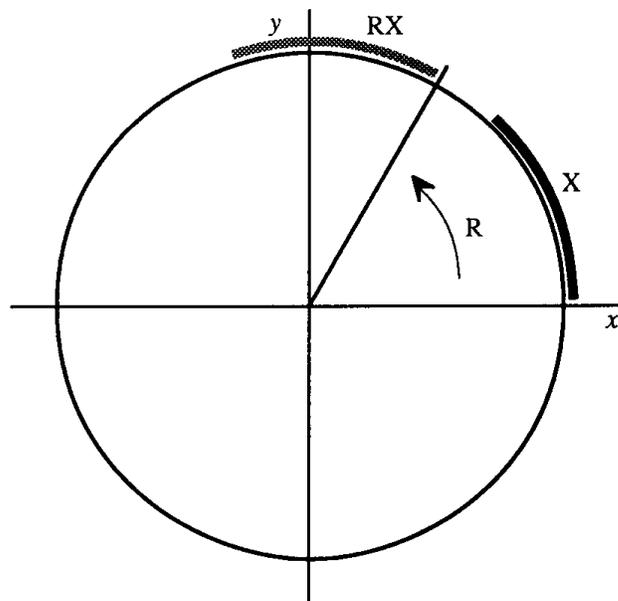

Figure 1. Haar test reveals distribution lumps.

computed as $\pm\sqrt{1 - x^2}$ (either sign being equally likely). For this distribution the average magnitude of x will be $\frac{1}{2}$, and so it cannot give uniformly distributed rotations.

Uniform Spherical Distribution

Shortly, we are going to want to know about points uniformly distributed on a sphere, and on a hypersphere. The sphere case is easier to visualize, and has a surprising result: Each coordinate of a point uniformly distributed on a sphere is uniformly distributed! Thus, for example, the average magnitude of the x coordinate is $\frac{1}{2}$. A uniformly distributed point can be generated by choosing z uniformly distributed on –1 to +1, and x and y uniformly distributed on a circle of radius $\sqrt{1 - z^2}$, a fact exploited in Arvo's algorithm (Arvo, 1991).

To derive the x distribution and average magnitude, we integrate circular slices. Since the sphere is symmetrical, and only positive x values lie in the right hemisphere, we will confine our attention there.
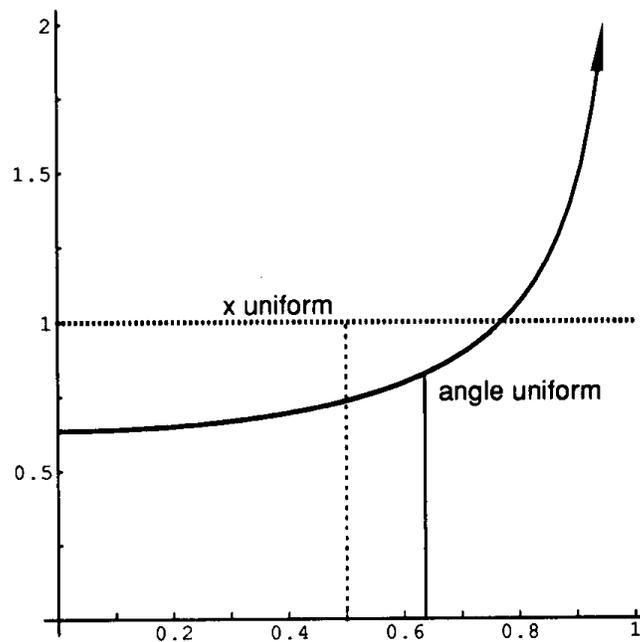

Figure 2. Density and average of x for planar rotation.

Calculation will be simpler if the variable of integration is θ, with $x = \sin\theta$; then the radius of the circular slice at x will be $\cos\theta$, and its perimeter length $2\pi\cos\theta$. The area of the hemisphere is, of course, just the integral of the perimeters for θ from 0 to $\pi/2$, which is 2π. For the circle, the integral for average magnitude weighted each x value by 2, since the one-dimensional slices gave exactly two points. Now the weight for $x = \sin\theta$ will be $2\pi\cos\theta$, because of the circular slice. (There, x was $\cos\theta$; here, $\sin\theta$.) So the average magnitude is

$$\frac{1}{2\pi}\int_0^{\pi/2} -2\pi\cos\theta\sin\theta\,d\theta = \frac{1}{2}.$$

To find the probability of a value being between 0 and x, we simply integrate the circular slices from 0 to $\arcsin x$, giving

$$\frac{1}{2\pi}\int_0^{\arcsin x} -2\pi\cos\theta\,d\theta = x.$$

This shows that the coordinate distributions are uniform (though not independently so!) for points uniformly distributed on a sphere.
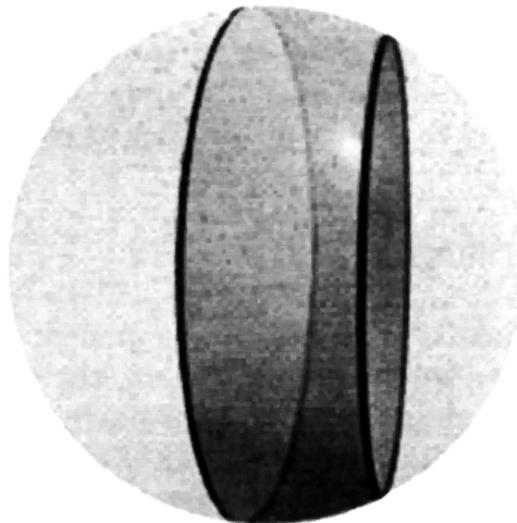

Figure 3. Sphere distribution integral.

Spatial Rotations

With these warm-up exercises behind us, we are ready to tackle spatial rotations. Once again there are several possible representations, including Euler angles $(\theta_x, \theta_y, \theta_z)$, unit quaternions $w + \mathbf{i}v + \mathbf{j}x + \mathbf{k}z$ (Shoemake, 1985, 1989), and 3×3 matrices

$$\begin{bmatrix} 1 - 2(y^2 + z^2) & 2xy - 2wz & 2xz + 2wy \\ 2xy + 2wz & 1 - 2(x^2 + z^2) & 2yz - 2wx \\ 2xz - 2wy & 2yz + 2wx & 1 - 2(x^2 + y^2) \end{bmatrix}.$$

In this case, unit quaternions are the best representation, and the geometric problem turns out to be one of generating a point uniformly distributed on a sphere in four dimensions, the quaternion unit sphere $x^2 + y^2 + z^2 + w^2 = 1$. Composition of a random rotation X with a given rotation R is given by multiplication of the corresponding unit quaternions, $q_R \blacklozenge q_X$ Multiplication turns (and/or reflects) the hypersphere, just as two-dimensional rotation composition turns the unit circle. Nonuniformity in the distribution of q_X values on the hypersphere shows up as a change—violating the Haar criterion—when it is turned by composition with some q_R. Turns around any four-dimensional axis are possible (Shoemake, 1991), so there are no "dead spots" where nonuniformity can hide (except equivalence of q and $-q$, but we avoid that loophole by dealing with magnitudes). So only uniformly distributed unit quaternions correspond to uniformly distributed rotations.

Angles Not Uniform

The average magnitude of, say, the w component can be computed in perfect analogy to the spherical case. There, the average x magnitude was obtained by integrating over a hemisphere (to give only positive values) and dividing by the associated area. Here, we integrate over half the hypersphere, and divide by the corresponding hypersurface measure. There, a circle of radius $\cos\theta$ contributed to each x value of $\sin\theta$; here, a complete three-dimensional sphere of radius $\cos\theta$ contributes to the w

value of sin θ. The area of a radius r sphere is $4\pi r^2$, while that of a unit hypersphere is $2\pi^2$. Thus, the average magnitude of w for uniformly distributed unit quaternions is given by

$$\frac{1}{\pi^2} \int_0^{\pi/2} 4\pi \cos^2\theta \sin\theta \, d\theta \; = \; \frac{4}{3\pi} \; \approx \; 0.4244.$$

We are now in a position to prove that a uniformly distributed spatial rotation does not have a uniformly distributed angle. For a unit quaternion, the w component is the cosine of half the angle of rotation. When the angle is uniformly distributed between 0 and 2π, the average magnitude of w will be $2/\pi \approx 0.6366$, which exceeds the correct value for a uniform rotation by a factor of $\frac{3}{2}$. Thus, the algorithm in Arvo (1991) cannot generate a uniformly distributed rotation, as claimed.

Uniform Rotations from Gaussians

Fortunately, it is easy to generate random unit quaternions—and hence rotations—with the correct distribution. Assign to the components of a quaternion the values of four Gaussian distributed independent random variables with mean 0 and any common standard deviation—say, 1. Then the quaternion itself will be Gaussian distributed in four-dimensional space (because of the separability of Gaussians) and can be normalized to give a uniformly distributed unit quaternion (because of the spatial symmetry of Gaussians). Pairs of independent variables with Gaussian distribution can easily be generated using the polar, or Box–Muller, method, which transforms a point uniformly distributed within the unit disk. The Gaussian generation can be folded into the unit quaternion generation to give an efficient algorithm (Knuth, 1981, p. 130).

Subgroup Algorithm

There is, however, a better way to generate uniform random rotations, an approach that generalizes efficiently to any number of dimensions, and to groups other than rotations. In our case, it reduces to the following simple prescription. Let X_0, X_1, and X_2 be three independent random

variables that are uniformly distributed between 0 and 1. Compute two uniformly distributed angles, $\theta_1 = 2\pi X_1$ and $\theta_2 = 2\pi X_2$, and their sines and cosines, s_1, c_1, s_2, c_2. Also compute $r_1 = \sqrt{1 - X_0}$ and $r_2 = \sqrt{X_0}$. Then return the unit quaternion with components $[s_1 r_1, \; c_1 r_1, \; s_2 r_2, \; c_2 r_2]$.

Before comparing the computed distribution with the correct distribution, let us look at how this code can be derived from first principles. As we saw earlier, a uniform plane rotation is easily obtained from a uniform angle between 0 and 2π. Plane rotations form a *subgroup* of the group of rotations in space, namely rotations around the z axis. The *cosets* of this subgroup are represented by the rotations pointing the z axis in different directions. By multiplying a uniformly distributed element from the subgroup with a uniformly distributed coset representative, the subgroup algorithm generates a uniformly distributed element of the complete group, as explained below.

To better understand the subgroup algorithm, consider a simpler example. The six permutations of a triangle's vertices form a group. Reversing the triangle generates a two-permutation subgroup (1, 2, 3) and (3, 2, 1), for which there are three cosets, {(1,2,3), (3,2,1)}, {(2,3,1), (1,3,2)}, and {(3, 1, 2), (2, 1, 3)}, each closed under composition with the subgroup permutations. Uniformly choose one of the cosets; then the combination of a permutation from that coset with a uniformly chosen permutation from the subgroup will be uniform over the whole group. Further examples are given in Diaconis and Shahshahani (1986).

In terms of quaternions, a rotation around z has the form $[0, 0, s, c]$, while a rotation pointing z in an arbitrary direction has the form
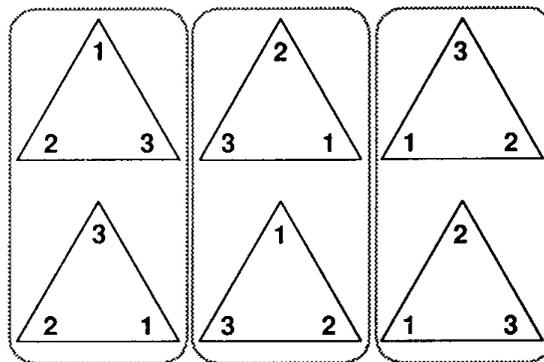

Figure 4. Cosets in dihedral group.

[x, y, 0, w]. If the direction is to be uniformly distributed, w must be distributed as the square root of a uniform distribution, and x and y must be a uniform plane rotation times $\sqrt{1 - w^2}$. The square root is necessary because the rotated z value should be uniform for a point uniformly distributed on a sphere. (Remember?) The z value comes out to be $2w^2 - 1$; substituting $w = \sqrt{X_0}$ gives $2X_0 - 1$, a uniform deviate between −1 and +1, as required. The product of the z placement with the z rotation is [cx + sy, −sx + cy, sw, cw], which is just one step away from the final code. We know, since this should give points uniformly distributed on a hypersphere, that all components have the same kind of distribution. In particular, the first two components are the product of two uniform plane rotations times a magnitude of $\sqrt{1 - X_0}$, which can be reduced to a single plane rotation of the correct magnitude, like the last two components. The result is the code stated.

Distribution Check

Ignoring the derivation, what is the distribution of a component, say, $\sqrt{X_0} \sin(2\pi X_2)$? The average magnitude can be computed by integrating square root from 0 to 1, and $1/\pi$ times the sine from 0 to π; taking their product gives the expected value $4/3\pi$. That this is correct is necessary but not sufficient to show that the distribution is correct, so we press on. The probability that the magnitude of a component is less than or equal to x should be $2/\pi \, (x\sqrt{1 - x^2} + \arcsin x)$, a value obtained from

$$\frac{1}{\pi^2} \int_0^{\arcsin x} -4\pi \cos^2\theta \, d\theta,$$

much as in the spherical case. Obtaining the computed probability is harder. For a uniform distribution, the probability of obtaining a value less than or equal to x is $F(x) = x$; for an invertible function $g(x)$ of a uniform distribution, the probability is $F(x) = g^{-1}(x)$. Thus, for $\sqrt{X_0}$ we have $F(x) = x^2$, while for $\sin\pi/2 \, X_2$ (with the range restricted for invertibility and magnitude) we have $F(x) = 2/\pi \arcsin x$. The density at x of this latter distribution is the derivative there, $2/\pi \, 1/\sqrt{1 - x^2}$. Now

the distribution of the product can be obtained. When the sine term has the value s, the only way to get a product less than or equal to x is for the square root term to be less than or equal to x/s, which—since the square root is at most 1—has probability $(\text{Min}(x/s,1))^2$. Weighting each s value by its density, and integrating over all s, we have

$$\int_0^1 \frac{2}{\pi} \frac{1}{\sqrt{1-s^2}} (\text{Min}(x/s,1))^2 \, ds$$

$$= \int_x^1 \frac{2}{\pi} \frac{1}{\sqrt{1-s^2}} (x/s)^2 \, ds + \int_0^x \frac{2}{\pi} \frac{1}{\sqrt{1-s^2}} \, ds$$

$$= \frac{2}{\pi} (x\sqrt{1-x^2} + \arcsin x),$$

as required.

Acknowledgments

Thanks to James Arvo for his questions and suggestions; this Gem is better as a result.

See also G2, 355; G3, C.4.

INTERPOLATION USING BÉZIER CURVES

Gershon Elber
Computer Science Department
University of Utah

Introduction

The Bézier representation (Eq. (1)) is well known and frequently used for CAD applications as it possesses extremely useful properties:

$$B(\text{t}) \;=\; \sum_{i=0}^{k} P_i B_i^k(t), \qquad B_i^k(t) \;=\; \binom{k}{i} t^i (1 \,-\, t)^{k-i}. \tag{1}$$

Bézier curves are easy to evaluate, derive, and subdivide and are unimodal for $t \in [0, \ldots, 1]$. This representation possesses important properties such as control polygon convex hull curve bounding, intuitive curve shape control using control points, and the variation diminishing property. All in all, the Bézier representation is a very useful tool for CAD applications.

A Bézier curve only approximates the shape of its control polygon. If an interpolation scheme is required, this representation cannot usually be used. It may be desired to find the Bézier curve that interpolates a set of points. This will enable the use of the simple and elegant evaluation algorithms (Goldman, 1990) of the Bézier representation with its useful properties.

In this Gem we will present a simple way to find the Bézier curve that *interpolates* a given set of points.

Numeric Solution

When one attempts to solve the interpolation problem for Bézier curves, a set of linear equations may be defined and solved. Let $\mathbf{B}(t)$ be the Bézier curve interpolating the point set $\diagup = (\mathbf{T}, \mathbf{V}) = (t_i, V_i)$, $i = 0, \ldots, k$, t, $\neq$ t_j, $\forall i \neq j$:

$$\mathbf{B}(t_i) = V_i, \qquad i = 0, \ldots, k \tag{2}$$

A one-dimensional Bézier curve of degree k has $k + 1$ degrees of freedom—its coefficients or control points. Therefore, given a set of $k + 1$ points to interpolate, a Bézier curve of at least degree k *is* required to ensure that solution exists.

A linear system of $k + 1$ equations is defined for the $k + 1$ Bézier control polygon points, $\mathbf{P} = P_i$, $i = 0, \ldots, k$, as the unknowns:

$$\begin{bmatrix} B_0^k(t_0) & B_1^k(t_0) & \cdots & B_k^k(t_0) \\ B_0^k(t_1) & B_1^k(t_1) & \cdots & B_k^k(t_1) \\ \vdots & \vdots & \ddots & \vdots \\ B_0^k(t_k) & B_1^k(t_k) & \cdots & B_k^k(t_k) \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ \vdots \\ P_k \end{bmatrix} = \begin{bmatrix} V_0 \\ V_1 \\ \vdots \\ V_k \end{bmatrix} \tag{3}$$

and given input $\diagup$ one can numerically solve and find $\mathbf{P}$. Note that V_i (and P_i) may be vectors in which the linear system should be solved coordinatewise. Let $\mathbf{M}$ be the B's square matrix of Eq. (3). As almost all $B_i^k(t_j) \neq 0$ in $\mathbf{M}$, the solution to Eq. (3) is of the order $O(k^3)$ or quite expensive. In the next section we present a way to perform this task without the need to solve such a linear system each time.

Symbolic Solution

While the numeric technique described in the preceding section is general, one can alleviate the need to solve the linear system each time by posing one more constraint on the problem. Let $\diagup = ((i/k), V_i)$, $i = 0, \ldots, k$, be the set of points to interpolate. In other words, the parameter value interpolating V_i is not free any more, but equal to i/k. One only

needs to specify $\mathbf{V}$ or $/ = (\mathbf{V}) = (V_i)$, $i = 0, \ldots, k$, as now the $t_i s$ are in fixed and equally spaced positions.

$B_i^k(t)$ maximum is at i/k. Therefore, P_i is most influenced from V_i, which is usually more intuitive. However, any fixed and distinguished set of $k + 1$ parameter values may be used in a similar way.

Updating Eq. (3) and using this new constraint, we get

$$
\begin{bmatrix}
B_0^k\left(\dfrac{0}{k}\right) & B_1^k\left(\dfrac{0}{k}\right) & \cdots & B_k^k\left(\dfrac{0}{k}\right) \\[2mm]
B_0^k\left(\dfrac{1}{k}\right) & B_1^k\left(\dfrac{1}{k}\right) & \cdots & B_k^k\left(\dfrac{1}{k}\right) \\[2mm]
\vdots & \vdots & \ddots & \vdots \\[2mm]
B_0^k\left(\dfrac{k}{k}\right) & B_1^k\left(\dfrac{k}{k}\right) & \cdots & B_k^k\left(\dfrac{k}{k}\right)
\end{bmatrix}
\begin{bmatrix}
P_0 \\[2mm] P_1 \\[2mm] \vdots \\[2mm] P_k
\end{bmatrix}
=
\begin{bmatrix}
V_0 \\[2mm] V_1 \\[2mm] \vdots \\[2mm] V_k
\end{bmatrix}. \tag{4}
$$

Interestingly enough, $\mathbf{M}$ in Eq. (4) is *independent* of $/$. In other words, one can solve this system (invert $\mathbf{M}$) without knowing anything about the input set $/$! Given a set of points, $/$, the control polygon of the Bézier curve interpolating $/$ is

$$
\mathbf{P} = \mathbf{M}^{-1} \mathbf{V}, \tag{5}
$$

where $\mathbf{M}^{-1}$ is $\mathbf{M}$ inverse.

By picking i/k as the parameters at which the Bézier curve is interpolating the input data, all the terms in $\mathbf{M}$, M_{ji}, are of the form

$$
M_{ji} = B_i^k\left(\frac{j}{k}\right) = \binom{k}{i}\left(1 - \frac{j}{k}\right)^{k-i}\left(\frac{j}{k}\right)^i = \frac{k!(k-j)^{k-i}j^i}{i!(k-i)!k^k} \tag{6}
$$

or the term in Eq. (6) and therefore all the terms in $\mathbf{M}$ and $\mathbf{M}^{-1}$ may be expressed as rational integers exactly.

Implementation

The C program provided uses the symbolic solution for $\mathbf{M}^{-1}$ for Bézier curves from order 2 (linear) to 9. A long rational integer (32 bits) is used to hold the exact symbolic solution (Reduce, 1987). The zero element of each row holds the line common denominator for the rest of the line numerators. Most of the implementation is, in fact, the symbolic solution represented as rational values. The following routines simply multiply this matrix solution by the given $\mathbf{V}$ input set to find the control polygon point set, $\mathbf{P}$.

See also G1, 75; G3, C.4.

III.8

RIGID PHYSICALLY BASED SUPERQUADRICS

A. H. Barr
California Institute of Technology
Pasadena, California

Introduction

Superquadric ellipsoids and toroids are recent geometric shapes, useful for computer graphics modeling. In this article, we provide equations needed to calculate the motion of these shapes in rigid physically based modeling: We present closed-form algebraic expressions for the volume, center of mass, and rotational inertia tensor for (constant density) superquadric shapes. We do not cover nonrigid physically based motion. In the appendices, we briefly review superquadrics, the equations of rigid body motion of Newtonian physics, and ancillary mathematical definitions and derivations.

Review of Superquadrics

Superquadrics (Barr, 1981) are three-dimensional extensions of Piet Hein's two-dimensional superellipses (Faux and Pratt, 1979). They allow us to easily represent rounded, square, cylindrical, pinched, and toroidal shapes with relatively simple equations. The superquadric parametric surface function is a profile surface based on trigonometric functions raised to exponents (which retain the appropriate plus or minus sign in their octant).

There are six shape parameters of the superquadrics:

- the roundness/squareness shape parameter in the north-south direction is *"n"*
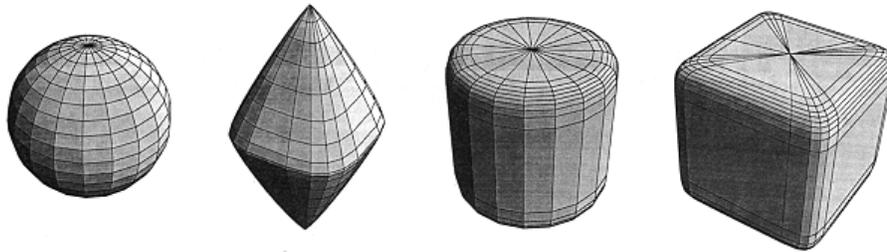
Figure 1. Examples of superquadric ellipsoids. From left to right, we produce a sphere, a pyramid, a cylindroid, and a cuboid. The north-south parameters, respectively, are 1.0, 1.8, 0.2, 0.2, and the east-west parameters are 1.0, 0.6, 1.0, and 0.2.

- the east–west roundness/squareness parameter is "e"

- a_1, a_2, and a_3 are length, width, and depth parameters

- for toroids, "α" is a "hole diameter" parameter and should be greater than one, to avoid self-intersection.

Equations reviewing the geometric properties of superquadrics are found in Appendix A.

Rigid Physically Based Superquadric Quantities

We need several quantities to calculate physically based computer graphics motions of rigid bodies. Specifically, we need to know

1. the position x_C of the center of mass of the object,

2. the net mass M of the bodies, and

3. the rotational inertia tensor, $\underline{\underline{I}}$, of the body.

We use these quantities in the equations of rigid body motion, which we review briefly in Appendix B. We use notation similar to that of Barzel and Barr (1988) and refer the reader to that article and to Barzel (1992) for a description of rigid body motion and methods to calculate dynamic constraints on rigid bodies. We refer to Barr (1984), Terzopoulis et *al.* (1987), and Pentland and Williams (1989) for different types of nonrigid motion.
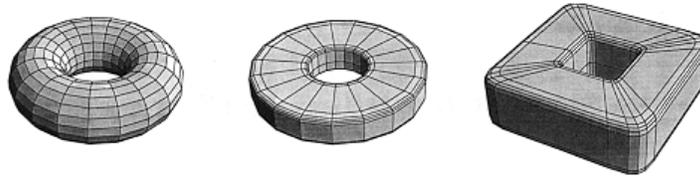
Figure 2. Examples of superquadric toroids. From left to right, we produce a round torus, a "pineapple-slice" toroid, and a square toroid. The north-south parameters, respectively, are 1.0, 0.2, 0.2, and the east-west parameters are 1.0, 1.0, and 0.2. The hole parameter, α, is 2.

Center of Mass

For the canonical superquadrics described in Appendix A, the position of the center of mass is at the origin:

$$\underline{x}_C = \underline{0}.$$

Of course, when we calculate a new center of mass of the object from the equations of rigid body motion, we need to translate the superquadric to the new position.

Volume, Density, and Mass

There are a number of ways to specify volume, density, and mass. In this article, we let the terms ρ and M be the density and mass of the superquadric object (ellipsoid or toroid). The user first chooses the substance of the object (say steel or wood), which determines ρ, its density. Then the mass of the object is determined from the object's volume.[1]

We let V_E signify the volume of the superquadric ellipsoids, and V_T the volume of the toroids (V without either subscript can signify the volume of either shape). We express the volume formula in terms of beta functions, $\beta(m, n)$. Methods for computing $\beta(m, n)$ are shown in Appendix C. Appendix D provides a sketch of the derivation of the volume and inertia tensor formulas.

---

[1]This is the recommended approach. Of course, we could also choose the mass first, without choosing a "real" material. Then the density would be the derived quantity, instead of the mass.

Volume, Superquadric Ellipsoids

$$V_E = \frac{2}{3} a_1 a_2 a_3 e \, n\beta\left(\frac{e}{2}, \frac{e}{2}\right)\beta\left(n, \frac{n}{2}\right).$$

Volume, Superquadric Toroids

$$V_T = 2 a_1 a_2 a_3 \, \alpha e \, n\beta\left(\frac{e}{2}, \frac{e}{2}\right)\beta\left(\frac{n}{2}, \frac{n}{2}\right).$$

Mass, Ellipsoid or Toroid

The mass M is expressed in terms of the volume V and density ρ. ρ = substance density, $M = \rho V$, where V is the volume of either ellipsoid or toroid.

Inertia Tensor

The formulas for the inertia tensor are the primary results of this paper. We let $\underline{\underline{I}}_E$ be the inertia tensor of the superquadric ellipsoids, and $\underline{\underline{I}}_T$ the inertia tensor of the toroids.

Inertia Tensor, Superquadric Ellipsoid

In body coordinates, the components of the inertia tensor are constant. The off-diagonal components are zero, due to symmetry arguments:

$$\underline{\underline{I}}_E^{\text{body}} = \rho_E \begin{pmatrix} i_{2E} + i_{3E} & 0 & 0 \\ 0 & i_{1E} + i_{3E} & 0 \\ 0 & 0 & i_{1E} + i_{2E} \end{pmatrix}, \qquad \text{where}$$

ρ_E = (constant) density of superquadric ellipsoid,

$$i_{1E} = \frac{2}{5} a_1^3 a_2 a_3 e \, n\beta\left(3\frac{e}{2}, \frac{e}{2}\right)\beta\left(2n, \frac{n}{2}\right),$$

$$i_{2E} = \frac{2}{5} a_1 a_2^3 a_3 e \, n\beta\left(\frac{e}{2}, 3\frac{e}{2}\right)\beta\left(2n, \frac{n}{2}\right),$$

$$i_{3E} = \frac{2}{5} a_1 a_2 a_3^3 e \, n\beta\left(\frac{e}{2}, \frac{e}{2}\right)\beta\left(n, \frac{3n}{2}\right).$$

In Appendix D, we show the derivation of i_{1E}, i_{2E}, and i_{3E}, along with the volume, V.

Inertia Tensor, Superquadric Toroid

Likewise, the components of the toroid inertia tensor are constant in body coordinates.

$$\underline{\underline{I}}_{T}^{\text{body}} = \rho_{T}\begin{pmatrix} i_{2T} + i_{3T} & 0 & 0 \\ 0 & i_{1T} + i_{3T} & 0 \\ 0 & 0 & i_{1T} + i_{2T} \end{pmatrix}, \qquad \text{where}$$

ρ_{T} = (constant) density of superquadric toroid,

$$i_{1T} = a_1^3 a_2 a_3 \alpha e\, n\beta\left(3\frac{e}{2}\,,\,\frac{e}{2}\right)\left(2\alpha^2\beta\left(\frac{n}{2}\,,\,\frac{n}{2}\right)+3\beta\left(\frac{3n}{2}\,,\,\frac{n}{2}\right)\right),$$

$$i_{2T} = a_1 a_2^3 a_3 \alpha e\, n\beta\left(\frac{e}{2}\,,\,3\frac{e}{2}\right)\left(2\alpha^2\beta\left(\frac{n}{2}\,,\,\frac{n}{2}\right)+3\beta\left(\frac{3n}{2}\,,\,\frac{n}{2}\right)\right),$$

$$i_{3T} = a_1 a_2 a_3^3 \alpha e\, n\beta\left(\frac{e}{2}\,,\,\frac{e}{2}\right)\beta\left(\frac{n}{2}\,,\,\frac{3n}{2}\right).$$

In Appendix D, we show the derivation of i_{1T}, i_{2T}, and i_{3T}, along with the volume, V.

Examples of the Volume and Inertia Tensors

For some values of n and e, the superquadric inertia tensor is particularly simple and can be compared to known inertia tensors of spheres, ellipsoids, blocks, and cones. Note that the values of a_1, a_2, and a_3 are the principal radii of the shapes (not the diameters!).

Superquadric Ellipsoid Examples

We present the volume and nonzero components of the inertia tensors for particular superquadric ellipsoids. The reader can compare the results of their numerical computations to these formulas, for verification purposes.

Name	n	e	Volume V_E	I_E^{body}: $\quad I_{11}/\rho$ $\quad I_{22}/\rho$ $\quad I_{33}/\rho$
block	0	0	$8a_1a_2a_3$	$\dfrac{8a_1a_2a_3(a_2^2+a_3^2)}{3}$ $\dfrac{8a_1a_2a_3(a_1^2+a_3^2)}{3}$ $\dfrac{8a_1a_2(a_1^2+a_2^2)a_3}{3}$
elliptical cylinder	0	1	$2\pi a_1a_2a_3$	$\dfrac{\pi a_1a_2^3a_3}{2}+\dfrac{2\pi a_1a_2a_3^3}{3}$ $\dfrac{\pi a_1^3a_2a_3}{2}+\dfrac{2\pi a_1a_2a_3^3}{3}$ $\dfrac{\pi a_1^3a_2a_3}{2}+\dfrac{\pi a_1a_2^3a_3}{2}$
pillow shape	1	0	$\dfrac{16a_1a_2a_3}{3}$	$\dfrac{64a_1a_2^3a_3}{45}+\dfrac{16a_1a_2a_3^3}{15}$ $\dfrac{64a_1^3a_2a_3}{45}+\dfrac{16a_1a_2a_3^3}{15}$ $\dfrac{64a_1a_2(a_1^2+a_2^2)a_3}{45}$
ellipsoid	1	1	$\dfrac{4\pi a_1a_2a_3}{3}$	$\dfrac{4\pi a_1a_2a_3(a_2^2+a_3^2)}{15}$ $\dfrac{4\pi a_1a_2a_3(a_1^2+a_3^2)}{15}$ $\dfrac{4\pi a_1a_2(a_1^2+a_2^2)a_3}{15}$
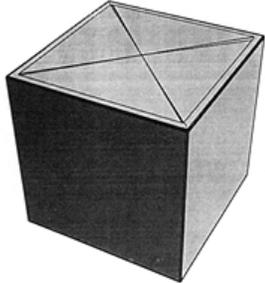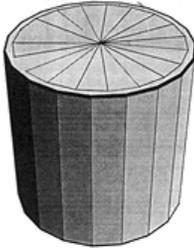
Name	n	e	Volume V_E	I_E^{body}: $\quad\begin{array}{c}I_{11}/\rho\\I_{22}/\rho\\I_{33}/\rho\end{array}$
double pyramid	2	2	$\dfrac{4a_1a_2a_3}{3}$	$\dfrac{2a_1a_2a_3(a_2^2+a_3^2)}{15}$ $\dfrac{2a_1a_2a_3(a_1^2+a_3^2)}{15}$ $\dfrac{2a_1a_2(a_1^2+a_2^2)a_3}{15}$
double cone	2	1	$\dfrac{2\pi a_1a_2a_3}{3}$	$\dfrac{\pi a_1a_2a_3(3a_2^2+2a_3^2)}{30}$ $\dfrac{\pi a_1a_2a_3(3a_1^2+2a_3^2)}{30}$ $\dfrac{\pi a_1a_2(a_1^2+a_2^2)a_3}{10}$
pinched double cone	3	1	$\dfrac{32\pi a_1a_2a_3}{105}$	$\dfrac{512\pi a_1a_2^3a_3}{15015}+\dfrac{32\pi a_1a_2a_3^3}{2145}$ $\dfrac{512\pi a_1^3a_2a_3}{15015}+\dfrac{32\pi a_1a_2a_3^3}{2145}$ $\dfrac{512\pi a_1^3a_2a_3}{15015}+\dfrac{512\pi a_1a_2^3a_3}{15015}$
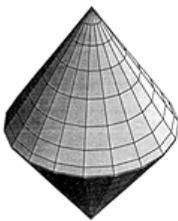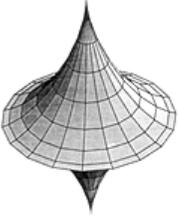
Toroid Examples

We provide similar test cases for the superquadric toroids.

Name	n	e	Volume V_T	I_T^{body}: $\begin{matrix} I_{11}/\rho \\ I_{22}/\rho \\ I_{33}/\rho \end{matrix}$
rectangular toroid	0	0	$32a_1a_2a_3\alpha$	$\dfrac{32a_1a_2a_3\alpha\left(2a_2^2 + a_3^2 + 2a_2^2\alpha^2\right)}{3}$ $\dfrac{32a_1a_2a_3\alpha\left(2a_1^2 + a_3^2 + 2a_1^2\alpha^2\right)}{3}$ $\dfrac{64a_1a_2\left(a_1^2 + a_2^2\right)a_3\alpha(1 + \alpha^2)}{3}$
pineapple slice	0	1	$8\pi a_1a_2a_3\alpha$	$\dfrac{4\pi a_1a_2a_3\alpha\left(3a_2^2 + 2a_3^2 + 3a_2^2\alpha^2\right)}{3}$ $\dfrac{4\pi a_1a_2a_3\alpha\left(3a_1^2 + 2a_3^2 + 3a_1^2\alpha^2\right)}{3}$ $\left(4\pi a_1a_2\left(a_1^2 + a_2^2\right)a_3\alpha(1 + \alpha^2)\right)$
pillow toroid	1	0	$8\pi a_1a_2a_3\alpha$	$\dfrac{2\pi a_1a_2a_3\alpha\left(6a_2^2 + 3a_3^2 + 8a_2^2\alpha^2\right)}{3}$ $\dfrac{2\pi a_1a_2a_3\alpha\left(6a_1^2 + 3a_3^2 + 8a_1^2\alpha^2\right)}{3}$ $\dfrac{4\pi a_1a_2\left(a_1^2 + a_2^2\right)a_3\alpha(3 + 4\alpha^2)}{3}$
torus	1	1	$2\pi^2 a_1a_2a_3\alpha$	$\dfrac{\pi^2 a_1a_2a_3\alpha\left(3a_2^2 + 2a_3^2 + 4a_2^2\alpha^2\right)}{4}$ $\dfrac{\pi^2 a_1a_2a_3\alpha\left(3a_1^2 + 2a_3^2 + 4a_1^2\alpha^2\right)}{4}$ $\dfrac{\pi^2 a_1a_2\left(a_1^2 + a_2^2\right)a_3\alpha(3 + 4\alpha^2)}{4}$
double cone toroid	2	1	$4\pi a_1a_2a_3\alpha$	$\dfrac{\pi a_1a_2a_3\alpha\left(3a_2^2 + 2a_3^2 + 6a_2^2\alpha^2\right)}{3}$ $\dfrac{\pi a_1a_2a_3\alpha\left(3a_1^2 + 2a_3^2 + 6a_1^2\alpha^2\right)}{3}$ $\left(\pi a_1a_2\left(a_1^2 + a_2^2\right)a_3\alpha(1 + 2\alpha^2)\right)$
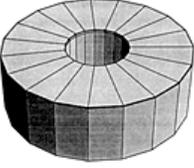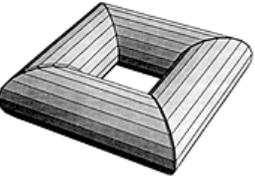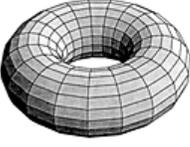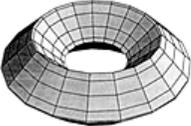
Obtaining the Tensors and Their Inverses in World Coordinates

To obtain the components of the inertia tensor in world coordinates, we use the same 3×3 rotation matrix $\underline{\underline{R}}$ that rotates the body vectors into world coordinates.

The inertia tensor in world coordinates is given by

$$\left(I^{\text{world}}\right)_{ij} = \sum_{k=1}^{3} \sum_{\ell=1}^{3} R_{ik} R_{j\ell} \left(I^{\text{body}}\right)_{k\ell} .$$

The components of the inverse matrix of the inertia tensor in world coordinates are given by

$$\left(\left(I^{\text{world}}\right)^{-1}\right)_{ij} = \sum_{k=1}^{3} \sum_{\ell=1}^{3} R_{ik} R_{j\ell} \left(\left(I^{\text{body}}\right)^{-1}\right)_{k\ell} .$$

Note that in the body coordinate system, the components of the matrices are constant and only need to be computed once (and that in world coordinates the components change as a function of time).

Conclusion

By applying the results of the preceding sections in the context of Appendix B, the reader is able to add superquadric shapes to a previously written physically based computer graphics modeling package.

Acknowledgment

Appendix A: Review of Superquadric Geometric Quantities

Superquadrics have an unusual property: They have closed-form algebraic expressions for their most important geometric features. This closed-form property makes them easier to use and more appropriate for computer graphics applications. Thus, like spheres, they have

1. a relatively simple parametric form of their surface,

2. an implicit function to test if a given 3-D point is inside or outside of the shape, and

3. simple expressions for their normal vectors.

Parametric Surface Functions

We need three functions, $c(w, m)$, $s(w, m)$, and $c_T(w, m, \alpha)$ to calculate the parametric surface for superquadric ellipsoids and toroids:

$$c(w, m) = \text{sgn}(\cos(w))|\cos(w)|^m,$$

$$c_T(w, m, \alpha) = \alpha + c(w,m), \qquad \alpha > 1,$$

$$s(w, m) = \text{sgn}(\sin(w))|\sin(w)|^m.$$

For us,

$$
\text{sgn}(x) = \begin{cases} -1, & x < 0 \\ 0, & x = 0 \\ 1, & x > 0 \end{cases}.
$$

Also, note that $c_T(w, m, \alpha)$ is always greater than zero (to avoid self-intersection).

Superquadric Ellipsoid

(Surface parameters u and v; dimensions: a_1, a_2, a_3; roundness/squareness shape parameters: n, e.)

$$x(u, v) = a_1 c(v, n) c(u, e),$$

$$y(u, v) = a_2 c(v, n) s(u, e),$$

$$z(u, v) = a_3 s(v, n),$$

$$-\pi/2 \leq v \leq \pi/2, \quad -\pi \leq u < \pi.$$

See Figs. 1 and 3.

Superquadric Toroid

(Surface parameters u and v; dimension parameters: a_1, a_2, a_3; hole diameter parameter: α, $\alpha > 1$; roundness/squareness shape parameters: n, e.)

$$x(u, v) = a_1 c_T(v, n, \alpha) c(u, e),$$

$$y(u, v) = a_2 c_T(v, n, \alpha) s(u, e),$$

$$z(u, v) = a_3 s(v, n),$$

$$-\pi \leq v < \pi, \quad -\pi \leq u < \pi.$$

Note that unlike the ellipsoids, the v parameter for the toroids goes completely around a circle, from $-\pi$ to π, instead of only halfway around.

See Figs. 2 and 4.

"Inside-Outside" Function

If $f(x, y, z) < 0$ we are inside the object; if $f(x, y, z) = 0$ we are on the object, while if $f(x, y, z) > 0$ we are outside the object.

"Inside-Outside" Function of Superquadric Ellipsoids

$$f(x,\ y,\ z)\ =\ \left(\left|x/a_1\right|^{2/e} + \left|y/a_2\right|^{2/e}\right)^{e/n} +\ \left|z/a_3\right|^{2/n} - 1$$

"Inside-Outside" Function of Superquadric Toroids

$$f(x,\ y,\ z)\ =\ \left|\left(\left|x/a_1\right|^{2/e} + \left|y/a_2\right|^{2/e}\right)^{e/2} - \alpha\right|^{2/n} +\ \left|z/a_3\right|^{2/n} - 1.$$

Normal Vectors

As the reader is aware of, normal vectors are used in computer graphics shading operations. To obtain unit normal vectors, you need to divide by the magnitude of the normal vectors in the following equations:

Normal Vectors, Superquadric Ellipsoids and Toroids, Parametric Form

$$N_1(u,\ v)\ =\ \frac{1}{a_1}\,c(v,\ 2\ -\ n)\,c(u,\ 2\ -\ e),$$

$$N_2(u,\ v)\ =\ \frac{1}{a_2}\,c(v,\ 2\ -\ n)\,s(u,\ 2\ -\ e),$$

$$N_3(u,\ v)\ =\ \frac{1}{a_3}\,s(v,\ 2\ -\ n).$$

Normal Vectors, Superquadric Ellipsoids and Toroids, Implicit Form

$$N_1\ =\ \frac{\partial f(x,y,z)}{\partial x},$$

$$N_2\ =\ \frac{\partial f(x,y,z)}{\partial y},$$

$$N_3\ =\ \frac{\partial f(x,y,z)}{\partial z}.$$

Appendix B: Some Equations of Rigid-Body Motion

Although we cannot provide a complete description of rigid-body motion within the scope of this article, we can review parts of it briefly.

The main purpose of the rotational inertia tensor $\underline{\underline{I}}$ is to allow us to convert between angular momentum $\underline{L}$ and angular velocity $\underline{\omega}$. The rotational inertia matrix times the angular velocity is the angular momentum. It is very similar to the conversion between linear momentum $\underline{P}$ and linear velocity $\underline{v}$. The mass (linear inertia) M times the velocity is the linear momentum $\underline{P}$.

There are several types of equations that rigid-body motion utilizes:

Differential equations:

$$\underline{x}' = \underline{v},$$

$$\underline{q}'_i = \frac{1}{2}\begin{pmatrix} -\omega \cdot r \\ s\omega + \omega \times r \end{pmatrix}$$

$$\underline{P}' = F,$$

$$\underline{L}' = T;$$

Initial conditions:

$$\underline{x}(0) = \underline{x}_0,$$

$$\underline{q}(0) = \underline{q}_0,$$

$$\underline{P}(0) = M\underline{v}_0,$$

$$\underline{L}(0) = \underline{\underline{I}}^{\text{world}}\underline{\omega}_0;$$

Auxiliary equations:

$$\underline{v} = \frac{P}{M},$$

$$\underline{\omega} = \left(\underline{\underline{I}}^{\text{world}}\right)^{-1}\underline{L},$$

$$\underline{\underline{R}} = \begin{bmatrix} 1-2\hat{q}_2^2-2\hat{q}_3^2 & 2\hat{q}_1\hat{q}_2-2\hat{q}_0\hat{q}_3 & 2\hat{q}_1\hat{q}_3+2\hat{q}_2\hat{q}_3 \\ 2\hat{q}_1\hat{q}_2+2\hat{q}_0\hat{q}_3 & 1-2\hat{q}_1^2-2\hat{q}_3^2 & 2\hat{q}_2\hat{q}_3-2\hat{q}_0\hat{q}_1 \\ 2\hat{q}_1\hat{q}_3-2\hat{q}_0\hat{q}_1 & 2\hat{q}_2\hat{q}_3+2\hat{q}_0\hat{q}_1 & 1-2\hat{q}_1^2-2\hat{q}_2^2 \end{bmatrix}$$

s = scalar part of quaternion

r = vector part of quaternion

where
  $\underline{x}$ is a three-by-one vector for the position of the center of mass of the object, for us to translate our object by;
  $\underline{\underline{R}}$ is a right-handed three-by-three rotation matrix that rotates the object;
  $\underline{P}$ is the linear momentum of the object (a three-by-one vector);
  $\underline{L}$ is the angular momentum of the object (three-by-one vector);
  $\underline{F}$ is the net force acting on the object's center of mass (a three-by-one vector);
  $\underline{T}$ is the net torque acting around the object's center of mass (a three-by-one vector);
  $\underline{v}$ is the linear velocity of the object (a three-by-one vector);
  $\underline{\omega}$ is the angular velocity of the object (a three-by-one vector);
  M is the mass of the object;
  $\underline{I}^{\text{world}}$ is a three-by-three matrix for the rotational inertia tensor of the object in world coordinates (see the section in the text on the inertia tensor);
  and $\hat{q} = q/\sqrt{q \cdot q}$
  Please see Goldstein (1980) for additional details.

Appendix C: How to Compute $\beta(m, n)$, and $\Gamma(n)$

Integral Form of the Beta Function

The β function is defined via the integral

$$\int_0^1 t^{n-1}(1 - t)^{m-1}\, dt = \beta(m, n).$$

The parameters m and n are nonnegative real numbers.

The volume and inertia tensor derivation is based on the following integral, where we transform the preceding definition. We let $t = \sin^2(x)$ and divide both sides by 2:

$$\int_0^{\pi/2} \cos^{2n-1}(x)\sin^{2m-1}(x)\, dx = \frac{1}{2}\beta(m, n).$$

Thus, we can evaluate any definite integral with an integrand consisting of sin and cos raised to (non-integer) powers (if we can coerce it into having limits from zero to $\pi/2$). This is the heart of the derivation shown in Appendix D.

How to Compute $\beta(n, m)$

When we wish to compute the value of the beta function, we take the ratio of numerically computed gamma functions:

$$\beta(m, n) = \frac{\Gamma(m)\Gamma(n)}{\Gamma(m + n)}.$$

How to Numerically Compute $\Gamma(x)$

The Γ function is a continuum form of the factorial function (with its argument shifted by 1). For all real $x > 0$ (integer or not),

$$\Gamma(x + 1) = x\Gamma(x).$$

When x is a positive integer,

$$\Gamma(x + 1) = x!.$$

Also,

$$\Gamma(1/2) = \pi^{1/2}.$$

Based on a continued fraction formulation found in Abremowitz and Stegun (1970) we provide a method to compute $\Gamma(x)$.

Let

$$\gamma_0 = 1/12,$$

$$\gamma_1 = 1/30,$$

$$\gamma_2 = 53/210,$$

$$\gamma_3 = 195/371,$$

$$\gamma_4 = 22{,}999/22{,}737,$$

$$\gamma_5 = 29{,}944{,}523/19{,}733{,}142,$$

$$\gamma_6 = 109{,}535{,}241{,}009/48{,}264{,}275{,}462,$$

$$G(x) = \frac{1}{2}\log(2\pi) - x + \left(x - \frac{1}{2}\right)\log(x)$$

$$+ \gamma_0 / (x + \gamma_1 / (x + \gamma_2 / (x + \gamma_3 / (x + \gamma_4 / (x + \gamma_5 / (x + \gamma_6 / x)))))),$$

$$\Gamma(x) = \exp[G(x + 5)]/(x(x + 1)(x + 2)(x + 3)(x + 4)).$$

Appendix D: Sketch of Derivation of Superquadric Volume, Mass, and Inertia Tensor

The volume V of an object is given by

$$V = \iiint_{\text{region}} 1 \; dxdydz.$$

The mass M of an object is given by

$$M = \iiint_{\text{region}} \rho(x, y, z) \; dxdydz,$$

where $\rho(x, y, z)$ is the density of the objects.

The rotational inertia tensor $\underline{\underline{I}}$ of an object is given by the following expression:

$$\underline{\underline{I}} = \iiint_{\text{region}} \rho(\textbf{\textit{x, y, z}}) \begin{pmatrix} y^2+z^2 & -(xy) & -(xz) \\ -(xy) & x^2+z^2 & -(yz) \\ -(xz) & -(yz) & x^2+y^2 \end{pmatrix} \textbf{\textit{dxdydz}}.$$

For constant density, by symmetry, we can determine that the off-diagonal terms are zero for the rotational inertia tensor of a superquadric in its home coordinate system. We can integrate x^2, y^2, and z^2, and then combine them additively to get the diagonal terms. We will also need to integrate "1" to get the volume.

If the density were not constant, but instead were a function $\rho(r, u, v)$ expressed in terms of sin and cos, and integer powers of r, the derivation would be similar to what follows, except we would need to compute seven quantities instead of four (six quantities from the three-by-three matrix, and the mass integral). The eight different pieces would need to be computed separately.

For constant density, however, it is sufficient to compute four quantities involving 1, x^2, y^2, and z^2, which we combine to get the volume, mass, and inertia tensor as shown in the section entitled "Inertia Tensor, Superquadric Ellipsoid."

Let

$$\underline{\underline{I}}_{\text{E}} \equiv \begin{pmatrix} \rho V_{\text{E}} \\ i_{1\text{E}} \\ i_{2\text{E}} \\ i_{3\text{E}} \end{pmatrix} = \rho \iiint_{\text{region}} \begin{pmatrix} 1 \\ x^2 \\ y^2 \\ z^2 \end{pmatrix} \textbf{\textit{dxdydz}}.$$

We will provide a superquadric coordinate system for the ellipsoids and the toroids, as radial "shells" to perform the integration. The shells are parameterized by "r" for radius and u and v for the surface. To change

the coordinates from x, y, z to u, v, r, we need to compute the determinant of the Jacobian matrix $\underline{\underline{J}}$ (which we will describe shortly):

$$\underline{i}_{\text{E}} = \rho \int_{r_{\min}}^{r_{\max}} \int_{v_{\min}}^{v_{\max}} \int_{u_{\min}}^{u_{\max}} \begin{pmatrix} 1 \\ x^2 \\ y^2 \\ z^2 \end{pmatrix} \det J \, du \, dv \, dr.$$

Superquadric Ellipsoids

For the superquadric ellipsoids, the eight shells are centered at the origin, parameterized by "r" for radius. In each octant of the x, y, z coordinate system, the superquadric shape is expressed via

$$x(u, v) = \pm \, ra_1 \, \cos(u)^e \, \cos(v)^n,$$

$$y(u, v) = \pm \, ra_2 \, \cos(v)^n \, \sin(u)^e,$$

$$z(u, v) = \pm \, ra_3 \, \sin(v)^n,$$

$$0 \le r \le 1, \quad 0 \le v \le \pi/2, \quad 0 \le u \le \pi/2.$$

Since the density is the same in each octant, we compute the integral in the first octant (where the signs are all positive) and multiply by eight. We need to compute the Jacobian determinant, multiply out x^2, y^2, and z^2 in terms of sin and cos, and then simplify using $\beta(\ )$ functions. Our

Figure 5. The eight octants for the superquadric ellipsoid. We integrate from the origin, where $r = 0$, to the surface, where $r = 1$.

integration limits are from 0 to 1 for r, to get all of the "shells," and from 0 to $\pi/2$ for the surface parameters:

$$i_E = 8\rho \int_0^1 \int_0^{\pi/2} \int_0^{\pi/2} \begin{pmatrix} 1 \\ x^2 \\ y^2 \\ z^2 \end{pmatrix} \det J \, du \, dv \, dr.$$

The Jacobian matrix is given by

$$J = \begin{pmatrix} \partial x / \partial u & \partial x / \partial v & \partial x / \partial r \\ \partial y / \partial u & \partial y / \partial v & \partial y / \partial r \\ \partial z / \partial u & \partial z / \partial v & \partial z / \partial r \end{pmatrix}.$$

I will spare the reader the expression for the matrix itself, but provide the expression for the determinant of the matrix. Symbolic derivatives can be computed using a symbolic manipulation program such as Mathematica (Wolfram, 1991). The determinant in the first octant is given by

$$\det J = a_1 a_2 a_3 enr^2 \cos(u)^{-1+e} \cos(v)^{-1+2n} \sin(u)^{-1+e} \sin(v)^{-1+n}.$$

Introducing the determinant into our equation for i_E and expanding, we obtain

$$i_E = 8\rho \int_0^1 \int_0^{\pi/2} \int_0^{\pi/2} \begin{pmatrix} a_1 a_2 a_3 enr^2 \cos(u)^{-1+e} \cos(v)^{-1+2n} \sin(u)^{-1+e} \sin(v)^{-1+n} \\ a_1^3 a_2 a_3 enr^4 \cos(u)^{-1+3e} \cos(v)^{-1+4n} \sin(u)^{-1+e} \sin(v)^{-1+n} \\ a_1 a_2^3 a_3 enr^4 \cos(u)^{-1+e} \cos(v)^{-1+4n} \sin(u)^{-1+3e} \sin(v)^{-1+n} \\ a_1 a_2 a_3^3 enr^4 \cos(u)^{-1+e} \cos(v)^{-1+2n} \sin(u)^{-1+e} \sin(v)^{-1+3n} \end{pmatrix} du \, dv \, dr.$$

Note that we have cosine and sine functions in the form mentioned in Appendix C. We can simplify the integral with respect to u using $\beta(\ )$ functions. Thus,

$$
\underline{i}_E = 4\rho \int_0^1 \int_0^{\pi/2} \begin{pmatrix} a_1 a_2 a_3 \, enr^2 \cos(v)^{-1+2n} \sin(v)^{-1+n} \beta\left(\dfrac{e}{2}, \dfrac{e}{2}\right) \\[2mm] a_1^3 a_2 a_3 \, enr^4 \cos(v)^{-1+4n} \sin(v)^{-1+n} \beta\left(\dfrac{3e}{2}, \dfrac{e}{2}\right) \\[2mm] a_1 a_2^3 a_3 \, enr^4 \cos(v)^{-1+4n} \sin(v)^{-1+n} \beta\left(\dfrac{e}{2}, \dfrac{3e}{2}\right) \\[2mm] a_1 a_2 a_3^3 \, enr^4 \cos(v)^{-1+2n} \sin(v)^{-1+3n} \beta\left(\dfrac{e}{2}, \dfrac{e}{2}\right) \end{pmatrix} dvdr.
$$

We also note that we can simplify the integral with respect to v using $\beta(\ )$ functions, so

$$
\underline{i}_E = 2\rho \int_0^1 \begin{pmatrix} a_1 a_2 a_3 \, enr^2 \beta\left(\dfrac{e}{2}, \dfrac{e}{2}\right) \beta\left(n, \dfrac{n}{2}\right) \\[2mm] a_1^3 a_2 a_3 \, enr^4 \beta\left(\dfrac{3e}{2}, \dfrac{e}{2}\right) \beta\left(2n, \dfrac{n}{2}\right) \\[2mm] a_1 a_2^3 a_3 \, enr^4 \beta\left(\dfrac{e}{2}, \dfrac{3e}{2}\right) \beta\left(2n, \dfrac{n}{2}\right) \\[2mm] a_1 a_2 a_3^3 \, enr^4 \beta\left(\dfrac{e}{2}, \dfrac{e}{2}\right) \beta\left(n, \dfrac{3n}{2}\right) \end{pmatrix} dr.
$$

Finally, we note that we can easily simplify the integral of r^n with respect to r:

$$
\underline{i}_E = \rho \begin{pmatrix} \dfrac{2}{3} a_1 a_2 a_3 \, en\beta\left(\dfrac{e}{2}, \dfrac{e}{2}\right) \beta\left(n, \dfrac{n}{2}\right) \\[2mm] \dfrac{2}{5} a_1^3 a_2 a_3 \, en\beta\left(\dfrac{3e}{2}, \dfrac{e}{2}\right) \beta\left(2n, \dfrac{n}{2}\right) \\[2mm] \dfrac{2}{5} a_1 a_2^3 a_3 \, en\beta\left(\dfrac{e}{2}, \dfrac{3e}{2}\right) \beta\left(2n, \dfrac{n}{2}\right) \\[2mm] \dfrac{2}{5} a_1 a_2 a_3^3 \, en\beta\left(\dfrac{e}{2}, \dfrac{e}{2}\right) \beta\left(n, \dfrac{3n}{2}\right) \end{pmatrix}.
$$

Then the four components of $\underline{i}_E$ (in other words ρV_E, i_{1E}, i_{2E}, and i_{3E}) are used to produce the volume and inertia tensor of the superquadric ellipsoids, as shown in the section entitled "Inertia Tensor, Superquadric Ellipsoid."

Superquadric Toroids

The toroids are broken down similarly, into eight "outer" pieces and eight "inner" pieces. In the first octant, the outer piece is given by

$$x_{outer} = a_1 \cos(u)^e \left(\alpha + r \cos(v)^n \right),$$

$$y_{outer} = a_2 \sin(u)^e \left(\alpha + r \cos(v)^n \right),$$

$$z_{outer} = a_3 \, r \sin(v)^n.$$

The inner piece is given by

$$x_{inner} = a_1 \cos(u)^e \left(\alpha - r \cos(v)^n \right),$$

$$y_{inner} = a_2 \sin(u)^e \left(\alpha - r \cos(v)^n \right),$$

$$z_{inner} = a_3 \, r \sin(v)^n.$$

(a)   (b)
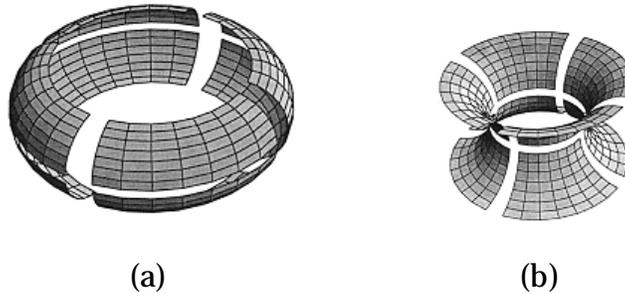
Figure 6 (a) The eight octants for the outer part of the superquadric toroid. We integrate from the torus centerline (not shown) where $r = 0$ outwards to the surface, where $r = 1$. (b) The eight octants for the inner part of the superquadric toroid. We integrate from the torus centerline (not shown) where $r = 0$ inward to the surface, where $r = 1$.

where

$$0 \leq r \leq 1, \quad 0 \leq v \leq \pi/2, \quad 0 \leq u \leq \pi/2.$$

We separately compute the Jacobian matrix, but we need to take the appropriate sign of the determinant of the outer and inner equations:

$$\det J_{\text{outer}} = a_1 a_2 a_3 enr \cos(u)^{-1+e} \cos(v)^{-1+n} \left(\alpha + r\cos(v)^n\right)\sin(u)^{-1+e}\sin(v)^{-1+n},$$

$$\det J_{\text{inner}} = -a_1 a_2 a_3 enr \cos(u)^{-1+e} \cos(v)^{-1+n} \left(\alpha - r\cos(v)^n\right)\sin(u)^{-1+e}\sin(v)^{-1+n}.$$

We need to correctly combine the outer and inner parts into one integral, to obtain the expression for the four terms used in the expression for the inertia tensor of a superquadric toroid. Since the "inner" Jacobian matrix has the opposite handedness from the outer one, we need to change the sign on the determinant to maintain continuity at the common boundary between the two representations.

Thus, we let

$$\underline{i}_T = 8\rho \int_0^1 \int_0^{\pi/2} \int_0^{\pi/2} \begin{pmatrix} 1 \\ x_{\text{outer}}^2 \\ y_{\text{outer}}^2 \\ z_{\text{outer}}^2 \end{pmatrix} \det J_{\text{outer}} + \begin{pmatrix} 1 \\ x_{\text{inner}}^2 \\ y_{\text{inner}}^2 \\ z_{\text{inner}}^2 \end{pmatrix} (-\det J_{\text{inner}})\, du\, dv\, dr.$$

As before,

$$\underline{i}_T \equiv \begin{pmatrix} \rho V_T \\ i_{1T} \\ i_{2T} \\ i_{3T} \end{pmatrix}.$$

The preceding integral expands to

$$
\underline{i}_T = 8\rho \int_0^1 \int_0^{\pi/2} \int_0^{\pi/2} \begin{pmatrix} 2\,a_1 a_2 a_3 \alpha e n r \cos(u)^{e-1} \cos(v)^{n-1} \sin(u)^{e-1} \sin(v)^{n-1} \\ 2\,a_1^3 a_2 a_3 \alpha e n r \cos(u)^{3e-1} \cos(v)^{n-1}(\alpha^2 + 3r^2\cos(v)^{2n})\sin(u)^{e-1}\sin(v)^{n-1} \\ 2\,a_1 a_2^3 a_3 \alpha e n r \cos(u)^{e-1} \cos(v)^{n-1}(\alpha^2 + 3r^2\cos(v)^{2n})\sin(u)^{3e-1}\sin(v)^{n-1} \\ 2\,a_1 a_2 a_3^3 \alpha e n r^3 \cos(u)^{e-1} \cos(v)^{n-1}\sin(u)^{e-1}\sin(v)^{3n-1} \end{pmatrix} du\,dv
$$

and simplifies into $\beta(\ )$ functions

$$
\underline{i}_T = \rho \begin{pmatrix} 2\,a_1 a_2 a_3 \alpha e n \beta\left(\dfrac{e}{2},\ \dfrac{e}{2}\right)\beta\left(\dfrac{n}{2},\ \dfrac{n}{2}\right) \\[2ex] 2\,a_1^3 a_2 a_3 \alpha^3 e n \beta\left(\dfrac{3e}{2},\ \dfrac{e}{2}\right)\beta\left(\dfrac{n}{2},\ \dfrac{n}{2}\right) \ +\ 3\,a_1^3 a_2 a_3 \alpha e n \beta\left(\dfrac{3e}{2},\ \dfrac{e}{2}\right)\beta\left(\dfrac{3n}{2},\ \dfrac{n}{2}\right) \\[2ex] 2\,a_1 a_2^3 a_3 \alpha^3 e n \beta\left(\dfrac{e}{2},\ \dfrac{3e}{2}\right)\beta\left(\dfrac{n}{2},\ \dfrac{n}{2}\right) \ +\ 3\,a_1 a_2^3 a_3 \alpha e n \beta\left(\dfrac{e}{2},\ \dfrac{3e}{2}\right)\beta\left(\dfrac{3n}{2},\ \dfrac{n}{2}\right) \\[2ex] a_1 a_2 a_3^3 \alpha e n \beta\left(\dfrac{e}{2},\ \dfrac{e}{2}\right)\beta\left(\dfrac{n}{2},\ \dfrac{3n}{2}\right) \end{pmatrix}.
$$

Then the four components of $\underline{i}_T$ (in other words ρV_T, i_{1T}, i_{2T}, and i_{3T}) are used to produce the volume and inertia tensor of the toroids.

IV

2-D GEOMETRY AND AND ALGORITHMS

IV

2-D GEOMETRY AND ALGORITHMS

Two-dimensional geometry is an important part of computer graphics. Figure-making, layout of geometric figures on a sheet, and projective geometry are examples of common applications. The Gems in this section focus on techniques for making pictures from two-dimensional entities.

The first, third, and fifth Gems deal with methods of drawing various two-dimensional curves. The first Gem describes how to draw elliptical arcs. The third Gem discusses an efficient technique for circle clipping. The fifth Gem provides a recipe for generating circular arc fillets between two lines.

The second Gem describes techniques for producing well organized figures, helping with the related problems of where to place items and how to connect them.

The fourth, sixth, and seventh Gems present clever improvements upon previous Gems. The fourth and sixth Gems discuss efficient computation of the intersection between two lines, while the seventh Gem discusses the construction of circles tangent to other figures, and related problems.

IV.1

A PARAMETRIC ELLIPTICAL ARC ALGORITHM

Jerry Van Aken and Ray Simar
Texas Instruments
Houston, Texas

Sometimes an arc of a circle or of an ellipse is a better choice than a cubic spline for representing a particular curved shape. Because circles and ellipses are inherently simpler curves than cubics, the algorithms for generating them should also be simpler. This is chiefly why conic splines are popular in applications such as the generation of font outlines, where drawing speed is of critical importance.

This note describes an algorithm for generating points along an elliptical arc. The points are separated by a fixed angular increment specified in radians of elliptical arc. The algorithm is based on a parametric representation of the ellipse. It is particularly inexpensive in terms of the amount of computation required. Only a few integer (or fixed-point) shifts, additions, and subtractions are needed to generate each point—without compromising accuracy.

The Algorithm

The *QtrElips* function in Fig. 1 is a version of the algorithm that uses floating-point operations. An integer-only version is presented later. The first six arguments to the function are the x and y coordinates of the vertices P, Q, and K of a control polygon (a triangle) that defines the curve. The arc beings at P, ends at Q, and is completely contained within the triangle formed by the three points. The last argument, Δt, specifies the (approximate) angular increment in radians between successive points plotted along the curve. This argument is a fractional value in the range $1 \geq \Delta t > 0$.

```
procedure  QtrElips (x_P, y_P, x_Q, y_Q, x_K, y_K, Δt : real)
x_J, y_J, u_x, v_x, u_y, v_y : real ;
i, n, x, y : integer :
begin
    v_x ← x_K − x_Q :
    u_x ← x_K − x_P :
    v_y ← y_K − y_Q :
    u_y ← y_K − y_P :
    x_J ← x_P − v_x :
    y_J ← y_P − v_y :
    u_x ← u_x √(1 − ¼Δt²) − ½v_xΔt ;
    u_y ← u_y √(1 − ¼Δt²) − ½v_yΔt ;
    n ← ⌊½π/Δt⌋ :
    for i ← 0 to n do
        DrawPoint (x ← round(v_x + x_J),  y ← round(v_y + y_J)) ;
        u_x ← u_x − v_xΔt ;
        v_x ← v_x − u_xΔt ;
        u_y ← u_y − v_yΔt ;
        v_y ← v_y − u_yΔt ;
        endloop
end
```

Figure 1. Quarter ellipse algorithm.

The *QtrElips* function was used to draw the curve shown in Fig. 2. Also shown is the control polygon that defines the arc. The arc is tangent to the sides of the control polygon at vertices P and Q. The arc is an affine transformation of a quarter circle; it spans $\pi/2$ radians of elliptical arc. We can refer to this curve as a *quarter ellipse.* (If you alter the function to assign to variable n the value $2\pi/\Delta t$, it will draw the entire ellipse.)

The function in Fig. 1 represents the arc as a series of $n = \lfloor 1 + \frac{1}{2}\pi/\Delta t \rfloor$ individual pixels. The smaller Δt is, the more pixels are drawn. Each call to *DrawPoint* turns on the pixel at the specified integer screen coordinates, x and y. The algorithm's inner loop calculates the coordinates of each point on the arc to floating-point precision, but calls the *round* function to round off to the nearest integer pixel coordinates. The ellipse is centered at coordinates (x_J, y_J). Variables u_x and v_z are used to
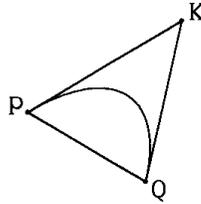
Figure 2 An elliptical arc and its control polygon.

generate the displacement in x of each point from the center; u_y and v_y generate the corresponding y displacement.

The algorithm in Fig. 1 is based on the principle illustrated in Fig. 3: an ellipse is formed by two mutually perpendicular sinusoidal oscillations moving at the same frequency. This is referred to as a *Lissajous* figure after the 19th-century French physicist. The bottom sinusoid in Fig. 3 generates the ellipse's x coordinates, and the sinusoid on the right generates the y coordinates.

The origin-centered ellipse in Fig. 3 is represented by the following parametric equations:

$$x(t) = X \sin(t + \phi_x),$$

$$y(t) = Y \sin(t + \phi_y). \tag{1}$$
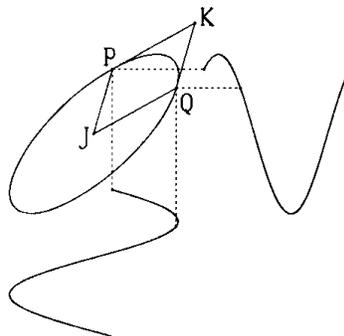
Figure 3. An ellipse and its two generating sinusoids.

The shape of the ellipse is determined by the phase and amplitude relationships of the two sinusoids, where X and Y are the amplitudes, and ϕ_x and ϕ_y are the phase angles. The full ellipse is formed as independent parameter t increases from 0 to 2π radians.

Digital Generation of Sinusoids

Explicitly calculating the sine and cosine terms in Eq. (1) could be quite time-consuming. As an alternative, consider this highly efficient but somewhat inaccurate method for approximating sinusoids:

$$x_n = x_{n-1} - y_{n-1}\Delta t,$$

$$y_n = y_{n-1} + x_n\Delta t. \tag{2}$$

The circle algorithm based on Eq. (2) is well known and has been described many times in the literature—refer to Cohen (1969), Newman and Sproull (1979), Blinn (1987), and Paeth (1990). Given the coordinates (x_{n-1}, y_{n-1}) of a point on a circle (actually, an ellipse that approximates a circle), these expressions calculate a new point (x_n, y_n) that is also on the (approximate) circle. Beginning at initial coordinates (x_0, y_0), these equations are evaluated iteratively to generate a series of points that lie roughly along a circular arc. Successive values of x_n and y_n approximate two sinusoids that have a relative phase difference of $\pi/2$. Note that the x_n value computed in the first equation above appears on the right-hand side of the second equation.

Parameter Δt in Eq. (2) is approximately the angular increment in radians between successive points. The range is $0 < \Delta t \leq 1$. The accuracy improves as Δt is made smaller, which also has the effect of spacing the points more closely together.

Equations (2) are an approximation to the ideal rotation

$$x_n = x_{n-1}\cos\Delta t - y_{n-1}\sin\Delta t,$$

$$y_n = x_{n-1}\sin\Delta t + y_{n-1}\cos\Delta t.$$

The full circle is generated as n increases from 0 to $\lfloor 2\pi/\Delta t \rfloor$. In terms of

initial coordinates (x_0, y_0), the effect of n such iterations can be represented as

$$x_n = x_0 \cos n\Delta t - y_0 \sin n\Delta t,$$

$$y_n = x_0 \sin n\Delta t + y_0 \cos n\Delta t.$$

If we had similar expressions for the effect of n iterations of Eq. (2), we could analyze the error in the approximation and try to compensate for it.
  In fact, the result of n iterations of the approximate rotation expressed in Eq. (2) can be shown to be

$$x_n = x_0 \cos n\alpha - \left(\frac{y_0 - \frac{\Delta t}{2} x_0}{\sqrt{1 - \frac{1}{4}\Delta t^2}} \right) \sin n\alpha,$$

$$y_n = \left(\frac{x_0 - \frac{\Delta t}{2} y_0}{\sqrt{1 - \frac{1}{4}\Delta t^2}} \right) \sin n\alpha + y_0 \cos n\alpha \tag{3}$$

Whereas Δt is the approximate angular increment, the precise increment in the equations above is $a = 2\arcsin(\Delta t/2)$ radians. For the sake of brevity, the proof of this result is not presented here. The interested reader is referred to the detailed derivation in an earlier (1988) paper by the authors.
  Relating Δt to the exact angular increment, α, are the equivalent expressions

$$\sin\frac{\alpha}{2} = \frac{\Delta t}{2},$$

$$\cos\frac{\alpha}{2} = \sqrt{1 - \frac{1}{4}\Delta t^2}.$$

Observe that if α represents the length of an arc on the unit circle, Δt represents the length of the corresponding chord. Accordingly, as Δt

grows smaller, the approximation $\alpha \approx \Delta t$ improves. For the special case $\Delta t = 1$, angle α is precisely $\pi/3$ radians.

   Based on Eq. (3), the error in the equation for either x_n or y_n can be canceled out merely by altering the initial value y_0 or x_0, respectively. For instance, define

$$
\chi_0 = x_0 \sqrt{1 - \tfrac{1}{4}\Delta t^2} + \frac{\Delta t}{2} y_0. \tag{4}
$$

Replacing all instances of x_0 with χ_0 in Eq. (3) yields an *exact* means of calculating y_n:

$$
x_n = \left(x_0 \sqrt{1 - \tfrac{1}{4}\Delta t^2} - \frac{\Delta t}{2} y_0 \right) \cos n\alpha - \left(y_0 \sqrt{1 - \tfrac{1}{4}\Delta t^2} - \frac{\Delta t}{2} x_0 \right) \sin n\alpha,
$$

$$
y_n = x_0 \sin n\alpha + y_0 \cos n\alpha. \tag{5}
$$

The significance of this result is that each of the two sinusoids that generate the ellipse in Fig. 3 can be calculated *precisely* by means of the simple iteration represented by Eq. (2). (A separate copy of the iteration formulas is needed for each of the two dimensions; were the arc *three*-dimensional, then three copies would be needed.) The error in the approximation can be entirely eliminated by modifying only the initial values. The inner loop calculation itself remains unmodified (and the determinant of the corresponding rotation matrix remains precisely unity).

Conjugate Diameters

Equation (1) can be restated in a form similar to that of the expression for y_n in Eq. (5):

$$
x(t) = X_P \cos t + X_Q \sin t,
$$

$$
y(t) = Y_P \cos t + Y_Q \sin t. \tag{6}
$$

The parameters in Eq. (l) are related by the formulas

$$X = \sqrt{X_P^2 + X_Q^2}, \qquad Y = \sqrt{Y_P^2 + Y_Q^2},$$

$$\phi_x = \arctan\frac{X_Q}{X_P}, \qquad \phi_y = \arctan\frac{Y_Q}{Y_P}.$$

This parameterization is described by Tran-Thong (1983), Bagby (1984), and Foley *et al.* (1990). For the origin-centered ellipse in Fig. 3, the geometric interpretation of Eq. (6) is that the point on the ellipse corresponding to $t = 0$ is at $P = (X_P, Y_P)$; the point corresponding to $t = \pi/2$ is at $Q = (X_Q, Y_Q)$. Over the interval from 0 to $\pi/2$, the arc is a blend of P and Q, with the respective contributions at each point determined by the cosine and sine functions. The two diameters formed by extending lines from points P and Q through the center to the other side of the ellipse are referred to as *conjugate diameters* of the ellipse, as described by Tran-Thong (1983) and Bagby (1984). In the proposed CGI standard, the elliptical arc primitive is specified in terms of conjugate diameters.

Figure 4 shows that defining an ellipse in terms of its conjugate diameters is equivalent to defining the ellipse in terms of an enclosing parallelogram. The ellipse inscribed in the parallelogram is an affine transformation of a unit circle inscribed in a square. The circle touches the square at the midpoint of each side, and the dotted lines shown connecting the midpoints of opposing sides are perpendicular diameters of the circle. The same affine transformation that transforms the circle into an ellipse also transforms the enclosing square into a parallelogram.
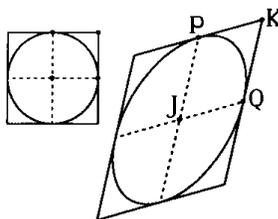
Figure 4. Affine transformation of a unit circle inscribed in a square into an ellipse inscribed in a parallelogram.

The ellipse's two conjugate diameters, shown as dotted lines, are no longer perpendicular, but still connect the midpoints of opposing sides, and remain parallel to and equal in length to the other two sides of the parallelogram. The diameters divide the larger parallelogram into four smaller ones, each of which contains one-quarter of the ellipse. One of the smaller parallelograms in Fig. 4 appears in Fig. 3 as well.

  A conic spline is most conveniently described in terms of its control polygon (a triangle), as shown in Fig. 2. This polygon corresponds in Fig. 3 to the triangle defined by points P, Q, and K. These points also define a parallelogram, for which ellipse center point J is the fourth vertex. Given three vertices P, Q, and K of a parallelogram, the coordinates of the fourth vertex, J, can be calculated as

$$X_J = X_P + X_Q - X_K,$$

$$Y_J = Y_P + Y_Q - Y_K.$$

The center is translated to the origin in order to use Eq. (6) to generate the ellipse.

Simplifying the Computations

An efficient version of the elliptical arc algorithm written in the C language appears as the *qtr_elips* function in the Appendix. This function is similar in form to the listing in Fig. 1, but all floating-point calculations have been replaced by integer(or fixed-point) calculations. A fixed-point value is represented as a 32-bit number with 16 bits of fraction that lie to the right of the binary point.

  On machines for which multiplications are more lengthy operations than shifts, the inner-loop evaluation of Eq. (2) can be sped up by forcing angular increment Δt to be an integral power of 2. Newman and Sproull (1979) suggest that this method be used to speed up the circle algorithm. If $\Delta t = 1/2^m$, where $m \geq 0$, Eq. (2) translates to the following two statements in C:

```
x − = y ≫ m;
y + + x ≫ M;
```

Variables x and y are fixed-point values, and m is an integer.

Similarly, the calculation of χ_0 in Eq. (4) can be sped up. Expanding the square root calculation in a Taylor series yields

$$\chi_0 = x_0 \left(1 - \frac{1}{8}\Delta t^2 - \frac{1}{128}\Delta t^4 - \frac{1}{1024}\Delta t^6 - \frac{1}{16384}\Delta t^8 - \ldots \right) + \frac{\Delta t}{2}y_0.$$

Assuming again that $\Delta t = 1/2^m$ and that variables x and y have been assigned the coordinates of the initial point on a circle, the following statements in C perform the modification of the initial x value:

```
x − = (w = x ≫ (2*m + 3));        /* cancel 2nd-order error */
x − = (w = w ≫ (2*m + 4));        /* cancel 4th-order error */
x − = w ≫ (2*m + 3);              /* cancel 6th-order error */
x +  = y ≫ (m+ 1);                /* cancel lst-order error */
```

Fixed-point variable w is a temporary. Note that this code does not include cancellation of the eighth-order error term. The decision not to include this term is somewhat arbitrary, and the code necessary to cancel higher-order terms can easily be added if additional precision is required. With the code as given above, the worst-case error occurs for $m = 0$, for which the error in the calculation of χ_0 is approximately $(5/16384)x_0$; the full magnitude of this error propagates into y, which represents the sinusoid.

The x and y variables in the code above correspond to the coordinates generated by the circle algorithm—not the ellipse algorithm. The ellipse algorithm requires two sinusoids, and a separate instance of the circle algorithm is needed to generate each sinusoid. To help distinguish circle coordinates from ellipse coordinates, the names of circle coordinates x and y have been changed in the *qtr_elips* function to u and v. Variables u_x and v_x are the coordinates of the circle that generates the ellipse's x coordinates; u_y and v_y are the coordinates of the circle that generates the ellipse's y coordinates.

See also G1, 57.

IV.2
SIMPLE CONNECTION ALGORITHM FOR 2–DÊDRAWING

Claudio Rosati
IRIS s.r.l.
Paliano, Italy

Introduction

Many editing processes (electric and electronic drawing, representation of numerical algorithms and flowcharts, etc.) need to connect two objects using a path of orthogonal segments rather than a straight line (which would make the picture hard to interpret, see Fig. 1).

A simple way to create this path is to draw it point by point and store each point's coordinate value in the picture data base.

This method, although simple, has the main drawback of keeping the user's attention on the physical path rather than on the logical one (one object connected to another), reducing the overall productivity.

A better solution is to user-specify only the path endpoints and let the system compute it.

This Gem presents a simple algorithm for computing the minimal orthogonal path connecting two objects. Only the endpoint's coordinates, the start-point outward direction, and the end-point inward direction are required.

Terms and Definitions

Hereinafter we call *object* the bounding box surrounding the real object and pads the connecting points (see Fig. 2).

Each pad can be *In* or *Out*, according to data flow direction (e.g., the data flow is entering the object or is leaving it), and its direction will be
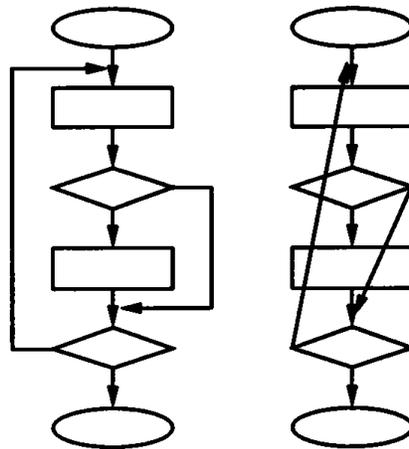
Figure 1. Orthogonal versus straight connections.
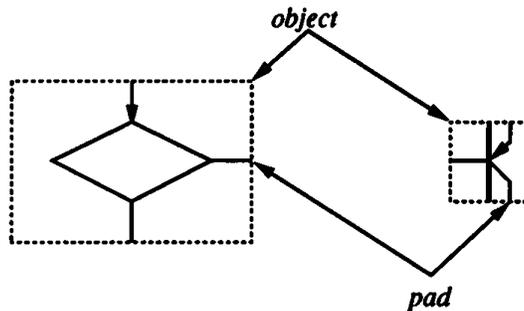
Figure 2. Objects and pads.

UP, DOWN, LEFT, or *RIGHT* according to the pad position in the object (see Fig. 3).

A minimal orthogonal path between two objects is the simplest orthogonal-segments path not crossing the objects, starting from an Out pad and ending at an In pad.

Translate and Rotate Algorithm

When connecting two objects with a minimal orthogonal path, we can see a rotation symmetry, so that the path shape depends solely on the relative position of the Out and In pads (rotation invariance property, see Fig. 4).
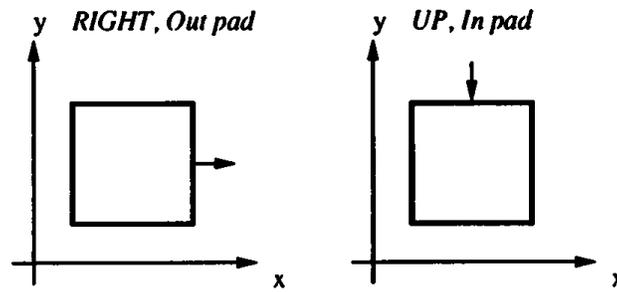
Figure 3. Pad characteristics.
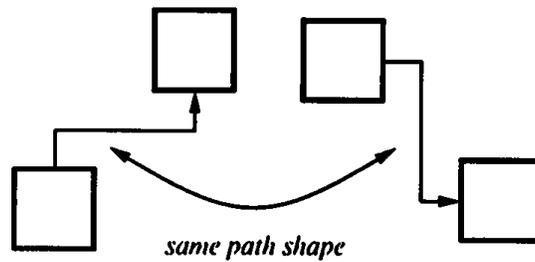
Figure 4. Rotation invariance of path shape.
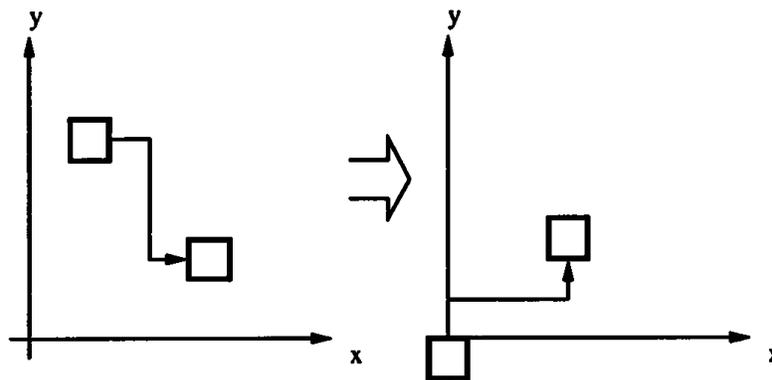
Figure 5. Translate and rotate.

start	rot.	$P' = P \cdot T \cdot R$	$P = P' \cdot R^{-1} \cdot T^{-1}$	end
UP	0°	$x' = x - x_0$ $y' = y - y_0$	$x = x' + x_0$ $y = y' + y_0$	UP $\Rightarrow$ UP RIGHT $\Rightarrow$ RIGHT LEFT $\Rightarrow$ LEFT DOWN $\Rightarrow$ DOWN
RIGHT	90°	$x' = -(y - y_0)$ $y' = x - x_0$	$x = y' + x_0$ $y = x' + y_0$	UP $\Rightarrow$ LEFT RIGHT $\Rightarrow$ UP LEFT $\Rightarrow$ DOWN DOWN $\Rightarrow$ RIGHT
DOWN	180°	$x' = -(x - x_0)$ $y' = -(y - y_0)$	$x = -x' + x_0$ $y = -y' + y_0$	UP $\Rightarrow$ DOWN RIGHT $\Rightarrow$ LEFT LEFT $\Rightarrow$ RIGHT DOWN $\Rightarrow$ UP
LEFT	270°	$x' = y - y_0$ $y' = -(x - x_0)$	$x = -y' + x_0$ $y = x' + y_0$	UP $\Rightarrow$ RIGHT RIGHT $\Rightarrow$ DOWN LEFT $\Rightarrow$ UP DOWN $\Rightarrow$ LEFT

Figure 6. Transformations table.

Now if we first translate the origin of the coordinate system in the Out pad position and then rotate it so that the y axis will have the same direction as the pad (see Fig. 5), the problem is reduced to the computation of the minimal orthogonal path connecting an *UP* Out pad positioned at the axis origin with an In pad.

The direct and inverse transformations relative to the Out pad direction as well as the new In pad direction are shown in Fig. 6, with the Out pad direction in the first column, the corresponding rotation angles in the second, the direct and inverse transformation formulas in the third and fourth, respectively, and the In pad direction transformation rules in the last column.

In this situation the paths connecting all In pads are reduced to one of 21 possibilities, depending on the pad's direction. These connections (see Fig. 7) can be easily derived from the In pad direction, from its position relative to the starting object upper edge and relative to the four vertical areas limited by the lines passing on the object left and right edges, and over the Out pad. Figure 7 shows the starting object areas (T, B, and 1, 2, 3, 4) and the *decision tree* that brings to the 21 path shapes, switching first on the In pad direction and then on the In pad position.

The lines passing over the pad and over the object upper edge divide the plane in regions having symmetrical shape properties, so that the 21
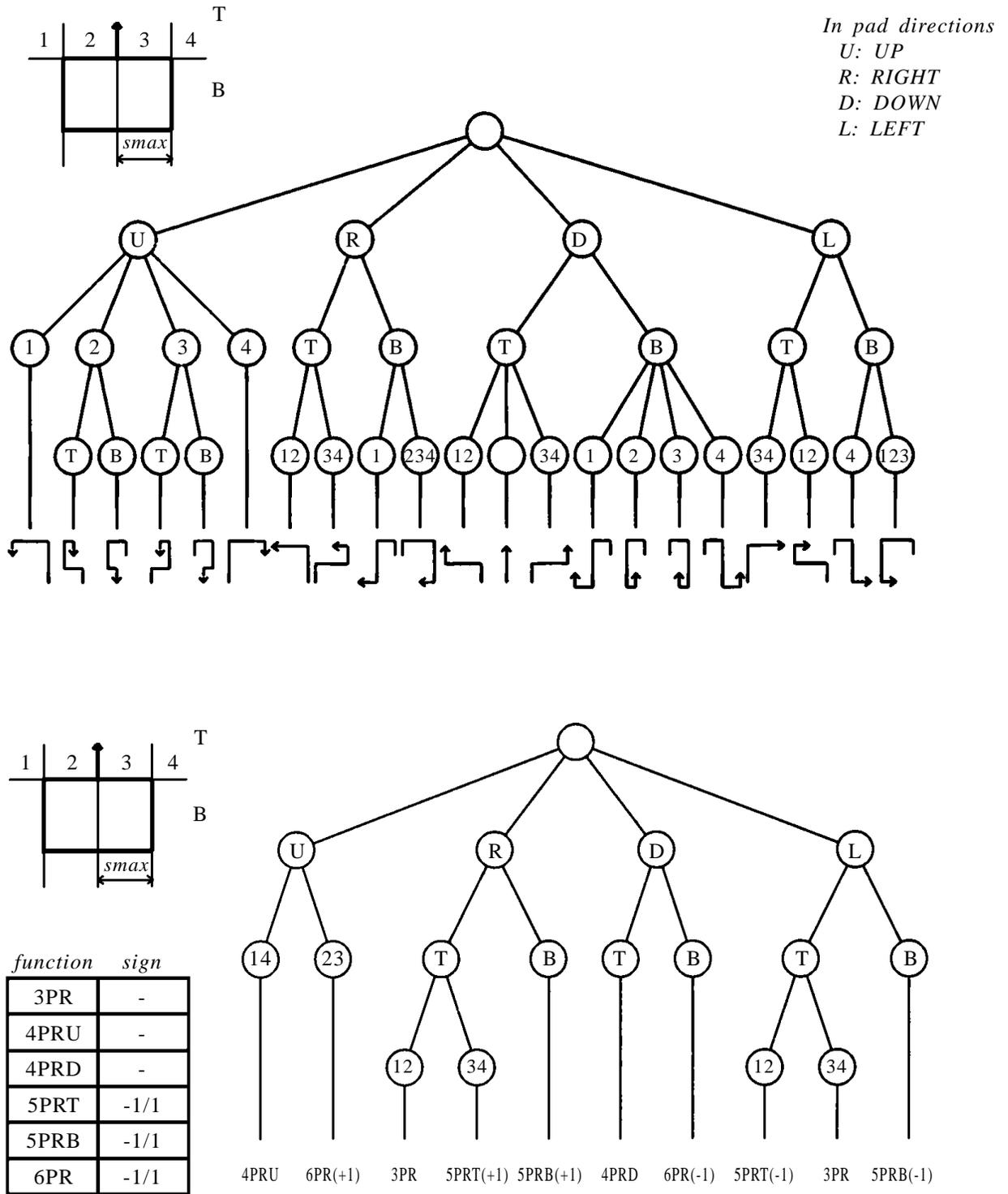
Figure 8. Reduced decision tree.

paths can be generated using only six functions, grouping them by number of points and shape similarity (see Fig. 8).

  At this point the computing algorithm can easily be derived:

    Pad: **record** [
          position: **point**;
          direction: **integer**;
      ]
    p: **array** [0..5] **of point**;
    inPad, outPad: Pad;

    DirectTransform(outPad.direction, inPad);
    select inPad.direction **from**
        UP: **if** *In pad is out of left and right edges*
                              then 4PU (inPad.position, p);
                              else 6P(+1 inPad.position, p);
        RIGHT: **if** In *pad is under the upper edge*
                              then 5PB(+1, inPad.position, p);
                              else if *In pad is on left side of outPad*
                              then 3P(inPad.position, p);
                              else 5PT(+1, inPad.position, p);
        DOWN: **if** *In pad is over the upper edge*
                              then 4PD(inPad.position, p);
                              else 6P(–1, inPad.position, p);

        LEFT: **if** *In pad is under the upper edge*
                              then 5P(–1, inPad.position, p);
                              else if *In pad is on left side of outPad*
                                  then 5PT(–1, inPad.position, p);
                                  else 3P(inPad.position, p);
      InverseTransform(outPad.direction, p);

where DirectTransform( ) transform; the In pad position according to the Out pad direction using the Fig. 6 transformation rules; InverseTransform( ) transforms the path points p[ ] using the inverse transformation rule of Fig. 6 selected by the original (untransformed) Out pad direction; the six shape functions compute the path points p[ ] according to the possible sign and to the transformed In pad direction.

For the implementation of the six shape functions, see the C code at the end of this book.

Overcrossing Correction

In some situations the computed path can overcross one of the two connected objects (see Fig. 9a).

The problem can be solved in two ways. The first and simplest method is to draw the path first and the two objects after, so that the overcrossing segment will be hidden (see Fig. 9b). This is a good solution for simple drawings with no more than a few dozen objects (or in a very fast computing system).

The second way requires moving the path segment out of the crossed object. Figure 10 shows some of the paths involved in this situation (the others being symmetrical).

From this figure it is possible to derive the correcting algorithm:

> *search the segment to be moved;*
> **if** *next segment* != *last segment* **or**
>   *previous segment direction* != *next segment direction*
>   **then** *move toward previous segment direction;*
>   **else** *move against previous segment direction;*

The implementation of the correcting algorithm requires a line-box crossing test in order to find the segment to be moved. This test can easily be derived from any known line clipping algorithm, as shown in the C code listing.
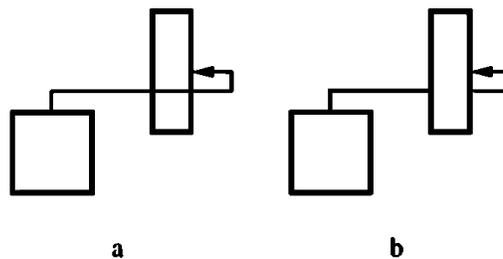
Figure 9. Simple solution of overcrossing problem: connect & redraw.
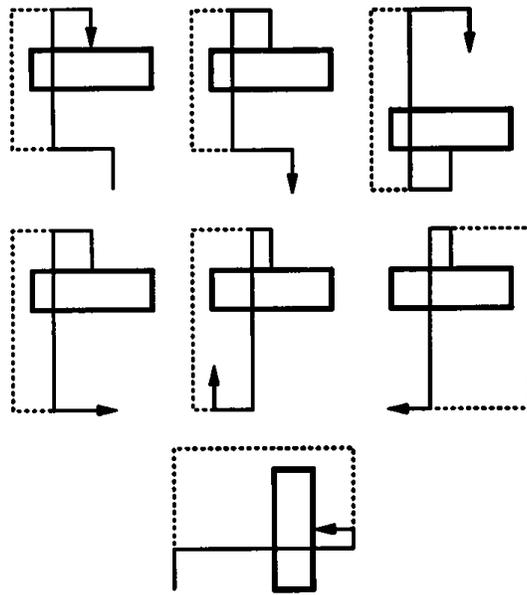
Figure 10. Overcrossing problem: paths correction.
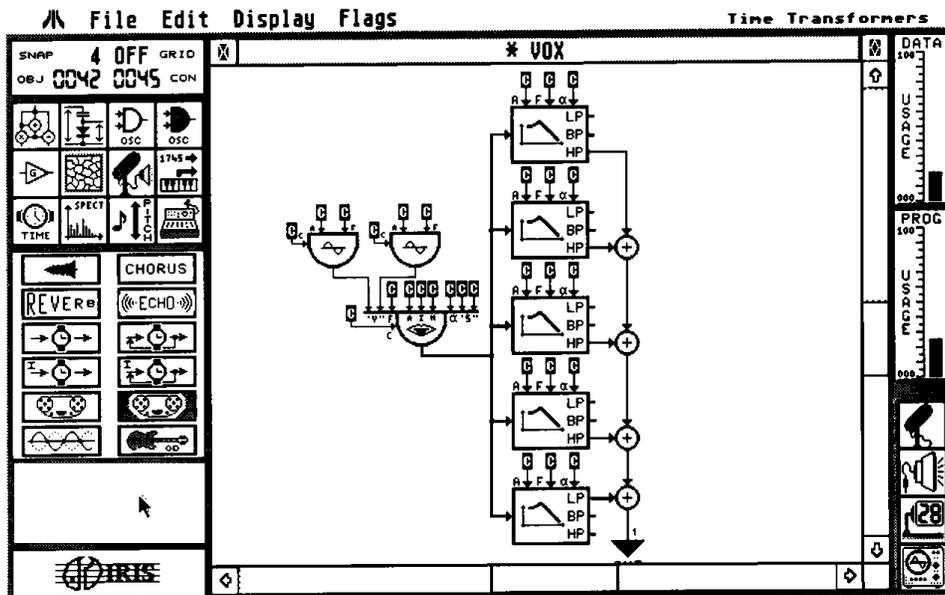
Figure 11. IRIS DSP patch editor.

Conclusions and Acknowledgments

The technique here outlined is a powerful tool in the repertoire of 2-D drawing systems, computing a simple minimal orthogonal path between two objects in a 2-D drawing on the basis of the connection endpoints only. The method is fast enough to be implemented even in a small computer system and does not require complex or large data structures.

In our Musical Audio Research Station, this technique is used for the editor of realtime audio algorithms (see Fig. 11). The editor can manage about a thousand objects with $3 \div 16$ pads each, and the user can move an object and its connections with realtime visual cues.

I would like to thank my colleagues Fabio Armani, Emmanuel Favreau, and Vincenzo Maggi for their help, and in addition, Fabio Armani for having developed the first implementation of the algorithm.

A FAST CIRCLE CLIPPING ALGORITHM

Raman V. Srinivasan
SDRC
Milford, Ohio

Introduction

Clipping a circle to a rectangular region potentially results in four circular arc segments. The algorithm described here uses a parametric approach. A circle may be represented parametrically as

$$x = x_c + R \cos(\theta),$$

$$y = y_c + R \sin(\theta),$$

where (x,y) is a point on the circle, (x_c, y_c) is the center, R is the radius, and θ, the parameter, is the angle measured counterclockwise from the x-axis to the radius vector at (x, y). For a complete circlc θ goes from 0° to 360°. Circular arcs are typically defined by start and end θ values in addition to the center and radius. The following algorithm basically involves analytically determining the start and end parameter values for each clipped segment (arc), and then culling these segments out of the circle.

Algorithm

First, a trivial rejection test is carried out using the bounding square of the circle as follows (Num_Segments is the number of resulting arcs after

clipping):

> **if** X_c + R < Left_Bound **or** X_c – R > Right_Bound **or**
> Y_c + R < Low_Bound or Y_c – R > Up_Bound
> **then**
> Num_Segments ← 0;
> **return**;

If the circle is not trivially rejected, parameter values are computed for the intersection points of the circle and clip boundaries. Only those clip boundaries that are crossed by the circle are considered. If none of the boundaries is crossed—that is, there are no clipped segments—the entire circle is inside the clip boundary and no further processing is required. However, if there are boundaries that are crossed, then for each such boundary the start and end angular parameters of the clipped arc are recorded. Care should be taken here to ensure that the angles lie in the range 0° to 360°. Hence, if the circle is clipped against the right bound-

Half-intersection angles
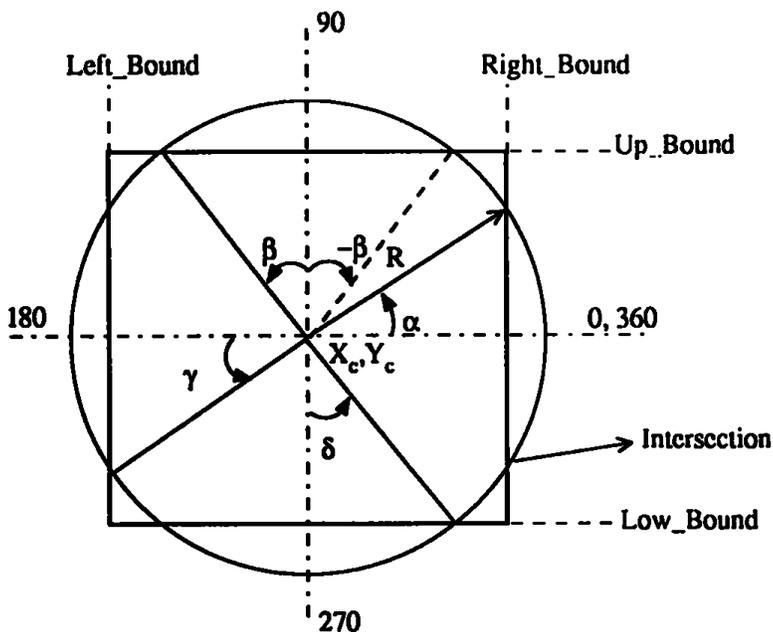
α = arccos (Right_Bound - Xc)/R
β = arccos (Up_Bound - Yc)/R
γ = arccos (Xc - Left_Bound)/R
δ = arccos (Yc - Low_Bound)/R

Figure 1.

ary, the clipped portion is stored as two arcs, 0° to α° and $(360 - \alpha)$° to 1360°, where α is one half the angle subtended by the clipped portion at the center. Refer to Fig. 1 for half-intersection angle computations. This process follows in pseudo-code for the right and upper clip boundaries. Note that with each intersection a type field is used to indicate if it is the start or end of a segment.

```
n ← 0; // number of intersections
if X_c + R > Right_Bound then
begin
    Compute α;
    n ← n + 1; Intersection[n].Angle ← 0; Intersection[n].Type ← START;
    n ← n + 1; Intersection[n].Angle ← α; Intersection[n].Type ← END;
    n ← n + 1; Intersection[n].Angle ← 360 − α; Intersection[n].Type ←
    START;
    n ← n + 1; Intersection[n].Angle ← 360; Intersection[n].Type ← END;
    end;
if Y_c + R > Up_Bound then
begin
    Compute β;
    if (90 − β) < 0 then
    begin
        n ← n + 1; Intersection[n].Angle ← 360 + (90 — β);
        Intersection[n].Type ← START;
        n ← n + 1; Intersection[n].Angle ← 360; Intersection[n].Type ←
        END;
        n ← n + 1; Intersection[n].Angle ← 0; Intersection[n].Type ←
        START;
        end;
        else
        begin
            n ← n + 1; Intersection[n].Angle ← 90 — β;
            Intersection[n].Type ← START;
            end;
    n ← n + l; Intersection[n].Angle ← 90 + β; Intersection[n].Type ←
    END
    end;
```

The left and lower bounds are similarly processed.

```
if n = 0 then // there are no intersections ⇒ nothing is clipped
begin
Num_Segments ← 1; Visible_Segment[1].Start ← 0; Visible_
Segment[l].End ← 360;
return;
end;
```

If there is one or more clipped segments ($n > 0$), the visible segments are extracted by culling out the clipped portions. First, all the intersections are sorted in increasing order of the angle parameter. Since it is possible that two clipped portions may overlap (see Fig. 2), an overlap index is maintained, such that during traversal of the sorted list, this index is incremented on encountering a crossing type of START and decremented on a crossing type of END. A value of zero for the overlap index denotes that we are entering a visible segment of the circle and must be added to the list of visible segments. The pseudo-code that follows performs these tasks.
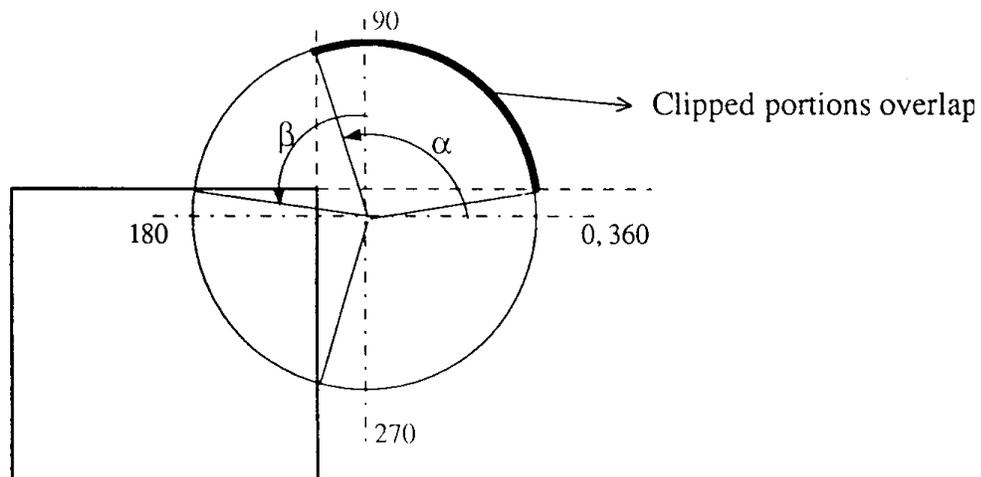
Figure 2.

```
    Sort Intersection array in increasing order of angles;
        Num_Segments ← 0; Overlap ← 0; prev ← 0;
        for i ← 1, n do
        begin
            if Overlap = 0 then
                if Intersection[i].Angle > prev then
                begin
                    Num_Segments ← Num_Segments + 1;
                    Visible_Segment[Num_Segments].Start ← prev;
                    Visible_Segment[Num_Segments].End ←
                    Intersection[i].Angle;
                    end;
            if Intersection[i].Type = START then
                Overlap ← Overlap + 1;
                else
                    Overlap ← Overlap — l;
            prev ← Intersection[i].Angle;
            end;
        if prev < 360 then
        begin
            Num_Segments ← Num_Segments + 1;
            Visible_Segment[Num_Segnlents].Start ← prev;
            Visible_Segment[Num_Segments].End ← Intersection[i].Angle;
            end;
```

The worst case occurs when all four clip boundaries (or their extensions) are crossed and the center of the circle is either above the upper bound or below the lower bound. The total number of intersections is 12, including the ones added due to $90 - \beta$ going below $0°$ or $270 + \delta$ exceeding $360°$. The computational effort required for this case is four divisions, four cosine inverse computations and 39 comparisons (not including sorting).

Notes

The speed of this algorithm was found to be satisfactory for circle rubber-banding and dragging applications where response time is critical.

The algorithm is efficient since, in addition to a trivial rejection test, it detects clip boundaries that are crossed prior to performing any computation.

Since this algorithm operates in the parameter space of the circle as opposed to a scan converted or image space, it is well suited to vector devices such as plotters. The accuracy of the intersection calculations are limited only by the precision of the cosine inverse computation.

See also G1, 51.

EXACT COMPUTATION OF
2-D INTERSECTIONS

Clifford A. Shaffer and Charles D. Feustel
Virginia Tech
Blacksburg, Virginia

In *Graphics Gems II,* Mukesh Prasad (1991) presents an efficient algorithm for determining the intersection point for two line segments in 2-D. His approach has the additional benefit that it can easily be modified to use purely integer arithmetic. However, the resulting intersection point is represented by coordinates using either floating-point or fixed-precision integers. Such implementations have major shortcomings with respect to numeric stability of the operation (see Hoffman, 1989, for a detailed discussion of the potential pitfalls and several suggested solutions). Either the intersection point must be represented using increased precision, or else the intersection point may be only an approximation to the true intersection point. Reduced accuracy leads to topological inconsistencies such as query points that are determined to fall within each of a pair of input polygons, but not within their intersection. A series of cascaded intersections (e.g., the intersection of polygons A and B intersected with C, intersected with D, . . .; alternatively a series of generalized clip operations on a set of line segments) further increases these problems, since either the accuracy becomes arbitrarily small, or the precision required to represent a vertex becomes unbounded.

We present a technique for exact representation of the intersections between line segments in 2-D with vertices of a fixed input resolution. In particular, we show how to represent a clipped line segment using bounded rational arithmetic, provided that both the clipping and clipped segments are defined by endpoints measured to fixed precision. This method allows a line segment to be clipped an arbitrary number of times

without increasing storage requirements, yet yielding the correct result. Equivalently, we recognize that the output from a series of polygon intersection operations must be composed of pieces of the line segments that make up the input polygons. By keeping track of the necessary input line segments (in their original form) along with descriptions of intersection points along these line segments in parametric form, we get exact computations using only about three times the storage needed by traditional methods. In addition, we must be able to calculate intermediate results for n-bit input points to a resolution of $4n + 4$ bits in order to compare parameters. Alternatively, the use of floating-point values to store the parameters, while not guaranteeing correct results, will give better accuracy than simply representing the intersection with floating-point values. Our method is simple to implement and quite efficient.

We restrict input vertex values to be fixed-precision numbers in the range $-16{,}383$ to $16{,}383$—i.e., 14 value bits and one sign bit. The stored precision could be greater if necessary, but by selecting 15-bit precision we simplify the implementation of our rational arithmetic approach.

A line subsegment is some portion of a line segment that is represented by (i) two endpoints with integer coordinates whose precision is 15 bits and (ii) two parameter values. The line segment specified by the vertex points will be referred to as the *a priori* line segment. The parameter values specify the extent of the line subsegment after clipping. These parameters represent positions some fraction of the distance from the first to the second *a priori* vertex points. Initially, input line segments are typically complete *a priori* line segments. Thus, the parameter values of such subsegments will be 0 and 1 (specifying the beginning and the end of the *a priori* line segment, respectively). In addition, a direction flag can be used to indicate whether this particular line subsegment goes from vertex 1 to vertex 2, or vice versa. This would allow for sharing of *a priori* line segment objects among a set of polygons.

We store parameters as two 32-bit quantities, representing the numerator and denominator of the parameter. The denominator will always be positive. Methods storing only the endpoints and a direction bit require $4n + 1$ bits to represent a line segment. Our method requires $4n + 1 + 8(n + 1)$ bits to store the endpoints, parameters and direction flag. In practice, assuming coordinates require 2 bytes and parameters require *4*

bytes, the parameters require 16 bytes per line segment beyond the 8 bytes plus one bit required to store endpoints and direction flag.

The parameter value for the intersection point between two line sub-segments is calculated by computing the intersection of the *a priori* line segments (taken from the structure definitions of the subsegments), and then checking that the intersection actually occurs within the subsegments. The intersection routine returns 0 if there is no intersection, 1 if the line segments intersect in a point, and 2 if they are collinear. A structure is also returned that describes the intersection location by its parameter values along each of the two intersecting *a priori* line segments, and information about the direction of crossing for each parameter—i.e, whether the line intersection is going from in to out, or out to in.

The intersection calculation we use is similar to that suggested by Prasad, although somewhat more efficient. First we check bounding boxes for a quick nonintersection test. Assuming the bounding boxes intersect, we generate the equation for the infinite line from the endpoints for one *a priori* line segment. Substituting the two endpoints from the other line into this equation results in two values, each of which is negative, positive, or zero. If the signs of the two values are the same, then the two lines do not intersect (i.e., both endpoints of the second line are to one side of the first line). If the signs are different, then we repeat the process with the roles of the two lines reversed. If the result of this second test gives two values with different signs, then we know that there is an intersection between the *a priori* line segments. This is stronger than saying that the *a priori* lines extended to infinity intersect.

We use the line equation

$$(X - X_1)(Y_2 - Y_1) - (Y - Y_1)(X_2 - X_1)2$$

when substituting point (X, Y) into the line with endpoints (X_1, Y_1) to (X_2, Y_2). If the result is positive, (X, Y) is to the right of the line segment; if the result is negative, (X, Y) is to the left; and if the result is zero, (X, Y) is on the line. We evaluate this equation four times, first substituting the endpoints of line segment Q into the equation for line segment P, then substituting the endpoints of line segment P into the equation for line segment Q. This is done to calculate the four quantities a, b, c, and d, defined as follows:

$$a = (Q_{1x} - P_{1x})(P_{2y} - P_{1y}) - (Q_{1y} - P_{1y})(P_{2x} - P_{1x}),$$

$$b = (Q_{2x} - P_{1x})(P_{2y} - P_{1y}) - (Q_{2y} - P_{1y})(P_{2x} - P_{1x}),$$

$$c = (P_{1x} - Q_{1x})(Q_{2y} - Q_{1y}) - (P_{1y} - Q_{1y})(Q_{2x} - Q_{1x}),$$

$$d = (P_{2x} - Q_{1x})(Q_{2y} - Q_{1y}) - (P_{2y} - Q_{1y})(Q_{2x} - Q_{1x}).$$

Once we know that there is a proper intersection between the two *a priori* line segments, we must calculate the two intersection parameters, each defined by their numerator and denominator. To do this, we need to solve the following vector equation for parametric variables s and t:

$$(1 - t)P_1 + tP_2 = (1 - s)Q_1 + sQ_2,$$

which can be rewritten as

$$t(P_2 - P_1) + s(Q_1 - Q_2) = Q_1 - P_1.$$

We solve this equation by calculating three determinants. We define $s = \text{sdet}/\text{det}$ and $t = \text{tdet}/\text{det}$, where det, sdet, and tdet are defined as

$$\det \ = (P_{2x} - P_{1x}) \ (Q_{1y} - Q_{2y}) - (P_{2y} - P_{1y}) \ (Q_{1x} - Q_{2x}),$$

$$\text{sdet} \ = (P_{2x} - P_{1x}) \ (Q_{1y} - P_{1y}) - (P_{2y} - P_{1y}) \ (Q_{1x} - P_{1x}),$$

$$\text{tdet} \ = (Q_{1x} - P_{1x})(Q_{1y} - Q_{2y}) - (Q_{1y} - P_{1y})(Q_{1x} - Q_{2x}).$$

By suitable rearrangement, we find that $\det = a - b$, sdet $= a$ and tdet $= -c$. Thus, our work to check if there is a proper intersection between the two line segments by substituting the endpoints into line equations provides most of the calculation required to determine the parameters of the insertion point. Each parameter (s and t) is actually stored as the numerator and denominator of a fraction. It should be easy to see from the preceding equations that if the initial endpoints for lines P and Q require n bits for their representation, than these numerators and denominators will each require at most $2n + 2$ bits.

Given the parameter definition or an intersection point, cascaded clipping operations are supported by the ability to calculate the relative position of two such intersection points along a line segment. In other words, we must determine if one intersection point is to the left or to the right of another intersection point along the line segment. This calculation is straightforward since it is equivalent to deciding which of the iractions N_1/D_1 and N_2/D_2 is greater by comparing the two products $N_1 * D_2$ and $N_2 * D_1$. This requires a (temporary) further doubling of the resolution, to $4n + 4$ bits. The intermediate calculations for comparing the magnitude of the parameters require multiplication of two 32-bit quantities, which can easily be done in software if 64-bit integers are not provided by the compiler (such a routine is provided in our C code). We note, however, that floating-point division can almost always be substituted. Only in the rare case where the two resulting floating-point numbers are equal must double-precision integers be used.

Another useful function is one that determines whether a point falls to the left or right of a line. Our C code contains the function SideOfPoint, which takes a point defined as a parameter along an *a priori* line segment, and a second a *priori* line segment that defines the line to he tested. The function substitutes the parameterized definition of the point into the line equation. By suitable rewriting of this expression, we can cast it as a comparison of two fractions, each with $(2n + 2)$-bit numerators and denominators.

We performed an experiment to compare the time required for our exact intersection representation against the implementation of Prasad. For testing, our code was slightly modified to return the (approximated) intersection point; both our version and Prasad's version were coded to use only integer arithmetic. On both a M68000 machine and a MIPS 3000 machine, our point intersection algorithm required only $\frac{3}{4}$ the time required by Prasad's algorithm. Thus, we conclude that little or no time penalty will result from using our exact arithmetic approach.

See also G2, 7; G3, D.6.

IV.5

JOINING TWO LINES WITH A CIRCULAR ARC FILLET

Robert D. Miller
East Lansing, Michigan

Problem

Given two lines, $\mathbf{l}_1$ $(\overline{p_1 p_2})$ and $\mathbf{l}_2$ $(\overline{p_3 p_4})p$ and a radius, r, construct a circular arc fillet that joins both lines. This algorithm finds the beginning angle and the angle subtended by the arc and the direction in which to draw the arc. New beginning and ending points of the lines will be computed so they will smoothly join the arc.

Method

1. Find the equation in the form $ax + by + c = 0$ for each line, as shown in the following pseudo-code procedure LineCoef. The center of the constructed arc, P_c, must lie at a distance r from both lines.

   Determine the *signed* distance d_1 from $\mathbf{l}_1$ to the midpoint of $\mathbf{l}_2$ and d_2 from $\mathbf{l}_2$ to the midpoint of $\mathbf{l}_1$. The midpoints are used because, in practice, one point may be common to both lines. The signs of d_1 and d_2 determine on which sides of the respective lines the arc center p_c resides. The signed distances are computed in the procedure LineToPoint. (See Fig. 1.)

2. Find $\mathbf{l}_1' \| \mathbf{l}_1$ at d_1 and $\mathbf{l}_2' \| \mathbf{l}_2$ at d_2. The center of the required arc p_c lies at the intersection of $\mathbf{l}_1'$ and $\mathbf{l}_2'$.

3. Compute the beginning and ending points, q_1 and q_2, on the arc. The procedure PointPerp finds the points q_1 so $\overline{q_1 p_c} \perp \mathbf{l}_1$ and q_2 so $\overline{q_2 p_c} \perp \mathbf{l}_2$.
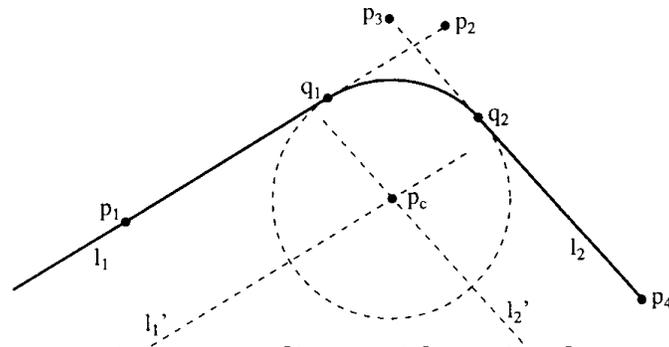
Figure 1. Joining two lines with a circular arc fillet.

4. Find the starting angle, s (with respect to the x-axis). Angle $s = \tan^{-1}(\overline{q_1 p_c})$. The two-argument arctangent is used to uniquely determine a in the range $0 \leq s < 2\pi$. Apply the vector dot product to the directed line segments $q_1 p_c$ and $q_2 p_c$ to find the angle a subtended by the arc from p_c.

5. Use the sign of the vector cross product to determine the direction in which to draw the arc $q_1 q_2$.

6. The line may be extended or clipped at points q_1 and q_2 so the end points of the line nearest the point of intersection of $\mathbf{l}_1$ and $\mathbf{l}_2$ will coincide with the end points of the arc. The fillet will result from drawing a line from p_1 to q_1, the arc from q_1 to q_2, and then the line from q_2 to p_4.

The accompanying cuboid "gem" (Fig. 2) has its corners rounded by this algorithm.

```
π/2: real      ← 1.5707963267949;
π: real        ← 3.14159265358979;
3π/2: real     ← 4.71238898038469;
2π: real       ← 6.28318530717959;

p1, p2, p3, p4, v1, v2: point;
r, xx, yy, sa, a: real;

function arctan2(y, x: real): real;
Two argument arc tangent. 0 < = arctan2 < 2π.
```
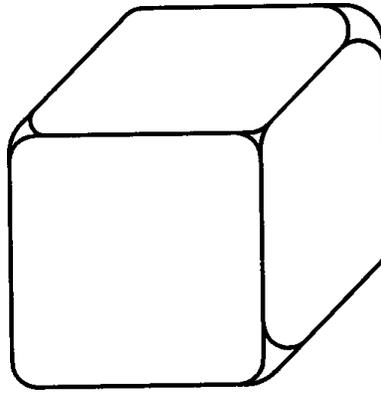
Figure 2.

```
r; real;
begin
    if abs(y) < abs(x) then
        if x ≠ 0 then
        begin r ← arctan(y/x);
            if x < 0 then r ← r + π
            else if y < 0 then r ← 2π + r;
        end
        else
    else
        if y ≠ 0 then
            begin
            r ← arctan(x/y);
            if y > 0 then r ← π/2 − r
            else r ← 3π/2 − r;
            end
    else r ← π/2;
arctan2 ← r;
endproc arctan2.

function cross2(v1, v2: point): real;
begin
    cross2 ← v1.x*v2.y − v2.x*v1.y;
endproc cross2.

function dot2(v1, v2: point): real;
d: real;
begin
```

```
        d ← sqrt((v1.x² + vl.y²)*(v2.x² + v2.y²));
        if d ≠ 0 then dot2 ← arccos((v1.x*v2.x + v1.y*v2.y)/d);
        else dot2 ← 0;
    endproc dot2.


    procedure DrawArc(xc, yc, r, startangle, a: real);
    Draw circular arc in one degree increments
    Center is xc, yc with radius r, beginning at startangle,
    through angle a. If a < 0, the arc is drawn clockwise.
    sindt: real ← 0.017452406; sin 1°
    cosdt: real ← 0 999847695; cos 1°
    a,x,y,xt,yt,sr: real;
    k: integer;
    begin
        a ← startangle:
        x ← r*cos(a); y ← r*sin(a);
        MoveTo(xc + x, yc + y);
        if a ≥ 0 then sr ← sindt else sr ← –sindt;
        for k ← 1 to trunc(abs(a)) do
            begin
            x ← x*cosdt – y*sr;
            y ← x*sr + y*cosdt;
            LineTo(xc + x, yc + y);
            end
    endproc DrawArc.


    procedure LineCoef(var a, b, c: real; p1, p2 point);
    Returns a, b, c in ax + by + c = 0 :for line p1, p2.
    begin
        c ← (p2 x*p1.y) – (p1.x*p2.y);
        a ← p2.y – p1.y;
        b ← pl.x – p2.x;
    end;
    endproc LineCoef.


    function LineToPoint(a, b, c real; p: point):real;
    Returns signed distance from line ax + by + c = 0 to point p.
    d, lp: real;
begin
```

```
    d ← √a² + b² ;
    if d = 0 then lp ← 0
    else lp ← (a*p.x + b*p.y + c)/d;
    LineToPoint ← lp;
end;
endproc LineToPoint.


procedure PointPerp(var x, y: real; a, b, c: real; point);
```
Given line l = ax + by + c = 0 and point p,
compute x, y so p(x, y) is ⊥ to l.
```
d, cp, t, u: real;
begin
    x ← 0; y ← 0; d ← a² + b²; cp ← a*p.y – b*p.x;
    if d ≠ 0 then
        begin
        x ← (–a*c – b*cp)/d;
        y ← (a*cp – b*c)/d;
        end;
endproc PointPerp.


procedure Fillet(var p1, p2, p3, p4:point; r: real;
                 var xc, yc, pa, aa: real);
```

Compute a circular arc fillet between lines l1 (p1 to p2)
and l2 (p3 to p4) with radius r. The arc center is xc, yc.

```
a1, b1, c1, a2, b2, c2, c1p, c2p: real;
d1, d2, xa, xb, ya, yb, d, rr: real;
mp, pc: point;
label XIT;
begin
    LineCoef(a1, b1, c1, p1, p2);
    LineCoef(a2, b2, c2, p3, p4);

    if (a1*b2) = (a2*b1) then goto XIT;          Parallel lines

    mp.x ← (p3.x + p4.x)/2;                      Find midpoint of p3 p4
    mp.y ← (p3.y + p4.y)/2;
    d1 ← LineToPoint(a1,b1,c1,mp);               Find D = distance p1 p2

                                                 to midpoint p3 p4
```

```
    if d1 = 0 then goto XIT;

    mp.x ← (p1.x + p2.x)/2;                    Find midpoint p1 p2
    mp.y ← (p1.y + p2.y)/2;
    d2 ← LineToPoint(a2, b2, c2, mp);          Repeat for second line.
    if d2 = 0 then goto XIT;

    rr ← r;
    if d1 ≤ 0 then rr ← −rr;
    c1p ← c1-rr* √a1² + b1² ;                  construct line ∏ to l at d.
    rr ← r; if d1 < = 0 then rr ← −rr;

    c2p ← c2 − rr* √a2² + b2² ;

    d ← al*b2 − a2*b1;                         Intersect constructed lines to
    pc.x ← (c2p*b1 − c1p*b2)/d;                find center of circular arc.
    pc.y ← (c1p*a2 − c2p*a1)/d;
    PointPerp(xa, ya, a1, b1, c1, pc);         Clip l1 at (xa, ya) if needed.
    PointPerp(xb, yb, a2, b2, c2, pc);         Clip l2 at (xb, yb) if needed.
    p2.x ← xa; p2.y ← ya; p3.x ← xb; p3.y ← yb;

    v1.x ← xa − pc.x; v1.y ← ya − pc.y;        Find angle wrt. x-axis
                                               from arc center, (xc, yc)
    v2.x ← xb − pc.x; v2.y ← yb − pc.y;

    pa ← arctan2(v1.y, v1.x);
    aa ← dot2(v1, v2);                         Find angle arc subtends.
    if cross2(v1, v2) < 0 then aa ← −aa;       Direction to draw arc.
XIT:
endproc Fillet.

begin                                          Main program
    MoveTo(p1.x, p1.y);
    LineTo(p2.x, p2.y);
    DrawArc(xc, yc, r, sa, a);
    MoveTo(p3.x, p3.y);
    LineTo(p4.x, p4.y);
end.
```

See also G1, 107

IV.6

FASTER LINE SEGMENT INTERSECTION

Franklin Antonio
QUALCOMM, Incorporated
Del Mar, California

Problem

Given two line segments in 2-D space, *rapidly* determine whether they intersect or not, and determine the point of intersection.

Rapid calculation of line segment intersections is important because this function is often a primitive called many thousands of times in the inner loops of other algorithms. An algorithm is presented here that uses approximately half as many operations as the intersection algorithm presented in Graphic Gems II (Prasad, 1991) and has tested faster on a variety of computers.

Algorithm

To develop the algorithm, it is convenient to use vector representation. Consider line segment $L12$ defined by two endpoints $P1$ and $P2$, and $L34$ defined by endpoints $P3$ and $P4$, as shown in Fig. 1.

Represent each point as a 2-D vector, i.e., $\mathbf{P}1 = (x1, y1)$, etc. Then a point $\mathbf{P}$ anywhere on the line $L12$ can be represented parametrically by a linear combination of $\mathbf{P}1$ and $\mathbf{P}2$ as follows, where α is in the interval $[0, 1]$ when representing a point on the line *segment L12*:

$$\mathbf{P} = \alpha\mathbf{P}1 + (1 - \alpha)\mathbf{P}2. \tag{1}$$

This can be rewritten in a more convenient form:

$$\mathbf{P} = \mathbf{P}1 + \alpha(\mathbf{P}2 - \mathbf{P}1) \tag{2}$$
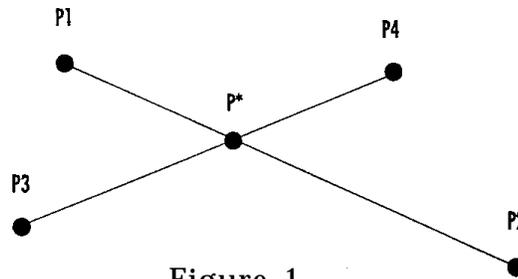
Figure 1.

In particular, we can locate the intersection point P^* by computing α and β by solution of the following linear system of equations. If the resulting *a* and *b* are in the interval [0, 1], then the line segments *L12* and *L34* intersect.

$$\mathbf{P}^* = \mathbf{P}1 + \alpha(\mathbf{P}2 - \mathbf{P}1), \tag{3a}$$

$$\mathbf{P}^* = \mathbf{P}3 + \beta(\mathbf{P}4 - \mathbf{P}3), \tag{3b}$$

Subtracting these equations yields

$$0 = (\mathbf{P}1 - \mathbf{P}3) + \alpha(\mathbf{P}2 - \mathbf{P}1) + \beta(\mathbf{P}3 - \mathbf{P}4) \tag{4}$$

Giving names to some intermediate values makes the equations easier to read:

$$\mathbf{A} = \mathbf{P}2 - \mathbf{P}1$$

$$\mathbf{B} = \mathbf{P}3 - \mathbf{P}4$$

$$\mathbf{C} = \mathbf{P}1 - \mathbf{P}3 \tag{5}$$

The solution of (4) for α and β is now

$$\alpha = \frac{ByCx - BxCy}{AyBx - AxBy}, \tag{6a}$$

$$\beta = \frac{AxCy - AyCx}{AyBx - AxBy}. \tag{6b}$$

Noting that the denominators of these expressions are the same, computing (5) and (6) requires nine adds and six multiplies in the worst case. (Prasad, 1991, required 14 adds and 12 multiplies in the worst case.) We avoid the division operation because we don't need α and β explicitly; we only need test them against the range [0, 1]. This is a little tricky, because the denominator may be either positive or negative. The division-avoiding test works like this:

> **if** denominator $>$ 0
> > **then if** numerator $<$ 0 **or** numerator $>$ denominator
> > > **then** *segments do not intersect*
> > **else if** numerator $>$ 0 **or** numerator $<$ denominator
> > > **then** *segments do not intersect*

Also note that (6a) can be computed and tested prior to computing (6b). If α is outside of [0, 1], there is no need to compute (6b). Finally, note that the case where denominator $=$ 0 corresponds to collinear line segments.

Timing Measurements/Further Optimizations

The algorithm above was timed against Prasad (1991), using random data, on a variety of computers (both RISCs and CISCs). It was found to execute 40% faster than Prasad's intersection test.

Because some tests can be performed on the numerator of (6a) and (6b) after knowing only the *sign* of the denominator, but without knowledge of its value, it is tempting to use a form of the fast-sign-of-cross-product algorithm (Ritter, 1991) on the denominator, after which the numerator can be tested against zero, eliminating some cases prior to *any* multiplications. In practice, this was found to slow the algorithm, as in this arrangement of the intersection algorithm the tests in question did not eliminate a large enough fraction of the test cases to pay back for the execution time of the many sign tests in Ritter's algorithm.

An additional speed increase *was* gained by testing to see if the bounding boxes of the two line segments intersect before testing whether the line segments intersect. This decreases the number of arithmetic operations when the boxes do not intersect, but involves an additional

overhead of several comparisons in every case. Therefore, there is a risk that this might actually slow down the average performance of the algorithm. However, the bounding box test was found to generate an additional 20% speed improvement, which was surprisingly consistent across different types of computer. Therefore, the C implementation includes a bounding box test.

Implementation Notes

The C implementation includes calculation of the intersection point coordinates in the case where the segments are found to intersect. This is accomplished via Eq. (3a).

The C implementation follows the same calling conventions as Prasad (1991) and produces identical results.

The C implementation uses integer arithmetic. Therefore, as Prasad warns, care should be taken to avoid overflow. Calculation of the intersection point involves operations that are cubic in the input coordinates, so limitation to input coordinates in the range [0, 1023], or other similar-sized range, will avoid overflow on 32-bit computers. When line segments do *not* intersect, input coordinates in the range [0,16383] can be handled.

See also G2, 7; G3, D.4.

SOLVING THE PROBLEM
OF APOLLONIUS AND
OTHER RELATED PROBLEMS

Constantin A. Sevici
GTS
Chelmsford, Massachusetts

The classical problem of Apollonius is to find a circle that is tangent to three given circles. In *Graphics Gems* (Rokne, 1991) a solution to this problem is given using bilinear transformations of the complex plane. Here we give a computationally simpler solution to two generalizations of this problem.

1.   Find the circle that intersects three other circles at given angles.

2.   Find the circle with a given radius that intersects two other circles at given angles.

Given two circles with r, r_i as radii, d the distance between the centers and θ_i their intersection angle, then the law of cosines in Fig. 1 gives

$$d^2 = r^2 + r_i^2 - 2rr_i\cos(\theta_i).$$

To solve the two general Apollonius problems, we will transform the preceding relation into a nearly linear equation between the coefficients of analytic equation of the two circles. Consider the analytic equation

$$\alpha(x^2 + y^2) + \beta x + \gamma y + \delta = 0$$

If $\alpha = 0$, this equation represents a line.

Figure 1.

If $\alpha \neq 0$, we can rewrite it as

$$\left(x + \frac{\beta}{2\alpha}\right)^2 + \left(y + \frac{\gamma}{2\alpha}\right)^2 = \frac{\beta^2 + \gamma^2 - 4\alpha\delta}{4\alpha^2}.$$

If $\beta^2 + \gamma^2 - 4\alpha\delta > 0$, the preceding equation represents a circle with

$$\text{center} = \left(-\frac{\beta}{2\alpha}, \frac{\gamma}{2\alpha}\right) \qquad \text{and} \qquad \text{radius} = \frac{\sqrt{\beta^2 + \gamma^2 - 4\alpha\delta}}{2\alpha}.$$

If $\beta^2 + \gamma^2 - 4\alpha\delta = 0$, the equation represents the point

$$\left(-\frac{\beta}{2\alpha}, -\frac{\gamma}{2\alpha}\right).$$

Now assume that the equations of the two circles in Fig. 1 are

$$\alpha(x^2 + y^2) + \beta x + \gamma y + \delta = 0,$$

$$\alpha_i(x^2 + y^2) + b_i x + c_i y + d_i = 0,$$

and let us introduce the notation

$$e_i = \sqrt{b_i^2 + c_i^2 - 4a_id_i}, \qquad \varepsilon = \sqrt{\beta^2 + \gamma^2 - 4\alpha\delta}.$$

Substituting the coordinates of the centers in the formula for the distance between points and using the values of radii given earlier, we get

$$\left(\frac{b_i}{2a_i} - \frac{\beta}{2\alpha}\right)^2 + \left(\frac{c_i}{2a_i} + \frac{\gamma}{2\alpha}\right)^2 = \frac{e_i^2}{4a_i^2} + \frac{\varepsilon^2}{4\alpha^2} - 2\frac{e_i\varepsilon}{4a_i\alpha}\cos(\theta_i) \ .$$

Algebraic simplification of the preceding equation gives

$$2d_i\alpha - b_i\beta - c_i\gamma + 2a_i\delta + e_i\cos(\theta_i)\varepsilon = 0.$$

Although this equation was derived assuming two circles, one can easily see that it remains valid for lines and points. Henceforth, when we refer to circles we also include the degenerate cases of points and lines.

For the first Apollonius problem, we write the preceding equation for $i = 0, 1, 2$ and recall the definition of ε, and we get

$$2d_0\alpha - b_0\beta - c_0\gamma + 2a_0\delta + e_0\cos(\theta_0)\varepsilon = 0,$$

$$2d_1\alpha - b_1\beta - c_1\gamma + 2a_1\delta + e_1\cos(\theta_1)\varepsilon = 0,$$

$$2d_2\alpha - b_2\beta - c_2\gamma + 2a_2\delta + e_2\cos(\theta_2)\varepsilon = 0,$$

$$\varepsilon^2 = \beta^2 + \gamma^2 - 4\alpha\delta.$$

To get the third equation for the second Apollonius problem, recall that

$$r = \frac{\sqrt{\beta^2 + \gamma^2 - 4\alpha\delta}}{2\alpha} = \frac{\varepsilon}{2\alpha} \ .$$

Thus, we can write the third equation as

$$2\alpha r - \varepsilon = 0.$$

If $r > 1$ (for example, for a line $r = \infty$), then if we define $k = 1/r$, we can replace the third equation with

$$2\alpha - k\varepsilon = 0.$$

We have reduced both problems to the solution of the nearly linear system of equations

$$m_{00}\alpha + m_{01}\beta + m_{02}\gamma + m_{03}\delta + m_{04}\varepsilon = 0,$$

$$m_{10}\alpha + m_{11}\beta + m_{12}\gamma + m_{13}\delta + m_{14}\varepsilon = 0,$$

$$m_{20}\alpha + m_{21}\beta + m_{22}\gamma + m_{23}\delta + m_{24}\varepsilon = 0,$$

$$\varepsilon^2 = \beta^2 + \gamma^2 - 4\alpha\delta.$$

To solve this nonlinear system of equations, first we find the general solution of the linear subsystem, and then, using that, we can solve the nonlinear equation.

For circles in general position, the rank of the preceding linear subsystem is 3, and from linear algebra we get the solution of the linear subsystem in the form

$$\alpha = f_{00}u + f_{01}v,$$

$$\beta = f_{10}u + f_{11}v,$$

$$\gamma = f_{20}u + f_{21}v,$$

$$\delta = f_{30}u + f_{31}v,$$

$$\varepsilon = f_{40}u + f_{41}v.$$

where u and v are arbitrary real numbers.

Substituting in $\varepsilon^2 = \beta^2 + \gamma^2 - 4\alpha\delta$, we get

$$(f_{40}u + f_{41}v)^2 - (f_{10}u + f_{11}v) - (f_{20}u + f_{21}v)^2$$

$$+ 4(f_{00}u + f_{01}v)(f_{30}u + f_{31}v) = 0.$$

Expanding and rearranging terms the preceding equation can be written as

$$A_{00}u^2 + A_{01}uv + A_{11}v^2 = 0.$$

Since only the relative ratio of u and v is relevant, one can set either to 1 and solve for the other value. We can avoid handling too many special cases by setting $u = \cos(\phi)$, $v = \sin(\phi)$, and solving the resulting trigonometric equation. The classical Apollonius problem is a special case of the first problem with $\theta_i = 0$, π for $i = 0$, 1, 2. In general, we get eight solutions for the first problem and four solutions for the second problem.

When the rank of the linear subsystem is 2, the circles are in a special position (for example, two of them are coincident). From linear algebra, we get

$$\alpha = f_{00}u + f_{01}v + f_{02}w,$$

$$\beta = f_{10}u + f_{11}v + f_{12}w, \; ,$$

$$\gamma = f_{20}u + f_{21}v + f_{22}w,$$

$$\delta = f_{30}u + f_{31}v + f_{32}w,$$

$$\varepsilon = f_{40}u + f_{41}v + f_{42}w,$$

where u, v, and w are arbitrary real numbers.

Substituting in $\varepsilon^2 = \beta^2 + \gamma^2 - 4\alpha\delta$, we get

$$(f_{40}u + f_{41}v + f_{42}w)^2 - (f_{10}u + f_{11}v + f_{12}w)^2$$

$$- (f_{20}u + f_{21}v + f_{22}w)^2$$

$$+ 4(f_{00}u + f_{01}v + f_{02}w)(f_{30}u + f_{31}v + f_{32}w) = 0$$

Expanding and rearranging terms, we can write the preceding equation as

$$A_{00}u^2 + A_{11}v^2 + A_{22}w^2 + A_{01}uv + A_{02}uw + A_{12}vw = 0,$$

which is the homogeneous equation of a plane conic. It is known that if a conic has a real point, then it has infinitely many, and all these points can be rationally parameterized with a degree 2 function in two parameters s and t. This parametrization can be obtained by stereographic projection of the conic from the known point to any line not passing through this point. Since α, β, γ, δ, ε depend linearly on u, v, w, it follows that α, β, γ, δ, ε depend quadratically on s and t. Thus, we can find a representation of the form

$$\alpha = g_{00}s^2 + g_{01}st + g_{02}t^2,$$

$$\beta = g_{10}s^2 + g_{11}st + g_{12}t^2,$$

$$\gamma = g_{20}s^2 + g_{21}st + g_{22}t^2,$$

$$\delta = g_{30}s^2 + g_{31}st + g_{32}t^2$$

$$\varepsilon = g_{40}s^2 + g_{41}st + g_{42}t^2.$$

If we recall that

$$\text{center} = \left(-\frac{\beta}{2\alpha}, -\frac{\gamma}{2\alpha}\right) \quad \text{and} \quad \text{radius} = \frac{\varepsilon}{2\alpha},$$

we get the following result.

In the case of rank 2, there are infinitely many solutions. The coordinates of the center and the value of the radius are rational homogeneous functions of degree 2 in two parameters.

Since a rational curve of degree two is conic, we get as a special case that the center of all circles tangent to two circles describes a conic.

The case when the linear subsystem is of rank 1 is handled in a similar manner.

As a final note, the technique presented above can be used to solve analogous problems in higher dimensions. For example, one can find the spheres that intersect four other spheres at given angles. In particular, one can find the sphere tangent to three spheres and orthogonal to a plane. This can be used to find the circle tangent to three given circles that are coplanar without having to transform the coordinates.

A C implementation of the algorithm is straightforward; one needs a routine that solves a linear homogeneous system.

See also G2,19

V
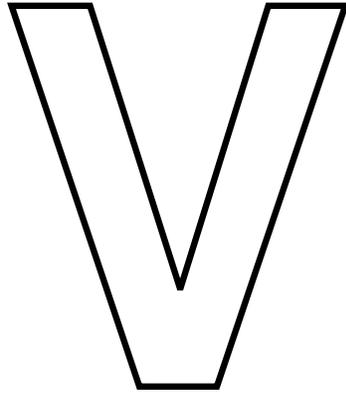
3-D GEOMETRY
AND ALGORITHMS

V
3-D GEOMETRY AND ALGORITHMS

The heart and soul of realistic computer graphics simulation is the manipulation of three-dimensional geometry. Although rendering provides the "skin" of computer graphics, the meat and bones are provided by the geometry. This section contains a variety of tools for manipulating three-dimensional geometry, ranging from clever tricks to more substantial algorithms.

The first Gem improves on a few previous Gems which discussed the manipulation of triangles. The third Gem provides a simple improvement to a well-known technique for calculating distances. The fifth Gem outlines Newell's method for computing the plane equation of a polygon. The method is particularly interesting in the way that it handles nonplanar polygons.

Several of the Gems provide nice building blocks for popular algorithms. The second Gem describes how to subdivide a convex polygon based on its intersection with a plane. This tool could be used as part of a binary space partitioning algorithm, which is described in the next section. The fourth Gem describes how to combine polygons that are very nearly planar, which often are produced accidentally when subdividing planar polygons, due to roundoff errors. The sixth and seventh Gems discuss efficient recipes for calculating intersections between planes, and between triangles and cubes. The triangle-cube intersection test is very useful for spatial subdivision algorithms such as octree encoding. The eighth Gem provides an efficient hierarchical technique for overlap testing, which is an important first step in intersection testing.

The ninth and tenth Gems are related in that they both discuss the manipulation of simplexes and related figures in three and more dimensions. The eleventh Gem discusses how to convert Bézier triangles into a more tractable representation, rectangular patches. The last Gem in this section discusses how to sample a three-dimensional curve for a two-dimensional drawing.

V.1

TRIANGLES REVISITED

Fernando J. López-López
Southwestern College
Chula Vista, California

In the book *Graphics Gems* I, in the chapter entitled "Triangles" (Goldman, 1990), the author gives closed-form expressions for important points and parameters in a triangle, such as the perimeter, the area, the center of gravity, the orthocenter, and the centers and radii of the circumcircle and the incircle.

These formulae can be written as vector expressions in a more succinct and useful form, more adaptable to programming. In what follows I have departed somewhat from the nomenclature used by Goldman: I denote vectors with an arrow, and scalars without the arrow. I have also added the positions and radii of the Feuerbach or nine-point circle and the excircles.

Let the triangle be given by the position vectors $\vec{z}1$, $\vec{z}2$, $\vec{z}3$.

Define the sides of the triangle as vectors $\vec{a}$, $\vec{b}$, $\vec{c}$, such that

$$\vec{a} + \vec{b} + \vec{c} = 0.$$

Namely,

$$\vec{a} = \vec{z}2 - \vec{z}3,$$

$$\vec{b} = \vec{z}3 - \vec{z}1,$$

$$\vec{c} = \vec{z}1 - \vec{z}2.$$

We now compute their magnitudes *a, b, c*, and the quantities

$$d1 = -\vec{b} \cdot \vec{c} \rightarrow \cos A = d1 \, / \, bc,$$

$$d2 = -\vec{c} \cdot \vec{a} \rightarrow \cos B = d2 \, / \, ca,$$

$$d3 = -\vec{a} \cdot \vec{b} \rightarrow \cos C = d3 \, / \, ab,$$

where · represents the dot product.

  We have also obtained the angles of the triangle, with the usual convention that angle *A* is opposite to side *a*, etc.

  We also need

$$e1 \;=\; d2 \, d3,$$

$$e2 \;=\; d3 \, d1$$

$$e3 \;=\; d1 \, d2$$

$$e \;=\; e1 \;+\; e2 \;+\; e3.$$

We now define

$$2s \;=\; a \;+\; b \;+\; c,$$

which is the perimeter of the triangle.

  Also,

$$2\Delta \;=\; |\vec{z}1 \times \vec{z}2 \;+\; \vec{z}2 \times \vec{z}3 \;+\; \vec{z}3 \times \vec{z}1| \rightarrow \text{area} = \Delta,$$

where × represents the cross product, vertical bars denote the magnitude of the vector, and Δ is the area of the triangle.

  From these definitions and results we can successively compute the following: The position of the center of gravity, i.e., the concurrence of the medians, as,

$$\vec{z}G \;=\; (\vec{z}1 \;+\; \vec{z}2 \;+\; \vec{z}3) \, / \, 3.$$

The position of the orthocenter, i.e., the concurrence of the altitudes, as,

$$\vec{z}H = \frac{e1\vec{z}1 + e2\vec{z}2 + e3\vec{z}3}{e}.$$

The position of the circumcenter, i.e., the concurrence of the perpendicular bisectors to the sides, and the radius of the circumcircle, i.e., the circle that passes through the three vertices, are given by

$$\vec{z}C = \frac{3\vec{z}G - \vec{z}H}{2}, \qquad RC = \frac{1}{2}\sqrt{\frac{a^2 + b^2 + c^2}{e}}.$$

The position of the incenter, i.e., the concurrence of the angle bisectors, and the radius of the incircle, i.e., the circle inscribed in the triangle, tangent to the three sides, are

$$\vec{z}I = \frac{a\vec{z}1 + b\vec{z}2 + c\vec{z}3}{2s}, \qquad rI = \frac{\Delta}{s}.$$

The positions of the excenters, and the radii of the excircles, i.e., the circles tangent to one side of the triangle and to the extensions of the other two sides, are given by

$$\vec{z}Ia = \frac{s\vec{z}I - a\vec{z}1}{s-a}, \qquad rIa = \frac{\Delta}{s-a};$$

$$\vec{z}Ib = \frac{s\vec{z}I - b\vec{z}2}{s-b}, \qquad rIb = \frac{\Delta}{s-b};$$

$$\vec{z}Ic = \frac{s\vec{z}I - c\vec{z}3}{s-c}, \qquad rIc = \frac{\Delta}{s-c};$$

And the center and radius of the Feuerbach circle (nine-point circle), i.e., the circle that passes through the three feet of the altitudes and the middle points of the sides, are

$$\vec{z}F = (\vec{z}C + \vec{z}H)/2, \qquad RF = RC/2 .$$

The Feuerbach circle is tangent to the incircle and to the three excircles. For the properties of these points and circles, see, for example, Coxeter (1969), pp. 10–20.

See also G1, 20; G1, 297.

V.2
PARTITIONING A 3-D CONVEX POLYGON WITH AN ARBITRARY PLANE

Norman Chin
Columbia University
New York, New York

Introduction

Partitioning 3-D polygons is required by various graphics algorithms as a preprocessing step, such as in the radiosity method (Cohen and Greenberg, 1985) and in the BSP tree visible-surface algorithm (Fuchs *et al.*, 1980).

Since the implementation of many applications is simplified by dealing with only convex polygons, this Gem describes how to partition planar convex polygons by an arbitrary plane in three-space. This is basically one pass of the Sutherland-Hodgman (Sutherland and Hodgman, 1974) polygon clipper, but optimized for clipping convex polygons.

The input polygon is represented as a linked list with both portions returned after partitioning. This differs from Heckbert's Gem (Heckbert, 1990) in three ways, in that his input polygon is stored as an array, with only the inside portion returned, after being clipped by an axis-aligned plane.

At most, two vertices need be inserted into both pieces after splitting. This is easy to code in array form but requires block copies of overlapping regions. In linked list form, manipulating a few pointers is tedious to code but is usually faster than block copies. This latter representation is very well suited for algorithms that require lots of splitting and that output variable-length polygonal fragments, such as in Thibault and Naylor (1987) and Chin and Feiner (1989).

Input

The inputs are the coefficients of the partitioning plane's plane equation and the polygon to be partitioned in linked list format. Collinear vertices are allowed. The last vertex must be a duplicate of the first vertex, since we are processing edges. (That is, a square has five vertices instead of four.) Other than this last vertex, coincident vertices are not allowed.

Output

The input polygon is divided into its negative and positive fragments, if any, with respect to the partitioning plane. Either fragment may still contain collinear vertices. Newly generated intersection vertices coincident with existing vertices are deleted. The last vertex is, as always, a duplicate of the first. Note that if the output polygon is embedded in the partitioning plane, there will be no negative and positive fragments.

Algorithm Overview

There are four basic steps to the algorithm:

- vertex classification to see which half-space they lie in with respect to the partitioning plane
- intersection detection
- intersection computation
- polygon splitting, including updating pointers

Procedure

By plugging the coordinates of a vertex into the partitioning plane's plane equation in the form $Ax + By + Cz + D$ = signed distance from plane, a vertex can be classified as to whether it's on the negative or positive side or embedded in the plane. All that is needed is the sign. Because of round-off errors, the plane must be assigned a small "thickness" to determine if the vertex is on the plane.

Table 1

	$c(V1)$	$c(V2)$
1	0	0
2	–	–
3	+	+
4	–	+
5	+	–
6	0	–
7	0	+
8	+	0
9	–	0

Only edges with ends having opposite signs are guaranteed to intersect the plane. Let $V1$ be the vertex at the start of an edge and let $V2$ be the vertex at the end of the edge. When these two vertices are classified, $c(V1)$ and $c(V2)$, there are nine possibilities, as shown in Table I.

In cases 1, 2, and 3, the edge is embedded in the plane, on the negative, and on the positive side of the plane, respectively, so there is no intersection.

In cases 4 and 5, the edge lies on both sides, so there is an intersection.

In case 6, the edge is touching the plane at its start vertex, $V1$. In order for an intersection to exist, a transition from positive to zero (case 8c) must have occurred just previously. In fact, by checking for case 8c or 9b (described below and in Table II), case 6 is taken care of as a result. Likewise, case 7 can be ignored, since checking for case 8b or 9c will handle it. Intersections through vertices are counted once, not twice.

In case 8 and 9, the edge is touching the plane at its ending vertex, $V2$. To determine if there is an intersection, we need to know the classification of the next vertex, $V3$. Recall that coincident vertices are not allowed, so $V2$ is not equal to $V3$. This results in the following cases, as shown in Table II.

In cases 8a and 9a, $V3$ lies on the plane. It appears that we may have to continually lookahead until we find a nonzero sign to determine if the polygon eventually crosses the plane. However, vertex sequences, "+00–" and "–00+" cannot occur since we are dealing with convex polygons, so there is no intersection.

In cases 8b and 9b, both $V1$ and $V3$ are on one side of the plane so there is no intersection.

Table 2

	$c(V1)$	$c(V2)$	$c(V3)$
8a	+	0	0
8b	+	0	+
8c	+	0	–
9a	–	0	0
9b	–	0	–
9c	–	0	+

In cases 8c and 9c, there is a sign change resulting in an intersection at $V2$.

Computing the actual intersection coordinate of an edge and a plane is very similiar to computing the intersection of a ray and a plane and is discussed in other texts (Glassner *et al.,* 1989, and Foley *et al.,* 1990).

After both intersections vertices are found, the vertex lists are adjusted in both split polygons and returned to the caller.

Depending upon the application, when the resulting split polygons are too small to be further processed, it may be best to ignore them. A good heuristic to determine "smallness" is to check their areas.

Enhancements

Suggested enhancements include maintaining a winged-edge data structure (Baumgart, 1975) after partitioning, storing each unique vertex once instead of multiple times, and flagging an edge if it was introduced via partitioning.

Acknowledgments

See also G3, E.6.

V.3

SIGNED DISTANCE FROM POINT TO PLANE

Príamos Georgiades
CrystalGraphics, Inc.
Santa Clara, California

The standard way of obtaining the shortest (Euclidean) distance from a point P to a plane **J** is finding the orthogonal projection Q of P onto **J**, so that the distance is the length of the vector $P - Q$. This involves computing a square root (Glassner, 1990). Furthermore, if the application requires ordering a set of points with respect to their distance from **J**, it is necessary to know on which side of the plane the points lie. This would require another dot product operation. This gem yields the signed distance from P to **J** with a single dot product and an addition. It assumes that the plane equation is represented by its unit normal, $\mathbf{J}_N$, and a scalar $\mathbf{J}_d$ such that for any point R on the plane, $J_N \cdot R + J_d = 0$. In most applications, it will become necessary to normalize the plane normal at one instance or another, so it is expected that all plane normals are thus maintained. It is a one-time square-root and division operation that will eliminate the need to evaluate a square root each time a distance from this plane is sought.

The intuition behind this is that since the line connecting P to its projection Q on the plane is collinear to J_N, its length can be measured as a scalar multiple of J_N. Assuming that the length of J_N is known to be 1, that scalar value is the wanted distance from the plane.

The derivation has as follows: Let $d = |P - Q|$, and express P using the parametric equation of the line through Q normal to **J**. Let $N = J_N$.

Then $P = dN + Q$, or

$$P_x = dN_x + Q_x,$$

$$P_y = dN_y + Q_y,$$

$$P_z = dN_z + Q_z.$$

Multiply each equation by the respective coordinate of N and add.

$$N_x P_x = dN_x^2 + N_x Q_x$$

$$N_y P_y = dN_y^2 + N_y Q_y$$

$$\frac{N_z P_z = dN_z^2 + N_z Q_z}{N \cdot P = dN \cdot N + N \cdot Q}$$

But $N \cdot N = 1$ by assumption, and $N \cdot Q = -J_d$ from the plane equation. Hence $d = N \cdot P + J_d$, or in Gems' notation

$$d \leftarrow J_N \cdot P + J_d$$

Note that, if d is positive, then P lies on the same side of **J** as its normal, if it's negative it lies on the back side, and if zero (or within some epsilon-small value) it lies on the plane. Moreover, the projection Q of P onto **J** can now be obtained from the parametric equation of the line with the computed d.

See also G1, 297.

V.4
GROUPING NEARLY COPLANAR POLYGONS INTO COPLANAR SETS

David Salesin and Filippo Tampieri
Cornell University
Ithaca, New York

Introduction

The propagation of errors in geometric algorithms is a very tricky problem: Small arithmetic errors in the computations may lead not only to errors in the *numerical* results, such as the coordinates of vertices and edges, but also to inconsistencies in the *combinatorial* results, such as the vertex and edge adjacencies in the topological description of an object. Inconsistencies can be a more serious problem than numerical errors, as they can lead to infinite loops or a program that "bombs out" on certain inputs.

For example, most polyhedral modelers that accommodate coplanar faces correctly handle polygons that are *really* coplanar or that meet at large angles, but may fail on polygons that meet at nonzero angles that are sufficiently small (Segal, 1990).

We present here a very simple, reliable technique for grouping polygons into coplanar sets, so that

- no two sets meet at an angle smaller than ϵ; and

- no polygon's normal or position is perturbed by more than a small, fixed δ (related to ϵ).

(The desired tolerance ϵ is arbitrary, and can be chosen according to the application.)

Applications

The ability to group nearly coplanar polygons into coplanar sets has a wide variety of applications in computer graphics:

Solid modeling. Overlapping, nearly coplanar polygons can lead to inconsistencies between the numerical and combinatorial data maintained by solid modelers based on boundary representations. For example, because of numerical inaccuracies the computed intersection of two overlapping faces may actually lie outside of each of these faces, leading to an inconsistency. By recognizing and explicitly merging nearly coplanar faces, a solid modeler can get around these problems.

BSP trees. A binary space partitioning tree is a partition of space used to impose an ordering over a set of polygons. The supporting plane *p* of one polygon is chosen as the root of the tree, and the other polygons are classified as lying either in front of *p*, behind *p*, or on *p*. If a polygon cannot be classified, it is split along *p*, and the resulting fragments are classified independently. Failing to classify nearly coplanar polygons in the same set results in a larger tree and may also cause many polygons to be unnecessarily split, thus adversely affecting performance. Applying our technique to BSP trees would benefit a wide range of applications, including visible surface determination (Fuchs *et al.,* 1980), shadow generation (Chin and Feiner, 1989), radiosity computations (Campbell and Fussel, 1990), and CSG operations (Naylor *et al*, 1990).

Lighting computations. Rendering a set of nearly coplanar polygons as independent surfaces may result in Mach bands or other annoying artifacts in the final image. The problem arises when numerical errors creep into the surface normal computations of polygons that are supposed to be coplanar. If nearly coplanar polygons are grouped back into coplanar sets, these artifacts can be avoided.

Radiosity. Current radiosity systems mesh every input polygon into a set of patches, solve for the radiosities at the vertices of these patches, and then display the results using linear interpolation. If the boundary vertices of the meshes of neighboring coplanar patches are not aligned, shading discontinuities and cracks between polygons may result (Baum *et al.,* 1991; Haines, 1991). Assembling coplanar polygons into surfaces and maintaining connectivity information across these polygons during meshing allows these problems to be avoided.

Texture mapping. Consider the problem of applying a 2-D texture map to a surface represented by a polygonal mesh. Simply texture-mapping each polygon separately would result in discontinuities of the texture across polygon boundaries. In the case of planar surfaces, this problem is easily solved by adopting a (*u, v*) reference system local to the surface and assigning (*u, v*) coordinates to every vertex in the polygonal mesh. If the input to the renderer is a simple stream of polygons, it is then necessary to first group these polygons into coplanar sets.

Derivation

We derive our technique from a more general method for sorting a collection of numbers approximately.

First, we need a precise way of describing a set of elements that is only "approximately" sorted:

Definition. Given a set of real numbers $S = \{s_1, s_2, \ldots, s_n\}$, we will say that S is *δ-sorted* if $s_i \leq s_j + \delta$ for all $1 \leq i < j \leq n$.

Suppose now that we are given a subroutine *Increasing*(a, b, ϵ) that is guaranteed to return TRUE if $a < b - \epsilon$ and FALSE if $b < a - \epsilon$, but may also return a third value UNKNOWN if $|a - b| \leq \epsilon$.

We would like to find a way of using $O(n \log n)$ calls to the *Increasing* routine in order to arrive at a δ-sorted list, where δ is related to ϵ in some reasonable way.

In order to accomplish this task, we introduce the following data structure, called a *representative tree*:

```
type   RepresentativeTree = record
          representative: Element
          bucket: set of Element
          left, right: pointer to RepresentativeTree
       end record
```

The idea of the tree is to keep a single "representative" element at each node, along with a "bucket" of elements that are within a given tolerance ϵ of the representative. We build the tree, one element at a time, using the following procedure:

```
procedure InsertElement takes
      tree: pointer to RepresentativeTree
      x: Element
      ε: Tolerance
   begin
      if tree = NIL then
          tree ← alloc RepresentativeTree[x, φ, NIL, NIL]
      else
          case Increasing(x,tree.representative, ε) of
             TRUE: InsertElement(tree.left, x, ε)
             FALSE: InsertElement(tree.right, x, ε)
             UNKNOWN: tree.bucket ← tree.bucket ∪ {x}
          end case
      end if
   end
```

The *InsertElement* routine works by comparing a new element x against the representative element y at the root of the tree. If $x < y - \epsilon$, we recurse on the left subtree; otherwise, if $y < x - \epsilon$, we recurse on the right subtree. Finally, if $|x - y| \le \epsilon$, then we merely insert x into a bucket associated with y, where it can take part in no further comparisons.

After all the points have been inserted in the tree, the approximately sorted list is produced by traversing the tree in order, and outputting each node's representative element, along with all the elements stored in its bucket, as the node is traversed.

It is not difficult to show that this algorithm uses $O(n \log n)$ comparisons to produce a list that is always 2ϵ-sorted (Salesin, 1991).

Comparing Two Planes

It remains only to define the comparison function *Increasing* for two planes. We will store each plane $ax + by + cz + d = 0$ in the following data structure:

> **type** *Plane* = **record**
>   *N: Vector*
>   d: **real**
> **end record**

where $N = (a, b, c)$ is the plane's unit normal, and d describes the distance between two parallel planes. Newell's method can be used to compute the normalized plane equations of the input polygons (Sutherland *et al.*, 1974; Tampieri, 1992).

For the *Increasing* function we use a sort of lexicographic comparison between the two planes, testing the angle between the planes first, and, if the angle is near zero, testing the distance between the planes second:

> **procedure** *Increasing* **takes**
>   *P, Q: Plane*
>   *ϵ.cosangle, ϵ.distance*: **real**
> **returns**
>
>   *cmp*: {TRUE, FALSE, UNKNOWN}
> **begin**
> $c \leftarrow P.N \cdot Q.N$
>   **if** $c < -\epsilon.cosangle$ **then**
>   **return** TRUE

```
        else if c > ε.cosangle then
            return FALSE
        else
            d ← P.d − Q.d
            if d < −ε.distance then
                return TRUE
            else if d > ε.distance then
                return FALSE
            else
                return UNKNOWN
            end if
        end if
    end
```

Here, two planes P and Q are considered to be nearly "coplanar" if the angle between their normals is no greater than $\cos^{-1}(\epsilon.cosangle)$, and if their relative distance is within $\epsilon.distance$.

See also G3, E.5.

V.5
NEWELL'S METHOD FOR COMPUTING THE PLANE EQUATION OF A POLYGON

Filippo Tampieri
Cornell University
Ithaca, New York

Here is a numerically robust way of computing the plane equation of an arbitrary 3-D polygon. This technique, first suggested by Newell (Sutherland *et al.*, 1974), works for concave polygons and polygons containing collinear vertices, as well as for nonplanar polygons, e.g., polygons resulting from perturbed vertex locations. In the last case, Newell's method computes a "best-fit" plane.

Newell's Method

It can be shown that the areas of the projections of a polygon onto the Cartesian planes, *xy*, *yz*, and *zx*, are proportional to the coefficients of the normal vector to the polygon. Newell's method computes each one of these projected areas as the sum of the "signed" areas of the trapezoidal regions enclosed between each polygon edge and its projection onto one of the Cartesian axes.

Let the *n* vertices of a polygon p be denoted by V_1, V_2, . . . , V_n, where $V_i = (x_i, Y_i, z_i)$, $i = 1, 2, . . . , n$. The plane equation $ax + by + cz + d = 0$ can be expressed as

$$(X - P) \cdot N = 0, \tag{1}$$

where $X = (x, y, z)$, $N = (a, b, c)$ is the normal to the plane, and P is an arbitrary reference point on the plane. The coefficients a, b, and c are

given by

$$a = \sum_{i=1}^{n}(y_i - y_{i\oplus1})(z_i + z_{i\oplus1}),$$

$$b = \sum_{i=1}^{n}(z_i - z_{i\oplus1})(x_i + x_{i\oplus1}),$$

$$c = \sum_{i=1}^{n}(x_i - x_{i\oplus1})(y_i + y_{i\oplus1}),$$

where $\oplus$ represents addition modulo n.

The coefficient d is computed from Eq. (1) as

$$d = -P \cdot N, \tag{2}$$

where P is the arithmetic average of all the vertices of the polygon:

$$P = \frac{1}{n}\sum_{i=1}^{n}V_i. \tag{3}$$

It is often useful to "normalize" the plane equation so that $N = (a, b, c)$ is a unit vector. This is done simply by dividing each coefficient of the plane equation by $(a^2 + b^2 + c^2)^{1/2}$.

Newell's method may seem inefficient for planar polygons, since it uses all the vertices of a polygon when, in fact, only three points are needed to define a plane. It should be noted, though, that for arbitrary planar polygons, these three points must be chosen very carefully:

1. Three points uniquely define a plane if and only if they are not collinear; and
2. if the three points are chosen around a "concave" corner, the normal of the resulting plane will point in the direction opposite to the expected one.

Checking for these properties would reduce the efficiency of the three-point method as well as making its coding rather inelegant. A good strategy may be that of using the three-point method for polygons that are already known to be planar and strictly convex (no collinear vertices,) and using Newell's method for the rest.

See also G3, E, 4.

V.6

PLANE-TO-PLANE INTERSECTION

Príamos Georgiades
CrystalGraphics, Inc.
Santa Clara, California

This is an algorithm for computing the parametric equation of the line of intersection between two planes. It's a basic, useful and efficient function with broad applications in 3-D graphics. Let planes **I** and **J** have normals $M = I_N$ and $N = J_N$, respectively, such that for any point Q on either plane, $I_N \cdot Q + I_d = 0$ and $J_N \cdot Q + J_d = 0$. The task at hand is to compute a vector L and a point P such that for any $Q \in \mathbf{I} \cap \mathbf{J}$, $Q = tL + P$.

The direction vector L of the line is found easily as the unit cross-product of the normals of the two planes. This follows from the observation that it must be perpendicular to both normals M and N, and that the vector $M \times N$ does exactly that ($M \times N \cdot N = M \times N \cdot M = 0$). Mathematically, since the solution space of the two equations is one-dimensional, it must be that L is collinear to $M \times N$. If M and N are linearly dependent, then the planes do not intersect.

Picking the point P on the line is just as easy. Since there are only two constraints P must satisfy (the two plane equations), a third constraint must be added. The simplest thing to do is set one of its coordinates to zero (so that it lies on a coordinate plane) and solve the two equations for the remaining two coordinates. In order to avoid numerical error in the derivation that follows, choose P on the coordinate plane to which L is nearest to normal. In other words, set to zero the coordinate of P corresponding to the coordinate of L of greatest magnitude.
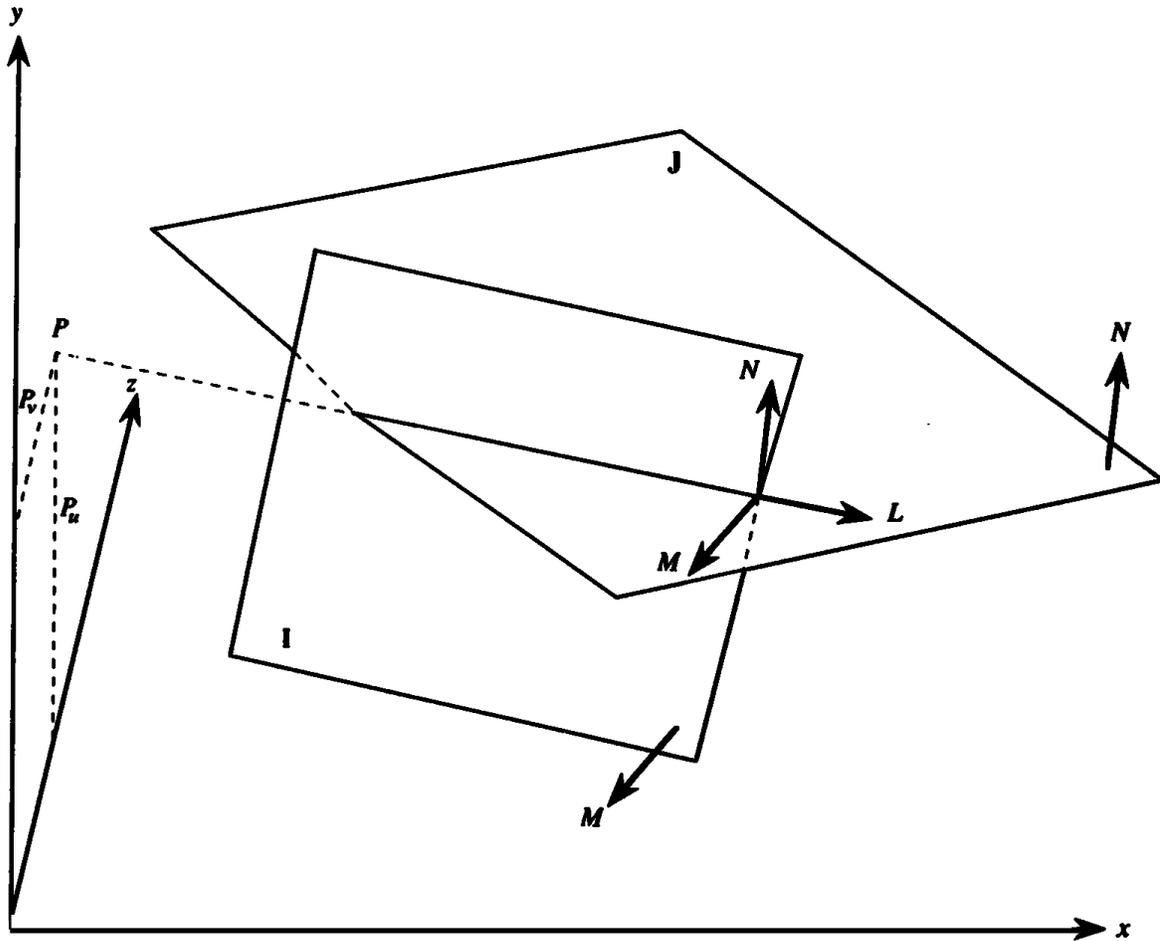
Figure 1. Plane-to-plane intersection.

Let w be the coordinate of maximal magnitude, and u and v the other two, such that $\{u, v, w\}$ is an ordered permutation of $\{x, y, z\}$. Then

$$
\begin{aligned}
M \cdot P + I_d &= 0 \\
N \cdot P + J_d &= 0 \\
P_w &= 0
\end{aligned}
\qquad \Rightarrow \qquad
\begin{aligned}
M_u P_u + M_v P_v &= -I_d \\
N_u P_u + N_v P_v &= -I_l
\end{aligned}
$$

$$
\Rightarrow \qquad
\begin{aligned}
P_u &= \frac{M_v J_d - N_v I_d}{M_u N_v + M_v N_u} \\[2mm]
P_v &= \frac{N_u I_d - M_u J_d}{M_u N_v \,\vdots\, M_v N_u}
\end{aligned}
$$

Note that $(M \times N)_w = M_u N_v + M_v N_u$. Since this is the denominator, it should not be near zero. This is why w is set to be the largest coordinate of L. Now the assignments can be made in this order:

$$L \leftarrow M \times N,$$

$$P_u \leftarrow \frac{M_v J_d - N_v I_d}{L_w},$$

$$P_v \leftarrow \frac{N_u I_d - M_u J_d}{L_w},$$

$$P_w \leftarrow 0,$$

$$L \leftarrow \frac{L}{|L|}.$$

See also G91, 305; G3, E.4.

V.7

TRIANGLE–CUBE INTERSECTION

Douglas Voorhies
Silicon Graphics Inc.
Mountain View, California

3-D triangles must occasionally be tested for intersection with axis-aligned cubes, most commonly when the cubes subdivide space in some classification scheme such as octrees. The conventional approach, successive polygon clipping by six cube-face half-spaces, is computationally expensive since it determines the actual polygonal intersection. This is not usually required when triangles are only being organized into cubic regions; in such cases, all that is needed is a yes-or-no detection of whether they touch.

This gem describes an algorithm that exploits simple tests to give quick answers much of the time and to minimize the work done in harder cases. There are three parts to the algorithm. First, we use three quick trivial-accept, trivial-reject tests to eliminate easy cases. Second, we detect triangle edges penetrating any face of the cube, and third, we detect cube corners poking through the interior of the triangle.

Testing the triangle vertices against the face-planes (the common bounding-box test) is done first. Finding any vertex inside the unit cube permits a trivial acceptance. On the other hand, if all vertices are outside the same face-plane, then we have a trivial rejection. A second simple test compares the vertices against the 12 planes that touch the 12 cube edges and are at 45° to their edge's adjacent faces. Although the enclosed volume is a rhombic dodecahedron, the comparisons are easy. Every pairing of coordinate components are simply added or subtracted and then compared with 1.0, for example $-X - Z > 1.0$. And a third test compares the triangle vertices against the eight planes that pass through one of the eight cube corners and are perpendicular to the corresponding cube diagonal, enclosing an octahedron. Here again, the math is simple,

such as $X - Y + Z > 1.5$. If all three vertices are outside of the same edge or corner plane, the triangle can be trivially rejected. Although the third test rejects triangles only rarely, those it does catch are nasty cases that would have otherwise required the full algorithm to assess.

If a triangle has survived these trivial accept/reject tests, we can handle the two major interesting cases. Testing triangle sides to see if they penetrate the cube consists of finding the intersection of a line segment and a plane, followed by detecting whether that intersection point lies within a cube face. The planes are, of course, the six face-planes of the cube, and by using the results of the initial trivial accept/reject tests, only those triangle edges that span particular face-planes need be investigated. First, a parametric line segment equation $\alpha P_1 + (1 - \alpha)P_2$ is solved for a particular face plane (X, Y, or Z being equal to ± 0.5 or -0.5), and then the resulting point is compared with ± 0.5 for the other two coordinates. A minimum of four and a maximum of 12 tests will be needed.

If no triangle edge penetrates the cube, the triangle and cube may still intersect if one or more cube corners poke through the interior of the triangle. To test for this, we find the intersection of the four cube diagonals and the plane of the triangle. If any of the intersection points lie inside the triangle, then the triangle intersects the cube. The plane of a triangle can be described as $AX + BY + CZ + D = 0$, where *A, B, C is* the plane normal vector (which can be obtained by the cross product of any two triangle side vectors). Cube diagonal lines can be described by $X = \pm Y = \pm Z$. Solving the plane equation for the $X = Y = Z$ diagonal, for example, gives $-D/(A + B + C)$ as a metric of the distance from cube center down the diagonal to the plane, where values $-0.5 \ldots +0.5$ imply the unit cube interior. If the intersection point is not in this range, then this diagonal does not need to be pursued further.

If the cube diagonal does intersect the plane of the triangle at a point inside the cube, then the cube/triangle problem reduces to a coplanar-point/triangle intersection test. First, a triangle bounding box test is used to trivially reject points far from the triangle. Then we use cross products. As you walk around the edges of a triangle clockwise, only the interior points are to the right of your path at all times. Similarly, a counterclockwise walk keeps the interior to your left. The cross product of two vectors is a third vector whose component signs reveal whether one vector is to the right or left of the other. Thus, by taking a cross product at each
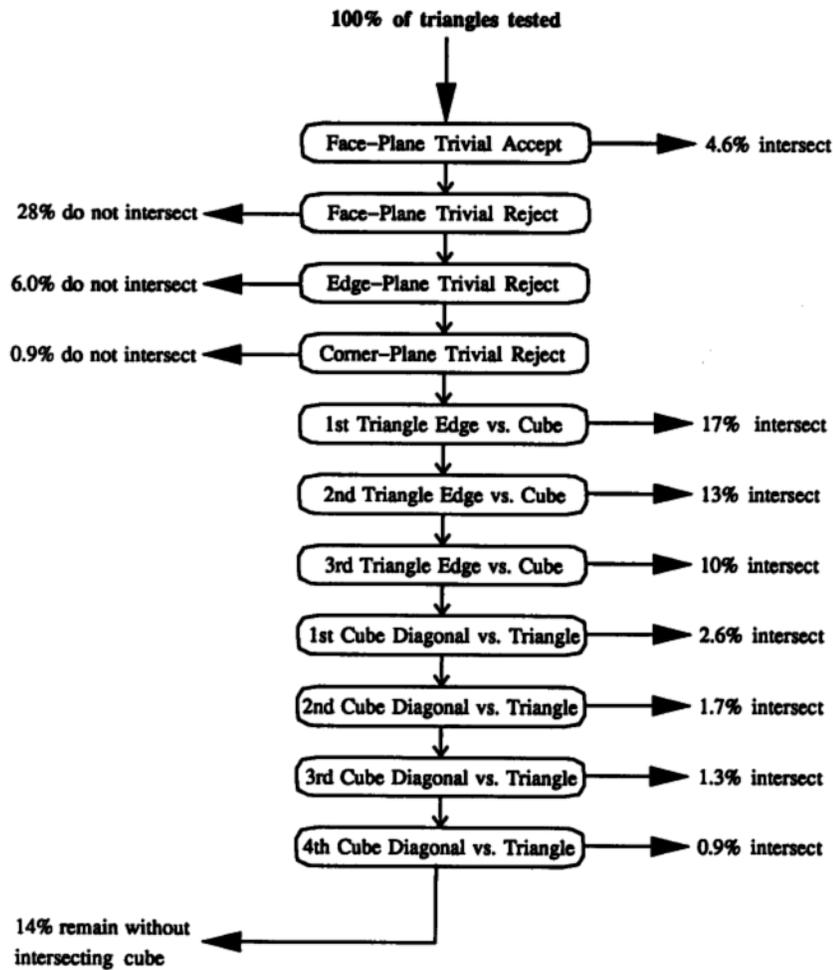
Figure 1. Experiences of random triangles within a + 2.0 volume potentially intersecting a centered axis-aligned unit cube.

vertex, we can ascertain from its component signs whether the intersection point lies to the left or right of each edge (from that vertex to the next). Only intersection points inside the triangle will have cross-product component signs that imply either left/left/left or right/right/right. (The vertex sequence is arbitrary; it is lucky that we need not discern clockwise from counterclockwise, since only interior points are on the *same* side of all three edges.)

The effectiveness of these individual tests varies with the relative size of the cube and the triangle. Small cubes that intersect large triangles usually touch the triangle's interior. Conversely, large cubes allow easy trivial acceptance of most intersecting small triangles. If they are of roughly similar size, then all stages of the algorithm participate. For example, if the triangles have vertices chosen at random within a cubic volume linearly four times larger than the cube, roughly half can be shown to intersect a unit cube placed at the volume's center. Using this test case reveals aspects of the algorithm's behavior. Figure 1 shows the percentage of triangles handled by each stage of the algorithm in this particular example.

See also G2, 219.

FAST *n*-DIMENSIONAL EXTENT OVERLAP TESTING

Len Wanger and Mike Fusco
SimGraphics Engineering
South Pasadena, Califonia

The detection of overlapping extents is a common technique used in operations such as clipping, hidden surface removal, and collision detection. Efficient determination of overlapping extents allows rejection of objects that do not overlap without resorting to computationally expensive intersection testing.

A naive $O(n^2)$ implementation of extent testing checks each extent against all others. By sorting the extents on each dimension, the runtime characteristics can be reduced to $O(n \log n)$; however, if care is not taken, a large amount of time is spent on linear searches among the list of extents. With a little ingenuity these linear searches can be eliminated. This gem presents a C++ class for performing efficient detection of overlapping extents.

Using the Class

The ExtHit class has four external methods:

ExtHit (size)—This method creates an instance of the class. The size argument specifies the maximum number of extents that can be tested for overlap with this instance of the class.

~ ExtHit( )—This method destroys the instance of the class.

Boolean add(extent, obj)—Adds an extent to be tested for extent overlaps. The obj argument specifies the object that the extent is associated with and is passed to the user-supplied overlap function when the object is involved in an extent overlap.

test(func, data)—This method performs the actual overlap checking. The func argument specifies the user-supplied function to be called for each pair of overlapping objects; the data argument specifies user-supplied data to be passed as an argument to func.

Using the ExtHit class is simple. After creating an instance of the class, add all of the extents to be tested by using the add method. Once all of the extents have been added testing is performed with a single call to the test method. The test method will call the user supplied overlap function for every pair of extents that overlap.

Implementation

Three accelerations are used to speed up the extent overlap checking:

1. Overlap checking is performed on one dimension at a time, and only those extents that overlapped in all of the previous dimensions are tested for any dimension. An overlap table is kept to register which extents overlapped in the previous dimensions.

2. The extent values for each dimension are sorted when testing for overlap. Both the minimum and maximum extent values are placed on the overlap list, which is then sorted. Two extents overlap in a dimension if the minimum extent value of one extent is between the minimum and maximum extent values of the other extent in the overlap list. This property is used to find all of the extent intersections in the overlap list. During traversal of the overlap list, any extent whose minimum extent value has been passed, and whose maximum value has not, is said to be open and is placed on the open list. When the minimum extent value for an extent is passed, the entries in the overlap table for the intersections of the extent and every extent currently on the open list are marked. The extent is removed from the open list when the maximum extent value is passed.

3. Only those extents that were involved in overlaps in the previous dimensions are tested for any dimension. Extents that are involved in an overlap are said to be active and are kept on the active list. Only

placing active extents on the overlap list has the effect of reducing the number of extents tested for any dimension.

Pseudo-code

The actual extent overlap testing is applied in the test method. The pseudo-code for the test method is as follows:

```
function test
  begin
    for dimension ← 0 to num_dimensions-1 do
      begin
        for each active extent
          begin
            add the minimum and maximum extent values to the overlap
            list
          endloop;

        sort the overlap list by extent value
        call subtest
      endloop;
    end;

function subtest
  begin
    for each extent value in the overlap list
      begin
        rec ← the extent associated with the extent value

        if rec is not on the open list then
          for each extent on the open list
            begin
              if rec overlapped with the extent from the open list in the
              previous dimension then
                if this is the highest dimension being tested then
                  begin this means the extents overlap
                    call the user supplied overlap function
                  end
```

```
            else
              begin
                Set the overlap table entry for rec and the extent from
                the open list
                Mark both rec and the extent from the open list as
                active
              end
          add rec to the open list
      endloop
      else
        begin
          remove rec from the open list
          if rec is not marked as active then
            begin
              remove rec from the active list
            end
        end
    endloop;
  end;
```

See also G1, 395.

SUBDIVIDING SIMPLICES

Doug Moore
Rice University
Houston, Texas

Problem

The most popular primitive shapes in graphics and geometric computation are the *boxes* (segment, square, cube, tesseract, . . .) and the *simplices* (segment, triangle, tetrahedron, 5-cell, . . .). Boxes fit together nicely in grids, which makes referring to a box by a set of coordinates easy. Boxes can be subdivided into smaller boxes, and quadtrees built from hierarchies of large and small boxes. On the other hand, simplices are ideal for rendering and shading algorithms, like Gouraud shading, based on linear interpolation. However, the way simplices fit together and break apart is not so obvious, so they are used less frequently than boxes in many geometric applications.

For example, it is not well known that simplices can be divided into subsimplices just as boxes can be divided into subboxes. This Gem presents two methods for dividing an n-simplex into 2^n simplices and discusses some applications of the subdivision of simplices.

Recursively Subdividing Simplices

The generalization of a triangle or tetrahedron to any dimension is a *simplex,* defined as the convex hull of an affinely independent point set. The structure of a simplex is such that any pair of its vertices form an edge, any three of its vertices form a face, any four of its vertices form a cell, and so on. A simplex is the simplest kind of polytope.

An n-simplex with vertices $\{v_0, \ . \ . \ ., \ v_n\}$ can be subdivided into 2^n smaller n-simplices by a simple recursive procedure. A hyperplane cut-

ting through the midpoint of edge $\overline{v_0 v_n}$ and through all the other vertices divides the simplex into two simplices. Split these new simplices by cutting through the midpoint of edge $\overline{v_0 v_{n-1}}$ and the midpoint of edge $\overline{v_1 v_n}$; that cuts one edge on each of the smaller simplices. Proceed at step k by cutting edge $\overline{v_i v_{i+(n-k)+1}}$ for all i, and each step cuts one edge per simplex and doubles the number of simplices. The result of the nth step is the *recursive* decomposition of the simplex. Figure 1 illustrates this process for a tetrahedron.

If, for all i, v_i is the point whose first i coordinates are ones and whose other coordinates are zeros, the simplex is called a *Kuhn simplex,* the natural generalization of a right isosceles triangle to higher dimensions. Generally, the subsimplices that result from recursive subdivision are of several different shapes, but a Kuhn simplex is recursively subdivided only into smaller Kuhn simplices.
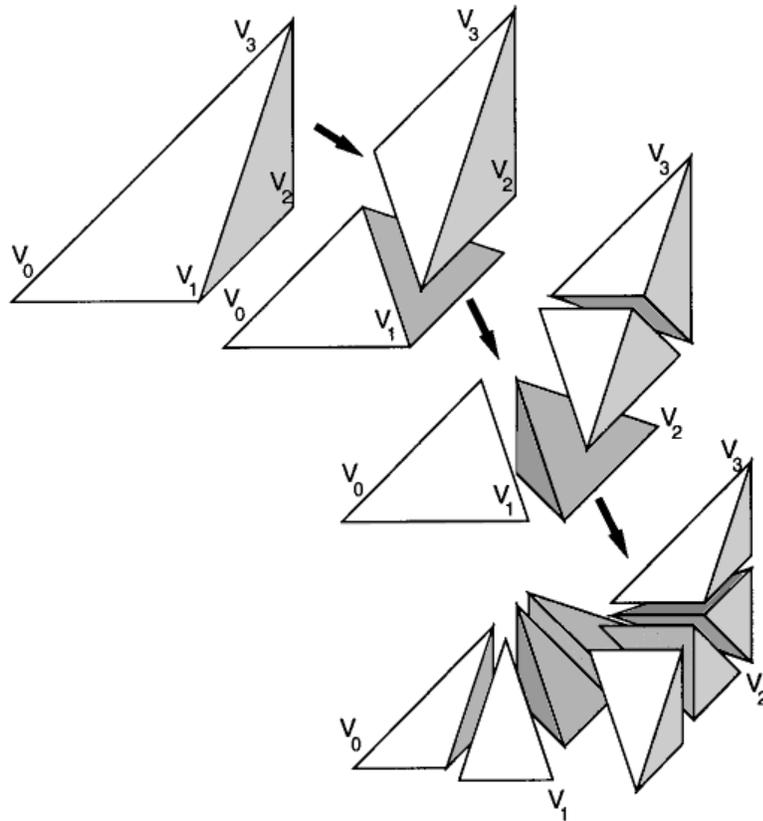
Figure 1. Phases of recursive subdivision.
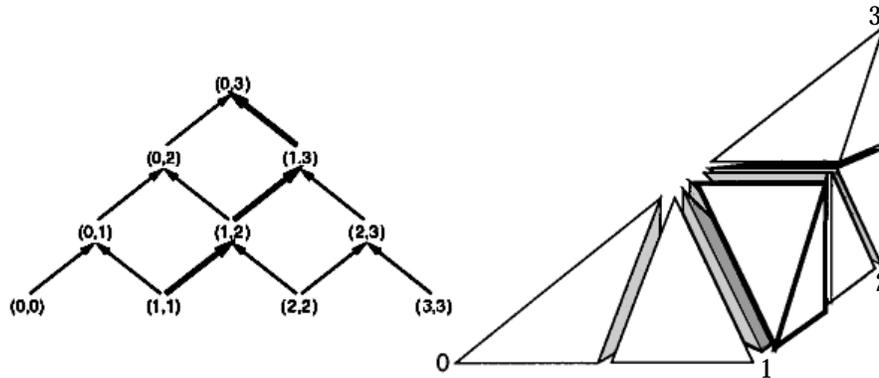
Figure 2. Recursive subdivision.

The treelike diagram of Fig. 2 provides another way of viewing the recursive decomposition of a tetrahedron. Let the pair (i, i) represent vertex v_i of the tetrahedron and let (i, j) represent the midpoint of edge $\overline{v_i v_j}$ of the original tetrahedron. With that understanding, each path from the bottom of the diagram to the top represents a set of vertices that define a simplex of the recursive subdivision. To generate just the kth simplex of the decomposition, start at the node labeled $(b(k), b(k))$, where $b(k)$ denotes the number of 1 bits in the binary representation of k. Starting with the ones digit, traverse a path from the bottom of the diagram to the top, turning right with every 0 bit and left with every 1 bit. The vertices encountered in that traversal are the vertices of the kth subsimplex. The diagram highlights the traversal for the fourth subsimplex of the recursive subdivision, and that subsimplex has a bold outline in the accompanying illustration.

Symmetrically Subdividing Simplices

When applied to a triangle, recursive subdivision does not yield the usual "corner-chopped" decomposition of the triangle into four subtriangles. A subdivision technique that does have that property for general simplices is the *symmetric* subdivision method. The symmetric subdivision method can best be understood by considering the Kuhn triangulation of boxes. A unit n-box can be divided into several n-simplices by considering every possible ordering of the coordinate axes. For each permutation p of the

integers 1 to *n,* there is a region of the box for which $0 \leq x_{p(1)} < x_{p(2)} \leq \ldots < x_{p(n)} < 1$. Altogether, there are *n*! of these regions, each traced out by a set of edges on the boundary of the box that begins at the origin, then moves in the $x_{p(1)}$ direction, then in the $x_{p(2)}$ direction, and so on, ending at the point with all 1 coordinates. Each of these paths includes $n + 1$ vertices that define a Kuhn simplex, and the set of all those simplices defines the Kuhn triangulation of the box.

The Kuhn triangulation of a box is not unique; it depends on the choice of a diagonal of the box that is an edge of every Kuhn simplex of the triangulation. By triangulating a box, and at the same time triangulating the 2^n half-size subboxes of the box along diagonals parallel to the original, two sets of Kuhn simplices are created. Each large Kuhn simplex contains exactly 2^n of the smaller Kuhn simplices. Figure 3 illustrates this set of decompositions for cubes and tetrahedra.
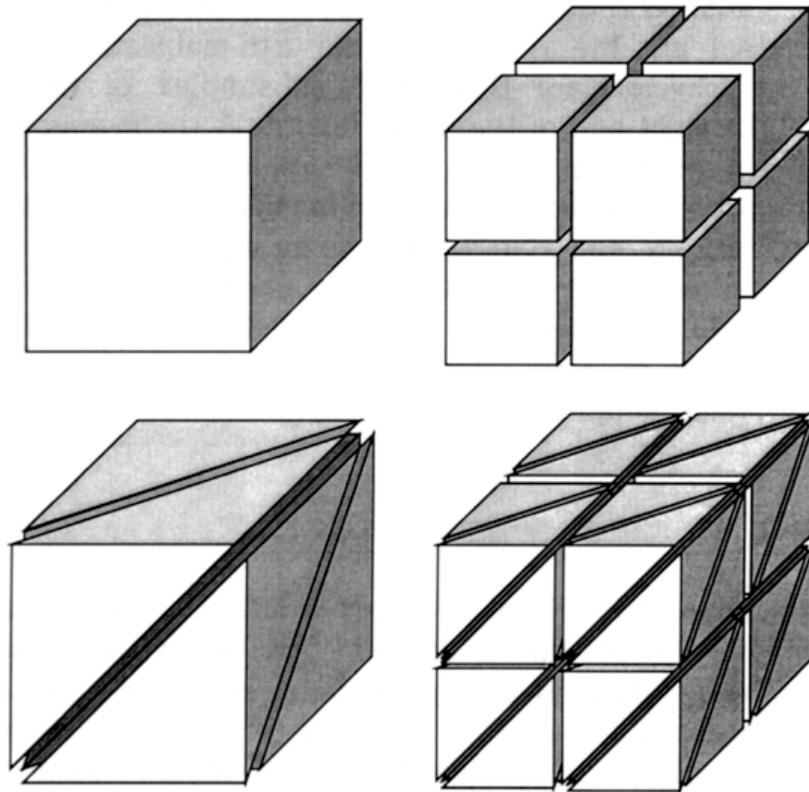
Figure 3. Symmetric subdivision from box subdivision.
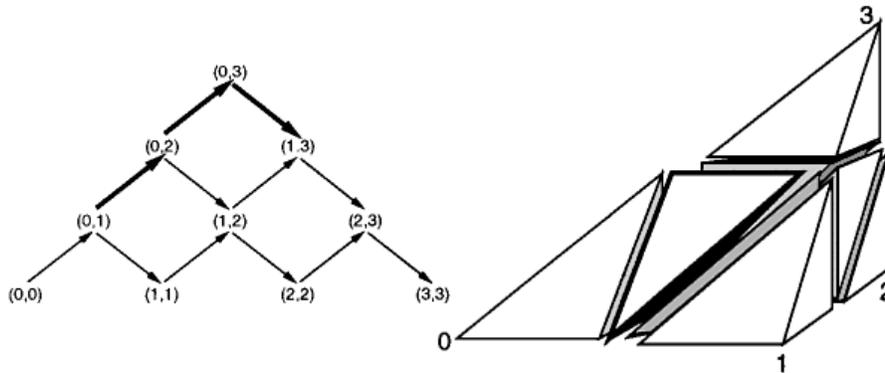
Figure 4. Symmetric subdivision.

Symmetric subdivision can also be understood by the diagram in the left half of Fig. 4. To generate the kth subsimplex of the symmetric subdivision, start at the node labeled $(0, b(k))$, and starting with the ones digit, traverse a path from the left side of the diagram to the right, turning up with every 0 bit and down with every 1 bit. The vertices encountered in that traversal are the vertices of the kth subsimplex. The figure highlights the traversal for the fourth subsimplex of the symmetric subdivision. The right half of the figure illustrates the symmetric subdivision of a tetrahedron, with the fourth subsimplex outlined in bold.

Although most simplices, when subdivided symmetrically, yield subsimplices of several different shapes, there is an infinite family of simplices for which the subsimplices have the same shape as the parent simplex. These are the Kuhn n-simplices compressed or elongated along the direction from v_0 to v_n. Compressing by a factor of $1/\sqrt{1 + n}$ yields the most nearly regular simplices in this family.

Applications

A subdivision scheme for simplices leads to alternate, simplicial forms of all the familiar concepts that arise from box subdivision. For example, simplicial quadtrees have the same advantages as the more usual box-based quadtrees, but also greater flexibility. A tree of tetrahedra can be refined by selecting a leaf of the tree and subdividing it, where any points along the edges, not just midpoints of edges, are used to divide the leaf. As a result, simplicial quadtrees can produce object descriptions without
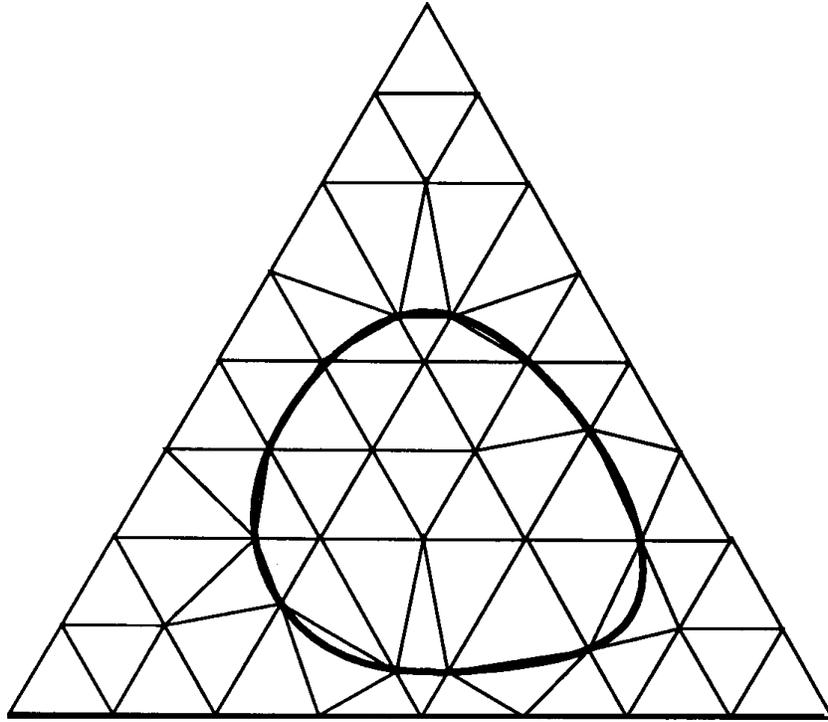
Figure 5. Simplicial decomposition of closed curve.

the jagged boundaries that characterize objects described with usual quadtrees. In three dimensions and higher, the same flexibility is possible in box-based trees only if the boxes are permitted to warp to produce curved faces. For example, Fig. 5 illustrates how a uniform simplicial quadtree can nearly approximate a closed curve.

Simplicial quadtrees are convenient tools in systems that manipulate multivariate Bernstein polynomials of fixed degree. Such polynomials are naturally defined by a set of coefficients over a domain simplex. Polygonalizers for algebraic surfaces and systems for modeling with trivariate freeform deformations are more naturally based around simplicial quadtrees than box-based trees.

See also G3, E.10.

UNDERSTANDING SIMPLOIDS

Doug Moore
Rice University
Houston, Texas

Problem

The set of shapes called *simploids* includes both the simplices (triangle, tetrahedra, . . .) and the boxes (square, cube, . . .) as special cases. Simploids are surprisingly easy to compute with, because they have a simple structure. This Gem describes simploids and several ways to break them apart. In particular, we present algorithms for splitting a simploid into a collection of simplices and splitting a simplex into a pair of simploids on opposite sides of a plane.

Simploids

The *product* of two polytopes P_1 and P_2 is the polytope obtained by replacing each vertex v of P_1 by a copy of P_2, denoted $c(v)$, and each edge (v, w) of P_1 by the set of edges between corresponding vertices in $c(v)$ and $c(w)$. The polytopes P_1 and P_2 are called *factors* of the product. The dimension of the product of polytopes is the sum of the dimensions of the factors. For example, the product of two one-dimensional segments is a two-dimensional rectangle. The product of a two-dimensional polygon with a one-dimensional segment is a three-dimensional right polygonal prism. Figure 1 depicts the product of two triangles, a four-dimensional figure.

Expressed as coordinates, the vertices of the product are formed by concatenating coordinates of the factors. For an n_1 dimensional polytope
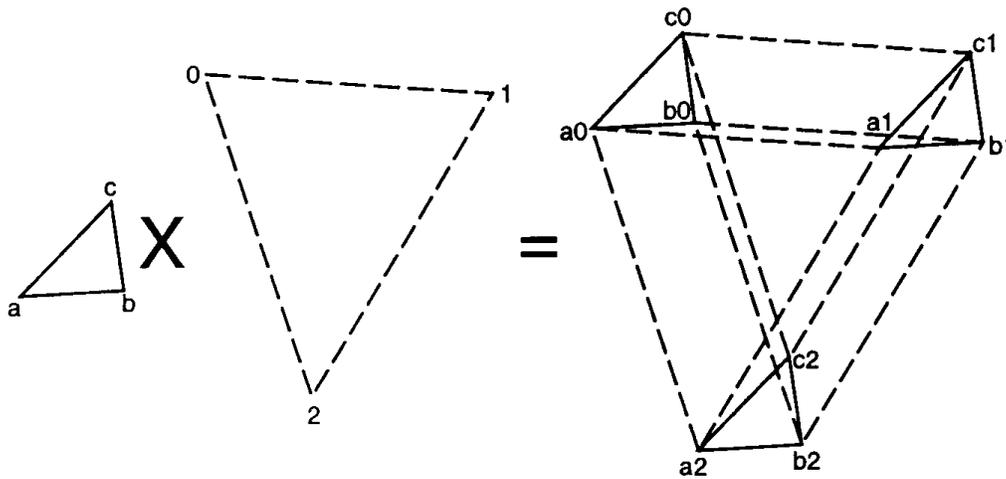
Figure 1. The product of two triangles.

with m_1 vertices $\{v_{11}, v_{12}, \ldots, v_1 m_1\}$ and an n_2 dimensional polytope with m_2 vertices $\{v_{21}, v_{22}, \ldots, v_2 m_2\}$, their product is the $n_1 + n_2$ dimensional polytope with $m_1 m_2$ vertices $(v_{1i} v_{2i})$, where coordinates of vertices of the factors are concatenated to form vertices of their product. A pair of vertices $(v_{1i_1} v_{2i_2})$ and $(v_{1j_1} v_{2j_2})$ in the product is connected by an edge if either $i_1 = j_1$, or $i_2 = j_2$.

A polytope in n dimensions can be the product of up to n lower-dimensional polytopes. For example, a cube is the product of three intervals. In a general polytope that is the product of $r \leq n$ polytopes in which the kth factor polytope has m_k vertices, there are $m_1 m_2 \ldots m_r$ vertices. Vertices $v_{1i_1} v_{2i_2} \ldots v_{ri}$, and $v_{1j_1} v_{2j_2} \ldots v_{rj_r}$, are adjacent in the product exactly when $i_k = j_k$ for all but one value of k.

A *simploid is* a polytope isomorphic to a product of simplices. That is, the vertices of a product of simplices can be moved around a bit, and the result is still a simploid if all the faces remain flat and the incidence of vertices, edges, and faces does not change. In three dimensions, there are three kinds of simploids, the (3)-simploids (tetrahedra), the (2,1)-simploids (triangular prisms), and the (1,1,1) simploids (parallelepipeds). Although a simploid is not necessarily a product, the structure is the same. Thus, algorithms for manipulating products of simplices, which

often work by operating separately on the separate factors, are easily converted to algorithms for working on general simploids.

Because simploids are products of simplices, they can be subdivided into smaller simploids. For example, if each of the three interval factors of a cube is subdivided, the cube is subdivided into eight subcubes. In general, subdividing each of the factors of a simploid produces 2^n smaller simploids of the same kind, where n is the dimension of the simploid.

Simploids to Simplices

There is a well-known method for dividing boxes into simplices called Kuhn's triangulation (Kuhn, 1960), described in the Gem "Subdividing Simplices." This method was generalized by W. Dahmen and C. Micchelli to divide arbitrary simploids into simplices (Dahmen and Micchelli, 1982). Order the vertices in each of the factor simplices of the simploid, so that each vertex but one in the simplex has a uniquely specified successor. A vertex w of the simploid is a successor of vertex v if it is adjacent to v and in the single factor where the two differ, w is a successor of v. Under this definition, the simploid vertex that is the product of all first simplex vertices is the successor of no vertex; we call it the initial vertex of the simploid. The simploid vertex that is the product of all last simplex
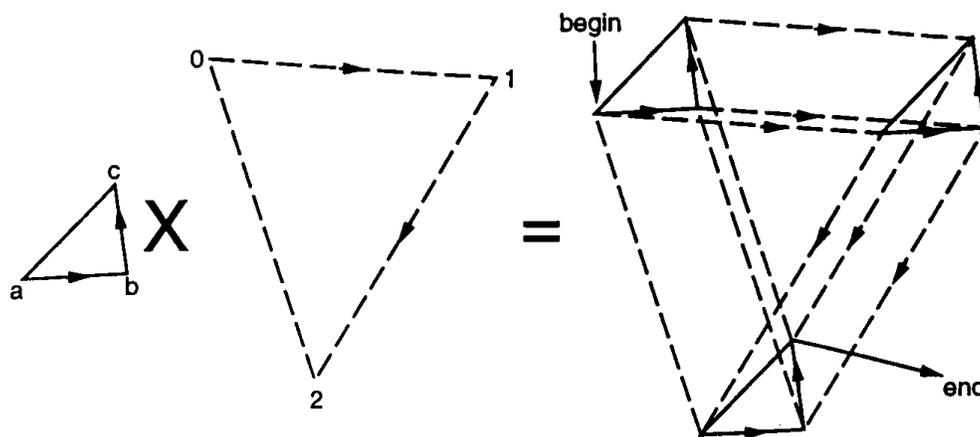
Figure 2. Edge orientation in a product of triangles.
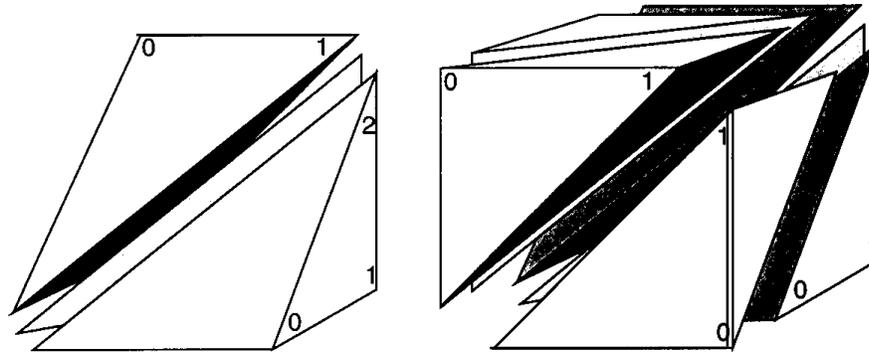
Figure 3. Decomposing a simploid into simplices.

vertices has no successors, and is called the terminal vertex of the simploid.

Each path from the initial vertex to the terminal vertex of the simploid that passes from a vertex to one of its successors at every step along the path traces out the vertices of a simplex, and those simplices taken together fill the volume of the simploid. Figure 2 illustrates the orientation of paths in the product of triangles. Figure 3 illustrates triangulations of a cube and a triangular prism, where the numeric labels in the figure represent orderings of the factor simplices.

Splitting Simplices into Simploids

A fundamental operation in constructive solid geometry is computing the intersection of two objects. When the objects are polytopes, the problem can be broken down further into many applications of a simplex/half-space intersection algorithm. Fortunately, such intersections have a structure that is easy to exploit algorithmically.

When a hyperplane cuts an n-simplex, it separates the $n + 1$ vertices of the simplex into two sets, with s vertices of the simplex on one side of the plane and $n + 1 - s$ on the other. The intersection of the simplex with a plane is an $(s - 1, n - s)$-simploid. The volume of the simplex is divided into an $(s, n - s)$-simploid that contains the intersection and the s vertices on one side of the plane, and an $(s - 1, n + 1 - s)$-simploid
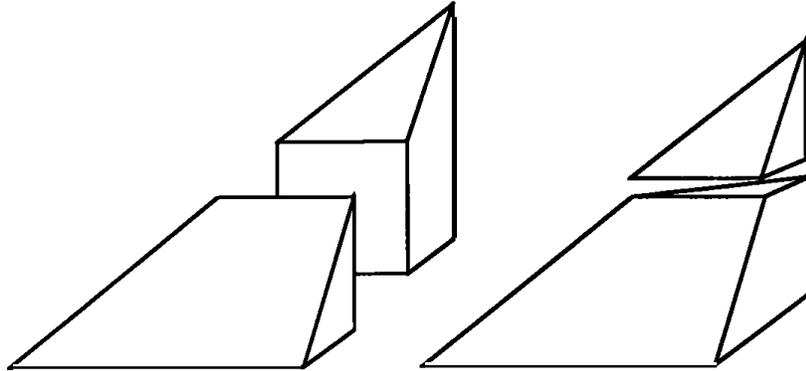
Figure 4. A tetrahedron splits into simploids two ways.

that contains the intersection and the other vertices of the simplex. Figure 4 illustrates the two interesting ways in which a tetrahedron can be cut by a plane into simploids.

In an application that works primarily with simplices, the simploid is a helpful intermediate step; the simploid can be decomposed into simplices immediately. On the other hand, an application that manipulates Simploids can cut a simploid by a plane by first decomposing the simploid into simplices, then cutting the simplices into simploids. In either case, the ability to understand the relationship between simplices and simploids makes more general object-intersection algorithms possible.

An example motivated my dissection of simploids. Suppose that a region of space is filled with several layers of materials of different densities. That is, there is a 12-inch layer of sand, below that five inches of lead, and below that 24 inches of concrete. Given a tetrahedron spanning several of the layers, how do you compute the weight of the material enclosed by the tetrahedron? Only by calculating the volume of the intersection of the tetrahedron with each layer.

The intersection of a tetrahedron with a layer can be complex. For example, if two vertices lie within the lead layer, and one each is above and below that layer, the intersection is a complex polyhedron with three-, four-, and five-sided pieces. It can be broken into five tetrahedra, but the technique is neither obvious nor easy to implement. However, that volume can be more easily computed as the volume of the entire tetrahe-

dron, reduced by the volumes of the two tetrahedra lying above and below the lead layer. By understanding that a single plane cuts the tetrahedron into simploids, all such volumes can be computed easily as sums and differences of simploid volumes, and simploid volumes are easily computed by summing the volumes of simplices.

See also G3, E.9.

V.11
CONVERTING BÉZIER TRIANGLES INTO RECTANGULAR PATCHES

Dani Lischinski
Cornell University
Ithaca, New York

Bézier triangles and rectangular Bézier patches are both commonly used for representing curved surfaces, but various software products often support only one or the other. This Gem describes a very simple way to convert triangular Bézier surface patches into rectangular ones. Two possible applications are

• ray-tracing triangular patches with a ray-tracer that supports only rectangular patches, and

• displaying triangular patches on graphics hardware that supports only rectangular patches.

We will describe how to convert quadratic and cubic triangles into biquadratic and bicubic rectangular patches, but the same principles can be used to derive a conversion for higher orders as well. We give only a terse definition for Bézier triangles and Bézier rectangular patches; an excellent comprehensive reference is Farin (1990).

The conversion of rectangular patches into triangles is not given here. It is a bit more involved, since, in general, one needs two triangles of higher degree in order to represent exactly a single rectangular patch.

Converting Quadratic Triangles

A Bézier triangle of degree n is essentially a mapping $T(u, v)$ from a unit triangle $((0, 0), (1, 0), (0, 1))$ into E^3. It is defined by a set of control

points $\mathbf{b}_{ijk} \in E^3$ as

$$T(u, v) = \sum \mathbf{b}_{ijk} \frac{n!}{i!\,j!\,k!} u^i v^j w^k,$$

where $w = 1 - u - v$, and the summation is over all i, j, $k \geq 0$ such that $i + j + k = n$. In the quadratic case this simplifies to

$$T(u, v) = \mathbf{b}_{002} w^2 + 2\mathbf{b}_{101} uw + \mathbf{b}_{200} u^2$$

$$+ 2\mathbf{b}_{011} vw + 2\mathbf{b}_{110} uv + \mathbf{b}_{020} v^2 \tag{1}$$

(see Fig. 1a).

A rectangular Bézier patch is a mapping $R(s, t)$ from a unit square $[0, 1] \times [0, 1]$ into E^3, defined by a set of control points $\mathbf{p}_{ij} \in E^3$ as

$$R(s, t) = \sum_{i=0}^{m} \sum_{j=0}^{n} \mathbf{p}_{ij} \frac{m!\,n!}{i!\,j!(m - i)!(n - j)!} s^i t^j (1 - s)^{m - i}(1 - t)^{n - j},$$

where m and n are the degrees of the patch in s and t, respectively. In the biquadratic case this simplifies to

$$R(s, t) = (\mathbf{p}_{00}(1 - s)^2 + 2\mathbf{p}_{01}s(1 - s) + \mathbf{p}_{02}s^2)(1 - t)^2$$

$$+ (\mathbf{p}_{10}(1 - s)^2 + 2\mathbf{p}_{11}s(1 - s) + \mathbf{p}_{12}s^2)2t(1 - t)$$

$$+ (\mathbf{p}_{20}(1 - s)^2 + 2\mathbf{p}_{21}s(1 - s) + \mathbf{p}_{22}s^2)t^2 \tag{2}$$

(see Fig. 1b).

Given the six control points $\mathbf{b}_{ijk}$ that define the quadratic triangle $T(u, v)$, we need to find a 3×3 array of control points $\mathbf{P}_{ij}$ that define the biquadratic rectangular patch $R(s, t)$, such that both define the same surface in E^3. Note that we have two different parametric domains here. To identify the two, we can "shrink" the top edge of the unit square onto the top vertex of the unit triangle by defining the following mapping
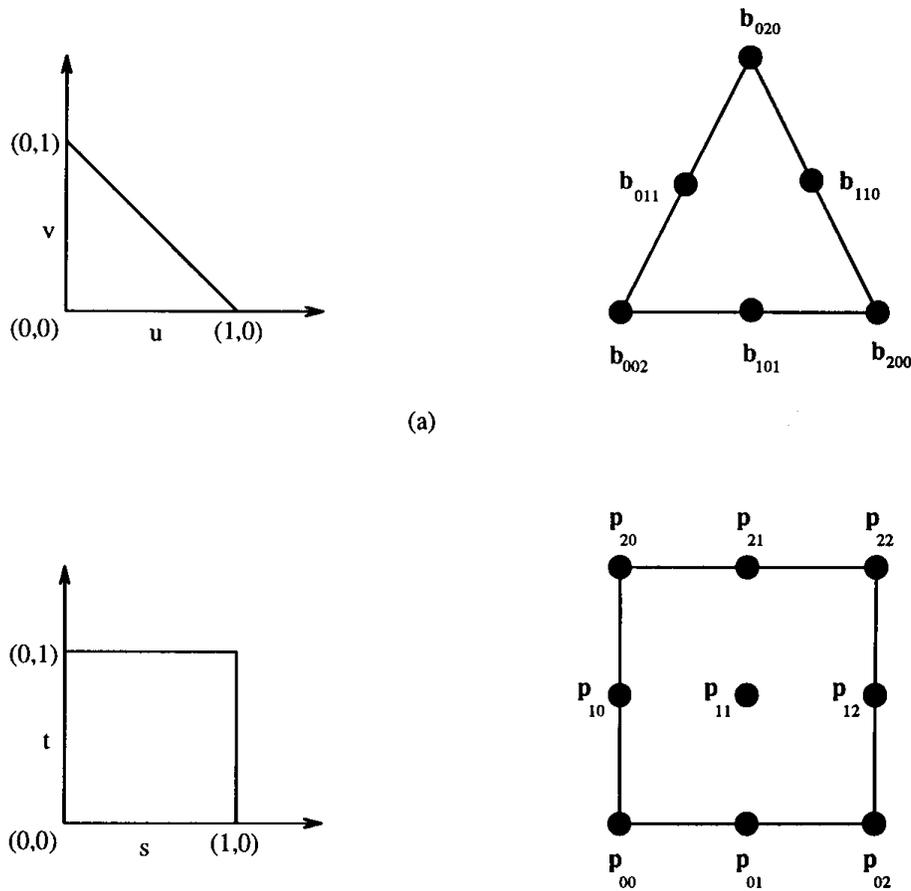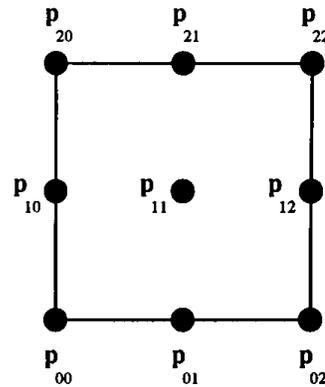
Figure 1. (a) The parametric domain and the six control points defining a quadratic Bézier triangle. (b) The parametric domain and the nine control points defining a biquadratic Bézier patch.

between them:

$$(u, \ v) \ = \ (s(1- \ t), \ t). \tag{3}$$

Thus, our goal is to find the points $\mathbf{p}_{ij}$ such that $R(s, \ t) \ = \ T(s(1 \ - \ t), \ t)$ for all $(s, \ t)$ in the unit square.

The points $\mathbf{b}_{002}$, $\mathbf{b}_{101}$, and $\mathbf{b}_{200}$ are the Bézier control points that define a quadratic boundary curve. Since $\mathbf{p}_{00}$, $\mathbf{p}_{01}$, and $\mathbf{p}_{02}$ must define the same curve, they should be equal to $\mathbf{b}_{002}$, $\mathbf{b}_{101}$, and $\mathbf{b}_{200}$ respectively. From the

same argument it follows that the other "boundary" points are also equal in both representations. Note that both $\mathbf{p}_{20}$ and $\mathbf{p}_{22}$ are equal to $\mathbf{b}_{020}$. By setting $\mathbf{p}_{21}$ to $\mathbf{b}_{020}$ as well, we will create a degenerate rectangular patch, which will have the required triangular shape. Thus, so far we have

$$\mathbf{p}_{00} \leftarrow \mathbf{b}_{002},$$

$$\mathbf{p}_{01} \leftarrow \mathbf{b}_{101},$$

$$\mathbf{p}_{02} \leftarrow \mathbf{b}_{200},$$

$$\mathbf{p}_{10} \leftarrow \mathbf{b}_{011},$$

$$\mathbf{p}_{12} \leftarrow \mathbf{b}_{110},$$

$$\mathbf{p}_{20},\ \mathbf{p}_{21},\ \mathbf{p}_{22} \leftarrow \mathbf{b}_{020}.$$

Now it only remains to obtain a value for the interior control point $\mathbf{p}_{11}$. Since by (3) the point $(s,\ t) = \left(\frac{1}{2},\ \frac{1}{2}\right)$ in the unit square corresponds to the point $(u,\ v) = \left(\frac{1}{4},\ \frac{1}{2}\right)$ in the unit triangle, we have the equation

$$T\left(\tfrac{1}{4},\ \tfrac{1}{2}\right) = R\left(\tfrac{1}{2},\ \tfrac{1}{2}\right).$$

Substituting in the patch definitions (1) and (2), we get

$$\frac{\mathbf{b}_{002}}{16} + \frac{\mathbf{b}_{101}}{8} + \frac{\mathbf{b}_{200}}{16} + \frac{\mathbf{b}_{011}}{4} + \frac{\mathbf{b}_{110}}{4} + \frac{\mathbf{b}_{020}}{4}$$

$$= \frac{\mathbf{p}_{00}}{16} + \frac{\mathbf{p}_{01}}{8} + \frac{\mathbf{p}_{02}}{16} + \frac{\mathbf{p}_{10}}{8} + \frac{\mathbf{p}_{11}}{4} + \frac{\mathbf{p}_{12}}{8} + \frac{\mathbf{p}_{20}}{16} + \frac{\mathbf{p}_{21}}{8} + \frac{\mathbf{p}_{22}}{16},$$

and using the identities already derived, we obtain the solution

$$\mathbf{p}_{11} \leftarrow (\mathbf{b}_{011} + \mathbf{b}_{110})/2.$$

Converting Cubic Triangles

A triangular cubic Bézier patch is specified by 10 control points $\mathbf{b}_{ijk}$ (see Fig. 2). Given these points, we need to find a 4×4 array of control points $\mathbf{p}_{ij}$ for the rectangular bicubic patch.

Similarly to the quadratic case, the "boundary" control points are the same in both representations:

$$\mathbf{p}_{00} \leftarrow \mathbf{b}_{003},$$

$$\mathbf{p}_{01} \leftarrow \mathbf{b}_{102},$$

$$\mathbf{p}_{02} \leftarrow \mathbf{b}_{201},$$

$$\mathbf{p}_{03} \leftarrow \mathbf{b}_{300},$$

$$\mathbf{p}_{10} \leftarrow \mathbf{b}_{012},$$

$$\mathbf{p}_{13} \leftarrow \mathbf{b}_{210},$$

$$\mathbf{p}_{20} \leftarrow \mathbf{b}_{021},$$

$$\mathbf{p}_{23} \leftarrow \mathbf{b}_{120},$$

$$\mathbf{p}_{30}, \ \mathbf{p}_{31}, \ \mathbf{p}_{32}, \ \mathbf{p}_{33} \leftarrow \mathbf{b}_{030}.$$
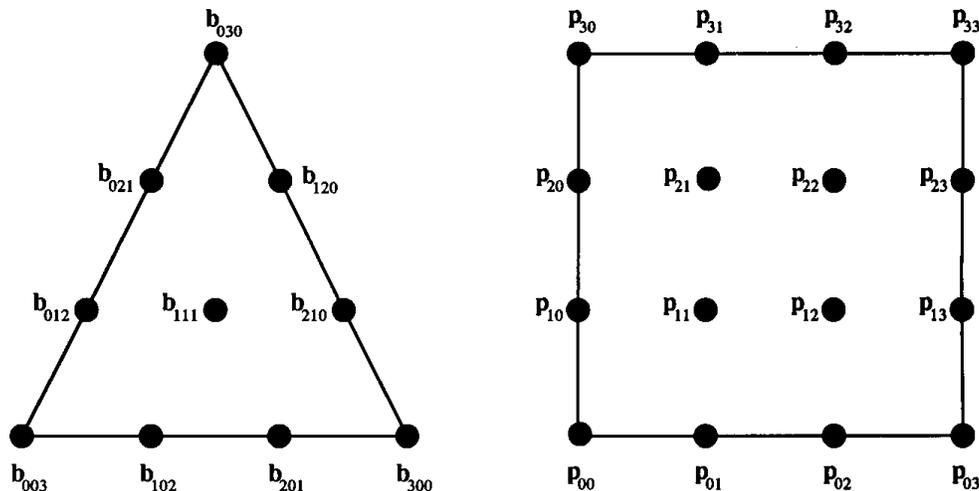
Figure 2. Control points defining a cubic Bézier triangle (left) and a bicubic Bézier patch (right)

We are left with the four interior points $\mathbf{p}_{11}$, $\mathbf{p}_{12}$, $\mathbf{p}_{21}$, $\mathbf{p}_{22}$ Their values can be obtained by solving the four simultaneous linear equations

$$T\left(\tfrac{2}{9}, \tfrac{1}{3}\right) = R\left(\tfrac{1}{3}, \tfrac{1}{3}\right),$$
$$T\left(\tfrac{4}{9}, \tfrac{1}{3}\right) = R\left(\tfrac{2}{3}, \tfrac{1}{3}\right),$$
$$T\left(\tfrac{1}{9}, \tfrac{2}{3}\right) = R\left(\tfrac{1}{3}, \tfrac{2}{3}\right),$$
$$T\left(\tfrac{2}{9}, \tfrac{2}{3}\right) = R\left(\tfrac{2}{3}, \tfrac{2}{3}\right),$$

which yields the solution

$$\mathbf{p}_{11} \leftarrow (\mathbf{b}_{012} + 2\mathbf{b}_{111})/3,$$

$$\mathbf{p}_{12} \leftarrow (2\mathbf{b}_{111} + \mathbf{b}_{210})/3,$$

$$\mathbf{p}_{21} \leftarrow (2\mathbf{b}_{021} + \mathbf{b}_{120})/3,$$

$$\mathbf{p}_{22} \leftarrow (\mathbf{b}_{021} + 2\mathbf{b}_{120})/3.$$

See also G1, 75.

V.12
CURVE TESSELLATION CRITERIA THROUGH SAMPLING

Terence Lindgren, Juan Sanchez, and Jim Hall
Prime/Computervision
Bedford, Massachusetts

The parametric curve is a widely used primitive in computer graphics. In this gem we will consider the problem of finding a suitable polyline representation for the curve. For our purposes we will consider a curve to be any $C2$ function, *F,* mapping an interval [*a, b*] into three-space. From the display perspective, the most important aspect of the final rendering is how the curve looks on the screen. Do the pixels that were selected to represent the curve look "$G1$ continuous"?

Basically, there are two approaches to rendering curves. One approach is to render the curve pixel by pixel. While this approach inherently "looks good," it is neither speedy nor the method of choice by most of today's graphics platforms. A second alternative is the one taken most frequently. The curve is converted into a polyline, and the polyline is rendered on the screen. While this approach results in fine performance, it forces the renderer to take great care in deciding which polyline to choose to represent the curve. A moment's reflection is all that is needed to see that if the polyline is always less than 1/2 pixels away from the true curve, the image will be the best achievable.

A common algorithm for generating a polyline close to a curve is to calculate the maximum distance from the curve to the line joining the endpoints of the curve. This distance is called the chordal deviation of the curve from the line segment. If the deviation is small enough, then the line may represent the curve; otherwise, the curve is subdivided into two halves. Each half is subjected to the same chordal deviation analysis. This procedure will generate line segments that are easily constructed into a polyline. However, this recursive generation of the line segments is slow.

point of
maximum
chordal
deviation
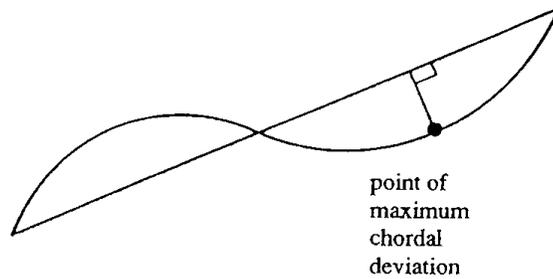
Figure 1.

Moreover, it can demand space, which is often a critical resource in graphics renderers.

Another approach, which avoids these problems, is to determine global parameter spaces δ to use in sampling the curve. The curve is then evaluated at points δ apart and a polyline is generated. This δ is chosen so that the maximal chordal deviation between the curve in the interval $[t, t + \delta]$ and the line segment joining $F[t]$ and $F[t + \delta]$ is less than some specified ε (1/2 pixels, for example). (See Fig. 1.) While this approach yields good rendering speeds, the calculation of a suitable δ is a complicated undertaking. We will present a heuristic algorithm that significantly reduces the complexities.

To understand the difficulties a bit better, let's examine the chordal deviation function, CD, of the curve $F: [a, b] \rightarrow R3$ with respect to the line segment joining $F[a]$ and $F[b]$. Let $(,)$ be the inner product defined on $R3$, and $| |$ be the corresponding norm operator. Let $P = F[t] - F[a]$ and $Q = F[b] - F[a]$, then CD is defined as

$$\text{CD}[t] \ = \ \bigl|P - ((P,Q)/(Q,Q)) * Q\bigr|.$$

The important result is an upper bound on the maximum value of CD, CD[max]:

$$|\text{CD}[max]| \leq \bigl(|\max \text{ CD}''| * (b - a)_2\bigr)/8 \tag{1}$$

where max CD$''$ is the maximum value of CD” on $[a, b]$. The interested reader may derive this bound by noting that if F *is* $C2$, then CD is also

$C2$, CD[a] = CD[b] = 0, (CD[max], CD$'$[max]) = 0, and for each t in [a, b], there is an s in [a, b] such that

$$CD[t] = CD[max] + CD'[max]*(t - max) + CD''[s]((t - max)2)/2.$$

Unfortunately, this theoretical bound is difficult to use in practice. There are two reasons for this. First, even though max CD$''$ is a simple function of max F'', max F'' may be a very difficult quantity to capture. This is especially true when the degree of F is greater than 3 or F is a rational function. Secondly, the derivation of (1) includes three inequalities, so even when we are able to calculate max CD$''$, our bound is likely to be too large. This results in polylines with many more segments than are needed to represent the curve.

Our solution involves using the theoretical bound as a way of understanding how the maximum chordal deviation converges to 0 as we approximate the curve with line segments whose endpoints are $F[a + m*\delta]$ and $F[a + (m + 1)*\delta]$ for $\delta = (b - a)/N$ and $m = 0 \cdots N - 1$, as N goes to infinity.

Our technique is to sample the curve at the points $a + m*(b - a)/P$ for some even integer P and for $m = 0 \cdots P$. Then for all $m = 1 \cdots P - 1$, we calculate the chordal deviation of point $F[a + m*(b - a)/P]$ from the line segment joining $F[a + (m - 1)*(b - a)/P]$ and $F[a + (m + 1)*(b - a)/P]$. We define α as the maximum of these deviations.

Our goal is to find the N that will ensure that the maximum deviation is within a tolerance of a given ε. First, as a heuristic we assume that α actually is the maximum chordal deviation the interval [a, b] for $\delta = |b - a|/P$. Our next step sets up the inequality

$$\left(\left(|\max \text{CD}''| * ((b - a)/N)^2\right)/8\right) \Big/ \left(\left(|\max \text{CD}''| * ((b - a)/P/2)^2\right)/8\right) \leq \varepsilon/\alpha$$

from which we obtain

$$(P/2N)^2 \leq \varepsilon/\alpha, \qquad \text{and}$$

$$P/2 * \sqrt{\alpha/\varepsilon} \leq N.$$

We have found letting $N = P/2 * \sqrt{\alpha/\varepsilon}$ to be an enticing choice. The result will be scaled if there is a change either in view state or a change in ε. In the former case we multiply N by $\sqrt{\text{norm}}$ of the viewing transformation, and in the latter case we multiply N by $\sqrt{\varepsilon/\varepsilon_{\text{new}}}$. It is true that scaling the calculations will introduce an inequality that may cause an overestimation of the number of segments in the polyline. However, we could always easily recalculate N. *Also,* the technique helps the problem of high degree and rationality. While we are quite pleased with the behavior of this algorithm, we do recognize that increasingly poorly parameterized curves will require increasingly larger sets of sample points to generate a meaningful δ. We also recognize that for curves with widely varying curvature, our approach will demand many samples in regions of the domain that did not need them, or will require an interval analysis to keep the number of evaluations small. Nonetheless, we have seen that setting $P = 2(\text{degree} + 1)$ generates good polyline approximations to the reasonable curves prevalent in CAD/CAM applications.

See also G1,64.

VI

RAY TRACING
AND RADIOSITY

VI

RAY TRACING AND RADIOSITY

Ray tracing and radiosity used to be fiercely competitive factions in the quest for ever more realistic rendering techniques. Now, they are often used together to produce images that are more compelling than those produced by either technique alone. Many wonderful images are created by hybrid algorithms which mix and match features from both ray tracing and radiosity. So, the Gems in this section are applicable to ray tracing, radiosity, or, in some cases, both!

The first Gem describes the well-known binary space partitioning algorithm very clearly and provides an efficient implementation. The second Gem describes the mathematics of intersecting a ray with a quadric surface.

For the efficiency-minded, there are three Gems that are intended to accelerate the process of tracing rays. The third Gem describes a construct known as "residency" masks to encode object position, which should be quite effective for accelerating ray tracing of simple scenes. The fifth Gem provides rectangular bounding volumes for some commonly used primitives, and the sixth Gem describes a technique for generating a bounding volume for an arbitrary set of points.

The fourth Gem describes an interesting viewing projection for rendering panoramic wide angle views. This Gem also provides an interesting discussion of the relation between computer graphics projections and those used by cartographers. The seventh Gem discusses the calculation of physically correct lighting for distribution ray tracing.

The last four Gems are oriented more toward the radiosity side of the fence. The eighth Gem describes how to project a triangle onto the face

of a hemisphere. The ninth Gem describes a radiosity formulation that uses vertex-to-vertex form factors. This Gem also suggests that the polygon rendering hardware present in many graphics workstations can be used to accelerate this operation. The tenth Gem is an implementation tip for a previous Gem, concerning the cubic tetrahedral radiosity algorithm, an alternative to the hemicube technique. The final Gem discusses the accurate computation of area-to-area form factors.

VI.1
RAY TRACING WITH THE BSP TREE

Kelvin Sung
University of Illinois   *and*   Peter Shirley
Urbana, Illinois   *Indiana University*
Bloomington, Indiana

Introduction

In order to speed up the intersection calculation in ray tracing programs, people have implemented divide-and-conquer strategies such as hierarchical bounding volumes and octrees. Uniform subdivision (essentially a three-dimensional binsort) has also been used to speed up this calculation.

Uniform subdivision is undesirable for applications where the objects may be unevenly distributed in space. This is because the amount of memory needed for uniform subdivision is proportional to the highest density of objects, rather than the total number. Hierarchical bounding volumes can be difficult to implement effectively, but can be used to good effect (Kay and Kajiya, 1986). Hierarchical space subdivision techniques do not suffer the memory problems of uniform subdivision and are also relatively easy to implement. In this Gem, we discuss what we think is the best overall hierarchical subdivision technique currently known.

It is often believed that adaptive spatial subdivision approaches to accelerating the tracing of a ray are messy and hard to implement. In our experience with different spatial structures and traversal algorithms, we have found this view to be untrue. It is straightforward to implement the Linear Time Tree Walking algorithm, as proposed by Arvo (1988), on a Binary Space Partitioning (BSP) tree. The resulting system outperforms all of the spatial subdivision approaches we have experienced.

We have implemented and compared the performance of several traversal algorithms on an octree (Glassner, 1984; Arvo, 1988; Sung, 1991) and on a BSP tree (Kaplan, 1985; Arvo, 1988). In order to obtain meaningful

comparisons, we have kept the rest of our ray tracing system unchanged while replacing the spatial structure building and traversal components for each of these methods. Our experience shows that the Linear Time Tree Walking method (Arvo, 1988) is consistently at least 10% faster than the rest and is usually better than that. We have observed that implementing the tree walking algorithm on a BSP tree and on an octree results in similar performance, but that the implementation is more straightforward on a BSP tree. Finally, it should be pointed out that the *recursive traversal* algorithm introduced independently by Jansen (1986) is a different implementation of the same tree walking idea.

There are two basic modules in a BSP tree intersection code. The first module builds the tree by recursively cutting the bounding box of all objects along a median spatial plane. The second module tracks a ray through the leaf nodes of the BSP tree checking for intersections.

The BSP tree is built by *InitBinTree()* and *Subdivide(), RayTreeIntersect() is* the actual tree walking traversal of the BSP tree following a ray. These are the key functions necessary to implement the algorithm. Subdivide( ) builds the BSP tree by subdividing along the midpoint of the current node's bounding volume.

```
Subdivide (Current Node, CurrentTree Depth, CurrentSubdividingAxis)
    if ((CurrentNode contains too many primitives) and (CurrentTreeDepth
    is not too deep)) then
        begin
            Children of CurrentNode ← CurrentNode's Bounding Volume
            /*Note that child[0].max.DividingAxis and
            Child[1].min.DividingAxis are always equal.*/
            /*Depending on CurrentSubdividingAxis, DividingAxis can be
            either x, y, or z.* /
            if (CurrentSubdividingAxis is X-Axis) then begin
                child[0].max.x ← child[1].min.x ← mid-point of CurrentNode's
                X-Bound
                NextSubdivideAxis ← Y-Axis
            end else if (CurrentSubdividingAxis is Y-Axis) then begin
                    child[0].max.y ← child[1].min.y ← mid-point of
                    CurrentNode's Y-Bound
                    NextSubdivideAxis ← Z-Axis
                end else begin
                        child[0].max.z ← child[1].min.z ← mid-point of
                        CurrentNode's Z-Bound
                        NextSubdivideAxis ← X-Axis
```

```
                    end
            for (each of the primitives in CurrentNode's object link list) do
                if (the primitive is within children's bounding volume) then
                        add the primitive to the children's object link list
                Subdivide (child[0], CurrentTreeDepth + 1, NextSubdivideAxis)
                Subdivide (child[1], CurrentTreeDepth + 1, NextSubdivideAxis)
        end
```

As suggested by Arvo (1988), RayTreeIntersect( ) avoids recursive proce-
dure calls in the inner loop of tree walking by maintaining an explicit
stack. The pseudo-code given here is based on Arvo's article in the Ray
Tracing News, where recursion is used for ease of understanding. When
calling RayTreeIntersect( ) the first time, initial values of *min* and *max*
should be the distances (measured from the ray origin along the ray
direction) to the two intersecting points between the ray and the bound-
ing volume of the root of the BSP tree. Notice that if a ray originates from
inside the BSP tree, then the initial value of *min* will be negative.

```
RayTreeIntersect (Ray, Node, min, max)
    if (Node is NIL) then return ["no intersect"]
    if (Node is a leaf) then begin
        intersect Ray with each primitive in the object link list
            discarding those farther away than "max"
        return ["object with closest intersection point"]
    end
    dist ← signed distance along Ray to the cutting plane of the Node
    near ← child of Node for half-space containing the origin of Ray
    far ← the "other" child of Node—i.e. not equal to near
    if ((dist > max) or (dist < 0)) then        /*Whole interval is on near side*/
        return [RayTreeIntersect (Ray, near, min, max)]
    else if (dist < min) then                   /*Whole interval is on far side*/
        return [RayTreeIntersect (Ray, far, min, max)]
    else begin                                  /*the interual intersects the plane*/
        hit_data ← RayTreeIntersect (Ray, near, min, dist)    /*test near side*/
        if hit_data indicates that there was a hit then return [hit_data]
        return [RayTreeIntersect (Ray, near, dist, max)]        /*test far side*/
    end
```

There are a few standard link list functions that are not listed in the C code: *FirstOfLinkList( ), NextOfLinkLIst( ), AddToLinkList( ),* and *DuplicateLinkList( ).*

One possible improvement over the basic algorithm is to keep track of the nearest intersection and its distance, regardless of whether it is inside of the current node. With this information, as soon as a node is found that starts beyond this nearest intersection distance, we know we are done.

If an object spans multiple tree nodes (spatial cells), then the intersection calculation between this object and a given ray may need to be carried out multiple times, once in each node traversed by the ray. The mailbox idea was proposed (Amanatides and Woo, 1987; Arnaldi *et al*, 1987) to avoid this problem. However, it is observed that this technique is effective only when primitives in the scene are large compared to the spatial cells. When the size of the primitives in a scene is small compared to the size of the spatial cells, the mailbox implementation may slow down the rendering process (Sung, 1991). Also, the mailbox implementation requires the scene database to be updated after each intersection calculation. This implies that to parallelize the algorithm, some kind of database coherence policy must be administrated; this would increase the complexity of an implementation. Based on these observations, we have chosen not to include the mailbox enhancement in our code.

Other work on BSP trees includes Fussell and Subramanian (1988), MacDonald and Booth (1989), and Subramanian and Fussell (1991).

Acknowledgments

See also G3, E.2.

VI.2
INTERSECTING A RAY WITH A QUADRIC SURFACE

Joseph M. Cychosz
Purdue University CADLAB
West Lafayette, Indiana

and

Warren N. Waggenspack Jr.
IMRLAB, Mechanical Engineering
Louisiana State University
Baton Rouge. Louisiana

Quadric surfaces are common modeling primitives for a variety of computer graphics and computer aided design applications (Glassner *et al.*, 1989; Blinn, 1984; Gardiner, 1984; Roth, 1982). Ray tracing or ray firing is also a popular method used for realistic renderings of quadric surfaces. Summarized in the Gem is an algorithm for locating the intersection point(s) between a ray and general quadratic surfaces including ellipsoids, cones, cylinders, paraboloids and hyperboloids. Also detailed is a means for determining the surface normal at the point of intersection. The normal is required in lighting model computations.

The Equation of a Quadratic Surface

The general implicit equation for a quadric surface can be written in the expanded algebraic form

$$\mathbf{F}(x,\ y,\ z)\ =\ ax^2\ +\ 2bxy\ +\ 2cxz\ +\ 2dx\ +\ ey^2$$

$$+\ 2fyz\ +\ 2gy\ +\ hz^2\ +\ 2iz\ +\ j\ =\ 0, \tag{1}$$

or it can be expressed in a more compact, symmetric 4×4 coefficient

matrix form (Q), sometimes referred to as the quadric form:

$$f(x,\ y,\ z)\ =\ [x\ y\ z\ 1]\begin{bmatrix} a & b & c & d \\ b & e & f & g \\ c & f & h & i \\ d & g & i & j \end{bmatrix}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \mathbf{X}\mathbf{Q}\mathbf{X}^{\mathrm{T}} = 0. \tag{2}$$

For a variety of common quadric surfaces in standard position, i.e., positioned at the origin with the y-axis being the axis of symmetry, the coefficient matrices can be quickly constructed. These include the following equations (2a–2e).

Ellipsoid: With axis lengths of $2a$, $2b$, and $2c$ along the principal directions. A sphere of radius r is simply a special case of the ellipsoid in which $r\ =\ a\ =\ b\ =\ c$.

$$f(x,\ y,\ z)\ =\ \frac{x^2}{a^2}\ +\ \frac{y^2}{b^2}\ +\ \frac{z^2}{c^2}\ -\ 1\ =\ \mathbf{X}\begin{bmatrix} \frac{1}{a^2} & 0 & 0 & 0 \\ 0 & \frac{1}{b^2} & 0 & 0 \\ 0 & 0 & \frac{1}{c^2} & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}\mathbf{X}^{\mathrm{T}}\ =\ 0.$$

$$\tag{2a}$$

Elliptical cylinder: With principal axis lengths of $2a$ and $2c$. As before, a circular cylinder of radius r is simply a special case in which $r\ =\ a\ =\ c$.

$$f(x,\ y,\ z)\ =\ \frac{x^2}{a^2}\ +\ \frac{z^2}{c^2}\ -\ 1\ =\ \mathbf{X}\begin{bmatrix} \frac{1}{a^2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{c^2} & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}\mathbf{X}^{\mathrm{T}}\ =\ 0. \tag{2b}$$

Elliptical cone: With principal axis lengths of $2a$ and $2c$ at a unit distance from the apex. Once again, a right circular cone is simply a special case in which $r = a = c$.

$$f(x, y, z) = \frac{x^2}{a^2} - y^2 + \frac{z^2}{c^2} = \mathbf{X} \begin{bmatrix} \frac{1}{a^2} & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & \frac{1}{c^2} & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \mathbf{X}^{\mathrm{T}} = 0. \qquad (2c)$$

Elliptical paraboloid: With principal axis lengths of $2a$ and $2c$ at twice the focal distance f.

$$f(x, y, z) = \frac{x^2}{a^2} + \frac{z^2}{c^2} - 4fy = \mathbf{X} \begin{bmatrix} \frac{1}{a^2} & 0 & 0 & 0 \\ 0 & 0 & 0 & -2f \\ 0 & 0 & \frac{1}{c^2} & 0 \\ 0 & -2f & 0 & 0 \end{bmatrix} \mathbf{X}^{\mathrm{T}} = 0.$$

$$(2d)$$

Elliptical hyperboloid (of one sheet): With principal axis vertex distances of $2a$ and $2c$ at the origin, and b determining the asymptotes.

$$f(x, y, z) = \frac{x^2}{a^2} - \frac{y^2}{b^2} + \frac{z^2}{c^2} - 1 = \mathbf{X} \begin{bmatrix} \frac{1}{a^2} & 0 & 0 & 0 \\ 0 & -\frac{1}{b^2} & 0 & 0 \\ 0 & 0 & \frac{1}{c^2} & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \mathbf{X}^{\mathrm{T}} = 0.$$

$$(2e)$$

Any of the standard quadric surfaces can be transformed to an arbitrary position and orientation using a Euclidean transformation $\mathbf{T}$. The equation $f^*(x, y, z)$ for the transformed quadric surface is given by the expression

$$f^*(x, y, z) = \mathbf{X}\mathbf{T}\mathbf{Q}\mathbf{T}^\mathrm{T}\mathbf{X}^\mathrm{T} = \mathbf{X}\mathbf{Q}^*\mathbf{X}^\mathrm{T} = 0,$$

where as indicated $\mathbf{Q}^* = \mathbf{T}\mathbf{Q}\mathbf{T}^\mathrm{T}$. It is important to note here that transforming implicit equations is essentially a change of coordinate reference frame or basis and requires the inverse mapping of the standard rigid body transformations often used throughout computer graphics. For example, suppose a mapping $\mathbf{M}$ is applied to all points on a unit sphere that translates the center of the sphere to the location (1, 1, 1). The transformation $\mathbf{T}$ used in mapping the implicit equation of the sphere must be the inverse of $\mathbf{M}$:

$$f^*(x, y, z) = (x - 1)^2 + (y - 1)^2 + (z - 1)^2 - 1$$

$$= x^2 - 2x + 1 + y^2 - 2y + 1 + z^2 - 2z + 1 - 1$$

$$= \mathbf{X}\begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ -1 & -1 & -1 & 2 \end{bmatrix}\mathbf{X}^\mathrm{T}$$

$$= \mathbf{X}\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -1 & -1 & -1 & 1 \end{bmatrix}$$

$$\times \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}\mathbf{X}^\mathrm{T}$$

$$= \mathbf{X}\mathbf{T}\mathbf{Q}\mathbf{T}^\mathrm{T}\mathbf{X}^\mathrm{T} = \mathbf{X}\mathbf{M}^{-1}\mathbf{Q}\mathbf{M}^{-\mathrm{IT}}\mathbf{X}^\mathrm{T} = \mathbf{X}\mathbf{Q}^*\mathbf{X}^\mathrm{T} + 0.$$

The position and orientation of a quadric can be conveniently defined using any two of the three unit vectors describing the local coordinate

system of that quadric. Suppose that $\mathbf{U}$ and $\mathbf{V}$ are unit vectors representing the local x and y axes (recall that the local y axis is the axis of symmetry) in terms of the global coordinate system. Let B define the appropriate base point for translating the quadric surface, i.e., the local origin. The base point is, for example, the center of an ellipsoid, or the apex of a cone. From these two unit vectors and the base point, the 4×4 transformation $\mathbf{M}$ for mapping individual points on the quadric surface and $\mathbf{T}$ for transforming the implicit equation are given by the following expressions. This formulation for $\mathbf{T}$ is similar to an observation transformation frequently applied in the 3-D viewing pipeline of computer graphics (Foley and Van Dam, 1982; Gasson, 1983).

$$\mathbf{M} = \begin{bmatrix} \mathbf{U} & 0 \\ \mathbf{V} & 0 \\ \mathbf{U} \times \mathbf{V} & 0 \\ B & 1 \end{bmatrix} = \mathbf{T}^{-1},$$

$$\mathbf{T} = \begin{bmatrix} \mathbf{U}^{\mathrm{T}} & \mathbf{V}^{\mathrm{T}} & (\mathbf{U} \times \mathbf{V})^{\mathrm{T}} & \begin{matrix} 0 \\ 0 \\ 0 \end{matrix} \\ -B \cdot \mathbf{U} & -B \cdot \mathbf{V} & -B \cdot (\mathbf{U} \times \mathbf{V}) & 1 \end{bmatrix} = \mathbf{M}^{-1}.$$

More general transformations including scaling, reflection, and shearing can also be included but require a different formulation for $\mathbf{M}$ and $\mathbf{T}$ than that shown.

Intersection of a Ray with a Quadric Surface

The intersection of a ray with a quadric surface can be found by substituting the vector expression of the parametric ray equations (Eq. (3) below) into the matrix form of the general quadric surface (Eq. (2)). This results in a quadratic polynomial in the ray parameter, t, which is proportional to the distance(s) along the ray at which the intersection(s) with the quadric

surface occur:

$$
\left.
\begin{array}{l}
x = x_0 + a_x t \\
y = y_0 + a_y t \\
z = z_0 + a_z t
\end{array}
\right\} \Rightarrow \mathbf{X} = X_0 + \mathbf{R}t.
\tag{3}
$$

Substitution into the matrix form of the implicit quadric, expanding, and taking advantage of the matrix symmetry produces the following result:

$$
f(x,\ y,\ z) = \mathbf{XQX}^{\mathrm{T}} = [X_0 + \mathbf{R}t]\mathbf{Q}[X_0 + \mathbf{R}t]^{\mathrm{T}}
$$

$$
= X_0\mathbf{Q}\,X_0^{\mathrm{T}} + t(X_0\mathbf{Q}\mathbf{R}^{\mathrm{T}} + \mathbf{RQ}\,X_0^{\mathrm{T}}) + t^2\mathbf{RQR}^{\mathrm{T}},
$$

$$
= k_2 t^2 + k_1 t + k_0 = f(t) = 0,
\tag{4}
$$

where

$$
k_2 = \alpha_x(\alpha_x\mathbf{Q}_{11} + \alpha_y(\mathbf{Q}_{12} + \mathbf{Q}_{21}))
$$

$$
\quad + \alpha_y(\alpha_y\mathbf{Q}_{22} + \alpha_z(\mathbf{Q}_{23} + \mathbf{Q}_{32}))
$$

$$
\quad + \alpha_z(\alpha_z\mathbf{Q}_{33} + \alpha_x(\mathbf{Q}_{13} + \mathbf{Q}_{31})),
$$

$$
k_1 = \alpha_x((z_0(\mathbf{Q}_{13} + \mathbf{Q}_{31}) + y_0(\mathbf{Q}_{12} + \mathbf{Q}_{21}) + x_0(2\mathbf{Q}_{11})) + (\mathbf{Q}_{14} + \mathbf{Q}_{41}))
$$

$$
\quad + \alpha_y((z_0(\mathbf{Q}_{23} + \mathbf{Q}_{32}) + x_0(\mathbf{Q}_{12} + \mathbf{Q}_{21}) + y_0(2\mathbf{Q}_{22})) + (\mathbf{Q}_{24} + \mathbf{Q}_{42}))
$$

$$
\quad + \alpha_z((y_0(\mathbf{Q}_{23} + \mathbf{Q}_{32}) + x_0(\mathbf{Q}_{13} + \mathbf{Q}_{31}) + z_0(2\mathbf{Q}_{33})) + (\mathbf{Q}_{34} + \mathbf{Q}_{43})),
$$

$$
k_0 = x_0(x_0\mathbf{Q}_{11} \qquad + y_0(\mathbf{Q}_{12} + \mathbf{Q}_{21}) + z_0(\mathbf{Q}_{13} + \mathbf{Q}_{31}) + (\mathbf{Q}_{14} + \mathbf{Q}_{41}))
$$

$$
\quad + y_0(\qquad\quad y_0\mathbf{Q}_{22} \qquad\qquad + z_0(\mathbf{Q}_{23} + \mathbf{Q}_{32}) + (\mathbf{Q}_{24} + \mathbf{Q}_{42}))
$$

$$
\quad + z_0(\qquad\qquad\qquad\qquad\qquad z_0\mathbf{Q}_{33} \qquad\qquad + (\mathbf{Q}_{34} + \mathbf{Q}_{43})) + \mathbf{Q}_{44}.
$$

To simplify the ray-quadric intersection with arbitrarily positioned and oriented quadrics, the Euclidean map $\mathbf{M}$ defined earlier can be used to transform the ray base X_0 and direction cosines $\mathbf{R}$ into equivalent rays (X_0^* and $\mathbf{R}^*$) relative to the local coordinate frame of the quadric. In this manner, geometric queries are resolved to dealing with quadrics in standard position:

$$X_0^* = X_0\mathbf{M}; \qquad \mathbf{R}^* = \mathbf{RM},$$

$$X^* = X_0^* + t\mathbf{R}^*.$$

Once the substitution is completed, the real roots of the quadratic polynomial in the parameter t are determined. Possible outcomes include no intersections (no real roots), a tangential intersection (repeated real roots), or two distinct points of intersection (two distinct real roots) between the ray and the quadric surface.

With $\mathbf{M}$ being a Euclidean map, the real parameter values, t_{int}, determined can be used to locate the intersection in either the *local* or true *global* coordinates of the intersection point:

$$X_{Int/Global} = X_0 + t_{int}\mathbf{R},$$

$$X_{Int/Local} + X_0^* + t_{int}\mathbf{R}^*.$$

An Efficient Ray Rejection Test

A quick and efficient bounding test for eliminating unnecessary ray-quadric surface intersections involves bounding the quadric with an infinite cylinder. Most quadrics, excluding spheres and ellipsoids, will be truncated or bounded along their symmetry axis by a pair of "end cap" planes. The position of these end caps sets the maximum distance the quadric surface is allowed to extend away from its symmetry axis and is used to set the radius of the corresponding bounding cylinder (r). When the shortest distance between the ray and the symmetry axis of the

quadric is greater than the radius of the bounding cylinder, no intersection exists, and there is no need for further unnecessary computations.

Using vector algebra, it can be shown that the perpendicular distance d between the ray and the axis of the arbitrarily positioned quadric surface is defined as shown where X_0, B, $\mathbf{R}$, and $\mathbf{V}$ are the points and vectors defined earlier:

$$d = |(B - X_0) \cdot (\mathbf{V} \times \mathbf{R})|.$$

Should the ray and symmetry axis of the quadric be parallel, the distance d can be computed as follows. This second expression also represents the shortest distance between the ray and the center ($\mathbf{B}$) of an ellipsoid and can therefore be used to provide a quick elimination test for the corresponding ray-ellipsoid intersections:

$$d = \|B - X_0\| \sqrt{1 - (B - X_0) \cdot \mathbf{R}} = \|(B - X_0) \times \mathbf{R}\|$$

Determining the Surface Normal of a Quadric Surface

The normal to an implicitly defined surface $f(x, y, z) = 0$ at intersection point, P, is established by evaluating the gradient of the surface function at that point. This entails computing the partial derivatives of the surface, f, with respect to each of the coordinates x, y, and z:

$$\mathbf{N}(x, y, z)\big|_P = \nabla f(x, y, z)\big|_P$$

$$= \left[\frac{\partial f(x, y, z)}{\partial x}\bigg|_P , \frac{\partial f(x, y, z)}{\partial y}\bigg|_P , \frac{\partial f(x, y, z)}{\partial z}\bigg|_P \right].$$

Recalling the compact matrix expression for the general quadric surface (Eq. (2)), the partial derivatives and normal vector are resolved to the

following set of computations:

$$\mathbf{N}_x = \frac{\partial f(x,\ y,\ z)}{\partial x} = \frac{\partial \mathbf{X}}{\partial x}\mathbf{Q}\mathbf{X}^{\mathrm{T}} + \mathbf{X}\mathbf{Q}\frac{\partial \mathbf{X}^{\mathrm{T}}}{\partial x} = 2\frac{\partial \mathbf{X}}{\partial x}\mathbf{Q}\mathbf{X}^{\mathrm{T}}$$

$$= 2\begin{bmatrix}1 & 0 & 0 & 0\end{bmatrix}\mathbf{Q}\mathbf{X}^{\mathrm{T}} = 2(ax + by + cz + d),$$

$$\mathbf{N}_y = \frac{\partial f(x,\ y,\ z)}{\partial y} = \frac{\partial \mathbf{X}}{\partial y}\mathbf{Q}\mathbf{X}^{\mathrm{T}} + \mathbf{X}\mathbf{Q}\frac{\partial \mathbf{X}^{\mathrm{T}}}{\partial y} = 2\frac{\partial \mathbf{X}}{\partial y}\mathbf{Q}\mathbf{X}^{\mathrm{T}}$$

$$= 2\begin{bmatrix}0 & 1 & 0 & 0\end{bmatrix}\mathbf{Q}\mathbf{X}^{\mathrm{T}} = 2(bx + ey + fz + g),$$

$$\mathbf{N}_z = \frac{\partial f(x,\ y,\ z)}{\partial z} = \frac{\partial \mathbf{X}}{\partial z}\mathbf{Q}\mathbf{X}^{\mathrm{T}} + \mathbf{X}\mathbf{Q}\frac{\partial \mathbf{X}^{\mathrm{T}}}{\partial z} = 2\frac{\partial \mathbf{X}}{\partial z}\mathbf{Q}\mathbf{X}^{\mathrm{T}}$$

$$= 2\begin{bmatrix}0 & 0 & 1 & 0\end{bmatrix}\mathbf{Q}\mathbf{X}^{\mathrm{T}} = 2(cx + fy + hz + i).$$

The vector $\mathbf{N}$ represents a non-unit surface normal to an arbitrary quadric surface. Where necessary, this vector can be normalized to a unit length. If the ray-quadric intersection points and gradient are determined in the local coordinate frame of the quadric, then the Euclidean mapping $\mathbf{M}$ must also be applied to the resulting normal vector $\mathbf{N}$.

See also G2, 251

VI.3
USE OF RESIDENCY MASKS AND OBJECT SPACE PARTITIONING TO ELIMINATE RAY—OBJECT INTERSECTION CALCULATIONS

Joseph M. Cychosz
Purdue University CADLAB
West Lafayette, Indiana

Introduction

Traditionally, ray tracing has been a computationally intensive process. A large portion of the computation is used to compute the intersection of the sampling rays with the objects that make up the scene. A number of algorithms and approaches have been published that reduce the number of ray-object intersection calculations through the use of bounding volumes that either enclose objects in the scene (Rubin and Whitted, 1980; Weghorst *et al.,* 1984; Kay and Kajiya, 1986) or partition the object space (Fujimoto *et al.,* 1986; Glassner, 1984; Kaplan, 1985). Alternative approaches to improving ray tracing performance exploit specific architectural features of a given computer, such as processor parallelism (Plunkett, 1984; Plunkett and Bailey, 1985; Cychosz, 1986; Dippé and Swensen, 1984; Cleary *et al.,* 1983; Nishimura et *al.,* 1983: Delany, 1988).

In object space partitioning, the extent of the objects within the scene is divided into a number of smaller spaces or cells. Each cell is then assigned a list of objects that are contained within it. Partitioning of the object space can be done in a number of ways Fujimoto *et al.* (1986) used uniform subdivision, Glassner (1984) used octrees, and Kaplan (1985) used BSP trees. The sampling rays are then traversed from cell to cell until a satisfactory conclusion is reached in determining the visible surface for the ray. As each cell is processed, the ray is intersected with the list of objects that reside within the cell. As pointed out by Kaplan (1985), this can result in unnecessary duplicated intersection calculations for objects that span multiple cells—for example, a long thin cylinder.

The Boolean properties of residency masks combined with spatial partitioning can be used either to eliminate duplicated intersection calculations, or as a selection mechanism for determining which objects in a scene must be examined.

Residency Masks

A residency mask is simply a bit vector in which each bit is assigned to a cell within the partitioned object space. A residency mask is assigned to each object defining the cells in which the given object resides. Figure 1 illustrates the residency masks for four objects and a ray. During the ray firing process, only objects A, B, and C need to be tested for intersection with the given ray. Furthermore, objects A and B need be intersected only once.

By updating the ray mask as it passes from cell to cell, a quick determination of whether an object within the cell needs to be intersected can be made by simply ANDing the residency mask of the object with the complement of the residency mask for the ray. If the result is nonzero, the object has already been intersected. This application of residency masks is analogous to the *mailbox* technique proposed by Arnaldi *et al.* (1987) which avoids duplicated intersections by tagging the objects with unique ray identifiers. If at some later time, the value of the tag for the object matches the ray identifier, the object has already been intersected, and

Item	Bit Mask			
Object A	0000	0000	0110	0110
Object B	0000	0000	1100	1100
Object C	0000	0010	0000	0000
Object D	1100	0000	0000	0000
Ray a	0000	0011	1110	0000
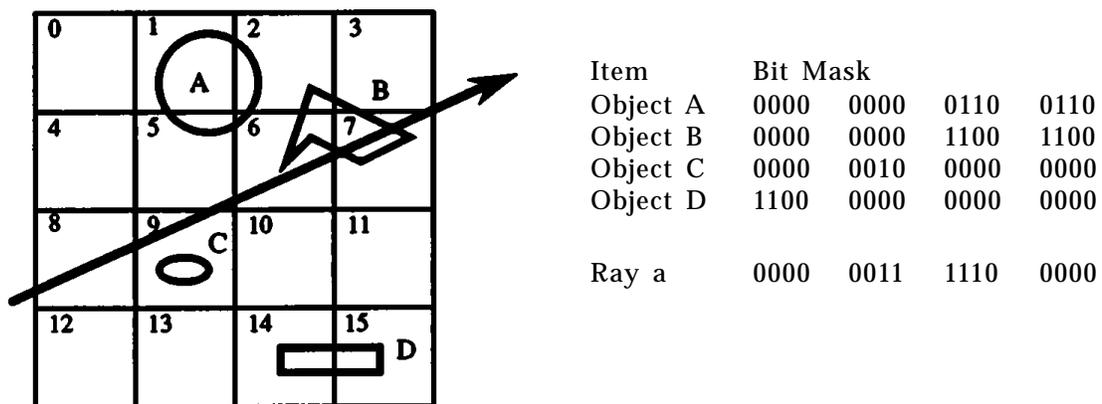
Figure 1. 2-D representation of ray-object residency masks.

thus the current query on the object can be skipped. The advantage residency masks has over *mailboxes* is that the object; database does not have to be updated during the ray firing process.

In the example shown in Fig. 1, the ray will pass through cells 8, 9, 5, 6, and possibly 7, depending on whether the visibility of B call be determined in cell 6. The object list for cell 5 will contain object A, and cell 6 will contain objects A and B. Therefore, object A. will be intersected in cell 5 and object B will be intersected in cell 6, thereby eliminating the unnecessary intersection calculation for objects A and B as the ray passes through cells 6 and 7, respectively.

For algorithms that must process a cluster of rays as a group—for example, the ray queue in vectorized ray tracing (Plunkett, 1984; Plunkett and Bailey, 1985; Cychosz, 1986)—a group residency mask can be used as a selection mechanism to determine if an object needs to be intersected with at least one member of the group of rays. This is important since it is unlikely that the rays within the group will coherently pass through the same cells. To illustrate this, the situation shown in Fig.1 can be modified to depict the residency mask for a queue of rays. Shown in Fig. 2 are the residency masks for the individual rays a, b, and c, and the resulting mask for the ray group. Determination of a list of objects that need to be considered by the group can be made by ANDing the residency mask for a given object with the residency mask of the ray group. If the result is nonzero, the object must be added to the list of objects to be

Item	Bit Mask			
Object A	0000	0000	0110	0110
Object B	0000	0000	1100	1100
Object C	0000	0010	0000	0000
Object D	1100	0000	0000	0000
Ray a	0000	0011	1110	0000
Ray b	0001	0011	0110	1100
Ray c	0000	1111	0000	0000
Ray Group	0001	1111	1110	1100
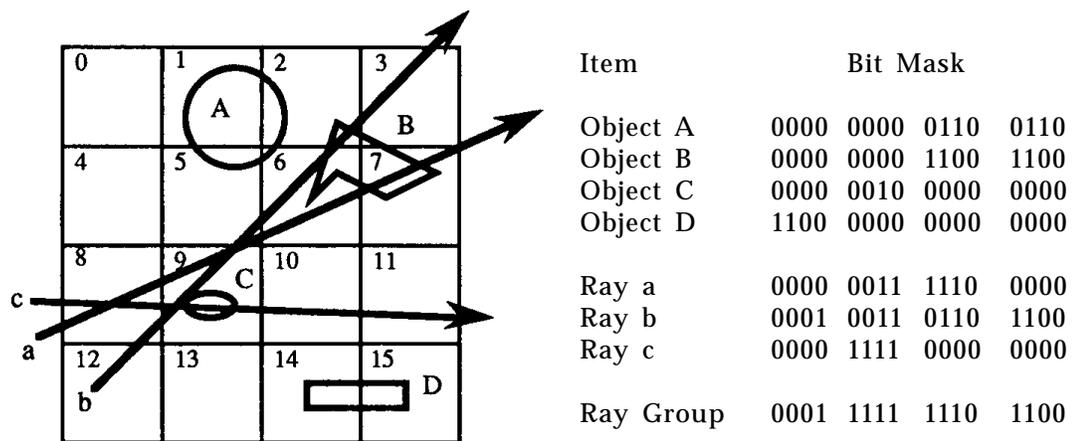
Figure 2. 2-D representation or residency masks for a ray group.

considered by the ray group. In the example shown in Fig. 2, the list of objects would consist of A, B, and C.

Effective performance of this bounding model relies on the assumption that the rays in the group can be geometrically bundled as tightly as possible, thereby minimizing the number of cells associated with the group of rays. This is true for the initial rays that are fired through the image plane, and for the shadow rays that are fired toward a common point, namely the light source being tested. For secondary rays resulting from reflections and refractions, a scattering occurs that reduces the effectiveness of the algorithm (this is especially true for surfaces with large changes in gradient). However, most scattering rays will remain in the same half-space (such as a group of rays striking a sphere).

See *also* G1, 385; G2, 264.

VI.4

A PANORAMIC VIRTUAL SCREEN FOR RAY TRACING

F. Kenton Musgrave
Yale University
New Haven, Connecticut

Introduction

With ray tracing's synthetic camera model, we can do something that would be extremely difficult, perhaps even impossible, to do with a real camera: create a 360° by 180° field-of-view panoramic "photograph." The standard imaging model used in ray tracing is that of a pinhole camera with a flat *virtual screen.* We can supplement this model with a cylindrical virtual screen to obtain a 360° (or greater) lateral field of view. By using appropriate angular distribution of samples on the vertical axis, we can obtain a 180° vertical field of view as well.

The *standard virtual screen* model used in ray tracing is equivalent to placing a piece of graph paper in the "world," in front of the eye and perpendicular to the view direction, then firing rays through the little squares (the pixels) on the grid. (A notable improvement to this naïve scheme was described by Mitchell, 1987.) Changing the field of view at a fixed image resolution corresponds to moving the piece of graph paper closer to, or farther away from, the eye. This scheme provides a good projection for relatively narrow fields of view, but it breaks down for wide angles: If we attempt to obtain a 180° field of view, the construction of the viewing projection becomes degenerate, as the eye lies in the plane of the screen. A field of view of greater than 180° in this scheme is, of course, nonsense.

Thus, with a standard virtual screen we can only approach, never achieve, a field of view of 180°. We can also see that as we approach the 180° field of view, the distortion introduced by the regular spatial sampling of the virtual screen grows: Near the center of the screen the
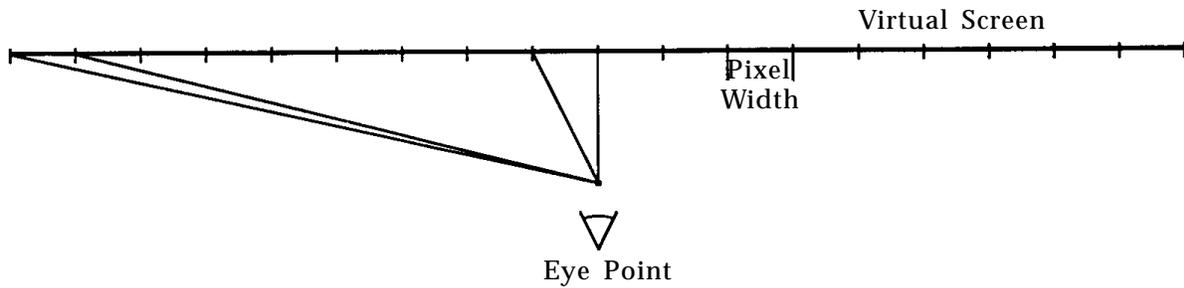
Figure 1. Angular width of a pixel as a function of position on the virtual screen: pixels near the edge of the screen subtend a smaller angle.

angular width of a pixel is much greater than near the edges (see Fig. 1). This distortion has the effect of, among other things, giving a sphere imaged near the edge of the screen, a pronouncedly elliptical projection on the image plane.[1] In a landscape image, features around the horizon get pinched down into a line as the vertical field of view approaches 180°.

Some distortion on the screen due to the viewing projection is inevitable. This is because the projection is an instance of a mapping of a sphere (the "celestial sphere" surrounding the eye) onto a plane (the image plane). The fact is, there exists no mapping from the sphere to the plane $f: S^2 \to R^2$, such that $f(x) - f(y) = x - y$.[2] Cartographers have long known this; hence the plethora of cartographic projections for maps of the globe (Paeth, 1990). Thus, we may choose among various evils, among various distortions in our images of the world, but we cannot avoid them altogether. (It is interesting to note that the same holds true for designers of camera lenses.)

In this Gem we describe a scheme for sampling a virtual screen in such a way that we can map the entire celestial sphere onto a rectilinear image plane, with (what we deem to be) "acceptable" distortion. Our viewing projection, known to cartographers as a *cylindrical equirectangular* projection (Paeth, 1990), generates a lateral-stretching distortion. The magnitude of this stretching grows with distance from the equator, i.e.,

---

[1]For an example of this, see the moon on the cover of the January, 1989 edition of *IEEE Computer Graphics and Applications.*

[2]One can find a proof of this in a textbook on projective or differential geometry.

with distance from the horizontal bisector of the image. Our projection does not become degenerate, however, at a 180° vertical field of view.

Cylindrical Virtual Screen

With the standard virtual screen, the horizontal field of view is determined by the width of the virtual screen in world space, and its distance from the eye. For a given (finite) world-space width of the virtual screen, as the horizontal field of view goes to 180°, its distance from the eye must go to zero. At 180°, the construction is degenerate, as the eye lies in the plane of the virtual screen.

In the case of the standard virtual screen, jittering aside, samples are generally taken at regular (e.g., equally spaced) intervals on the screen. The vector defining a primary ray (i.e., a ray from the eye that samples the virtual screen) is therefore generally determined by taking the vector difference of the sample point on the virtual screen and the eye point, and normalizing the resultant vector. As the sample cells on the virtual screen are equally spaced in screen space, we can determine the *x* (i.e., horizontal) offset as a linear function of the screen column being sampled:

$$x\_offset \ [i] \ = \ i*sample\_spacing$$

for i ∈ [*–screen_width/2 .. screen_width/2*].

For a cylindrical virtual screen, the construction of the primary ray is not quite so simple. We need a linear increment in *angle,* not in screen space. This can be accomplished by applying a rotation to a ray directed at the center of the virtual screen. As this requires a matrix multiplication, and thus several floating-point operations, we may want to precompute an array of horizontal (relative to the "up" vector for the screen) directions so that we only need perform this matrix multiplication once per pixel column. The size of this array of vectors is, of course, equal to the horizontal resolution of the virtual screen. The accompanying C code illustrates the construction of this array.

This array stores the cardinal horizontal directions for rays sampling the *n* columns of the virtual screen. One might then ask, how do we handle jittering? For reasonably large screen resolutions, a simple linear interpolation between adjacent cardinal directions is an adequate solu-

tion. (Linear interpolation across large angles would not be a good approximation to the proper cosine distribution, but pixels are generally of small angular size, so a linear approximation to the cosine is sufficient.) Again, this is illustrated in the accompanying C code.

Vertical Sampling

The cylindrical virtual screen just described allows unlimited horizontal field of view: One can as easily render a 1° field of view as a 720° field of view; fields of view greater than 360° yield periodic images, like wallpaper. There remain problems, however, with the vertical field of view. These include distortion at very wide fields of view, as the screen gets (relatively) too close to the eye, and a degenerate projection at 180° field of view.

The way we have chosen to obtain equal angular increments on the vertical axis is to vary the vertical increments as the tangent of the y (i.e., vertical) index of the pixel on the virtual screen. (We assume that $y = 0$ at the center of the screen.) In our scheme we construct two arrays of vectors, one for the horizontal directions and another for the vertical increments. To generalize for an arbitrary "up" vector, the latter is also an array of vectors, rather than of scalar increments. All vertical increment vectors are colinear (i.e., scalar multiples of the "up" vector), and orthogonal to all horizontal direction vectors. To get the direction vector for a ray to sample pixel (x, y) on the virtual screen then, we take the vector sum of entry x in the horizontal directions array and entry y in the vertical increments array, and normalize. (Again, see the code fragment.)

With this scheme, vertical fields of view greater than 180° yield periodic (but always right-side-up) images of the scene.

An Application: Martian Panoramas

The panoramic virtual screen was developed for a specific application: creating realistic panoramic views of Martian terrains to be viewed in a virtual reality setting. The work was undertaken at the Visualization for Planetary Exploration (VPE) Lab at NASA Ames, in Mountain View, California, where the author worked in the summer of 1991. The goal of

the development of the panoramic virtual screen model was to supplement the existing, comparatively less-realistic, real-time rendering capability for terrain height fields, with the enhanced realism and aesthetic quality available in (far from real-time!) ray-traced imagery.

The virtual reality implementation at VPE features interactive viewing of landscape panoramas, using a head-mounted display that tracks the user's movements. In this mode a large (e.g., 6,000 by 3,000 pixel) static image is loaded into video memory, and the display presents an appropriate viewport on the panoramic scene via real-time pan-and-scroll frame buffer animation. The image loaded into memory may be of arbitrary complexity, as update requires only presentation of a new viewport, as opposed to rendering an entirely new frame. Thus, both the update rate and the visual quality are generally better than with real-time rendering animation.

At the outset of this work, VPE possessed the capability of Z-buffer rendering static panoramas using available hardware rendering capabilities. These panoramic views are constructed by abutting a series of flat-screen views edge-to-edge. The net result is a kind of faceted-cylinder virtual screen. (A cylindrical projection equivalent to that presented here can be obtained by reducing the facet widths to one pixel.) As these renderings use the hardware implementation of the viewing projection, they are prone to the same kind of vertical distortion as is seen when using a standard virtual screen. We sought to recreate this cylindrical projection, with improvements to the vertical sampling, in a ray tracer. A ray tracer gives access to certain realistic effects not readily available in a hardware Z-buffer renderer, e.g., shadows, atmospheric effects, and procedural textures (Musgrave, 1990, 1991).

A result of this effort is seen in Plate 1 (see color insert), a panoramic view of the Valles Marineris on Mars. Note the extreme bow-shaped distortion of the (in reality) nearly linear, parallel valley features. This distortion is a natural and inevitable by-product of the viewing projection we have constructed. Note also that the image was not designed to be viewed in its entirety, as it is reproduced here, but rather in a virtual reality system, wherein only a viewport on relatively small area of the image is visible at any given time. The idea was to construct an image such that anywhere the user looked, they would see an appropriate (if distorted) area of the Martian environs. The distortion near the bottom of the image serves to discourage the viewer from investigating that area;
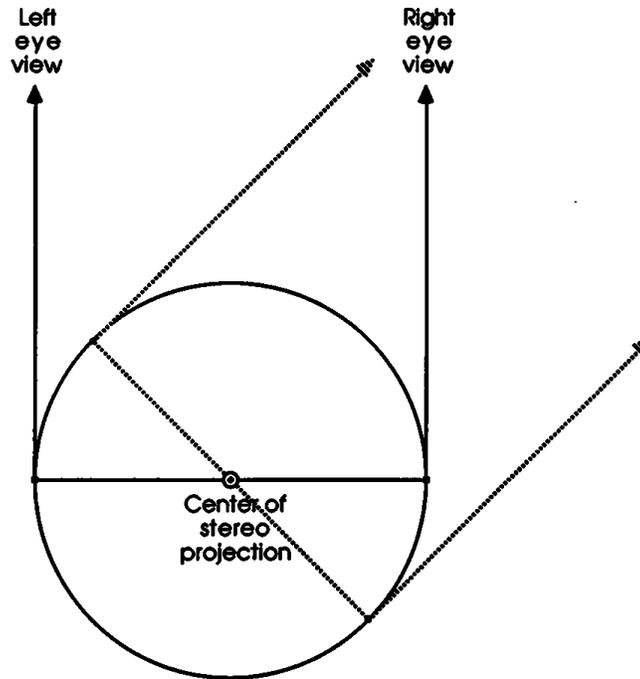
Figure 2. Eye point rotation scheme for stereo panoramas.

not altogether a bad thing, as the fixed-resolution terrain data being imaged shows little detail there, where it is closest to the eye point.

We have also implemented a method for creating stereo panoramas suggested by Lew Hitchner at VPE. In this scheme, the eye points describe a circle as the view direction scans about the virtual screen (see Fig. 2). A conventional stereo rendering using two fixed eye points will lack stereoscopy around the direction defined by the line through the two eye points, as the stereo separation goes to zero there. This rotating-eye model yields good stereoscopy over the entire 360° horizontal field. Its implementation appears in the code segment.

Conclusion

We have constructed a panoramic virtual screen for ray tracing. This projection maps the entire view-dependent celestial sphere to a rectilinear screen. Introduction of distortion is unavoidable in this sphere-to-plane

mapping; the distortion in this construction of the viewing projection is of a different character than that of a standard virtual screen, taking the form of horizontal stretching of the image as one approaches the poles of the sphere. The resulting panoramic images may be useful for interactive viewing of static imagery in a virtual reality system.

This panoramic viewing projection is interesting, as it is something that is relatively straightforward to implement in a synthetic camera, but difficult to impossible to accomplish with a real camera.

See also G2, 179.
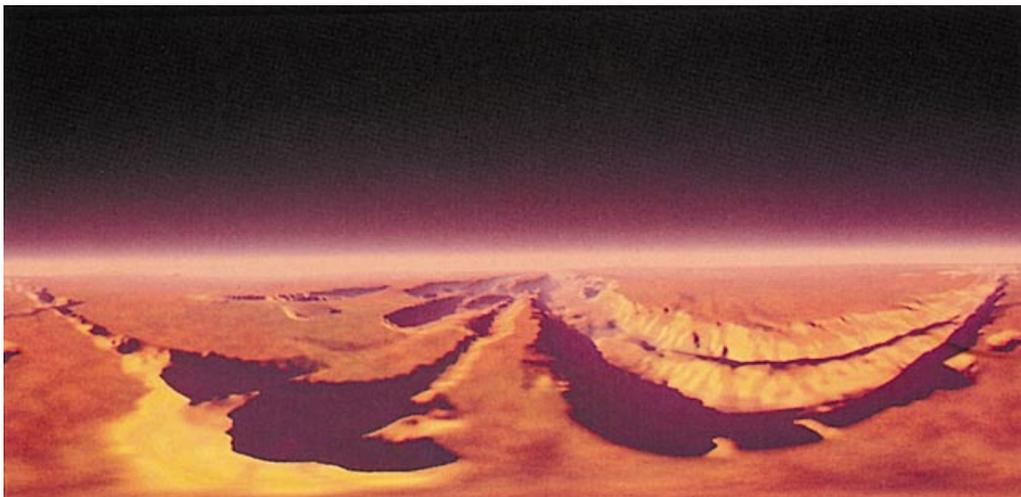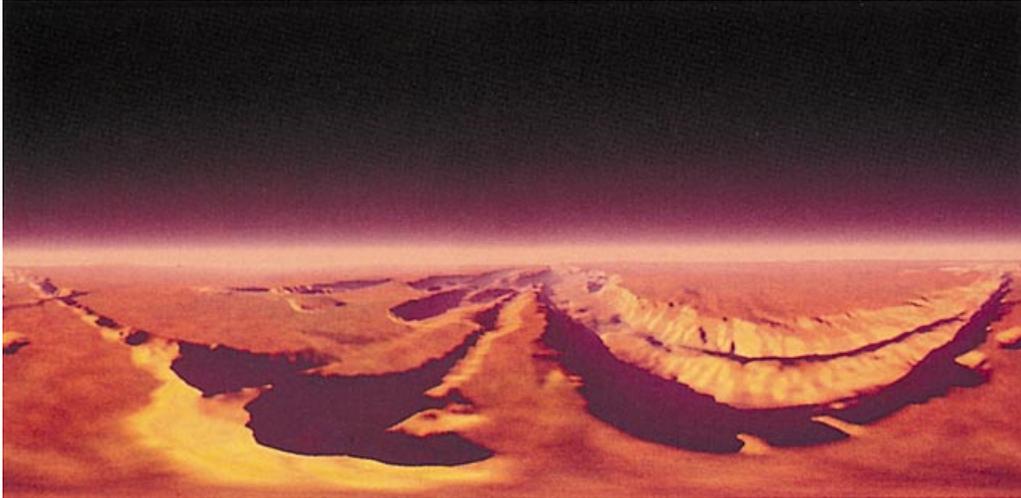
6.4 Plate 1. Valles Merineris, Mars.

VI.5
RECTANGULAR BOUNDING VOLUMES FOR POPULAR PRIMITIVES

Ben Trumbore
Cornell University
Ithaca, New York

Many efficient ray tracing algorithms require that each primitive object in an environment be bounded by an orthogonal, rectangular volume. This Gem provides derivations and source code for the generation of minimal bounding volumes for a cube, a group of polygons, a sphere, a cylinder, a cone, and a torus.

The methods described here assume that each object in an environment is created by applying a cumulative transformation matrix (CTM) to a canonical primitive object. This CTM is defined to be

$$M = \begin{bmatrix} M_{00} & M_{01} & M_{02} & 0 \\ M_{10} & M_{11} & M_{12} & 0 \\ M_{20} & M_{21} & M_{22} & 0 \\ M_{30} & M_{31} & M_{32} & 1 \end{bmatrix}.$$

To calculate these bounding volumes, components of the primitive object are transformed using a CTM. Extrema of these transformed components are then calculated. These extrema are used to determine the extents of the bounding volume.

Cube

The bounding volume for a cube is easily calculated by transforming each corner of the cube and performing simple min/max tests. In the provided C code, the cube is defined to extend from –1.0 to 1.0 in *X*, *Y*, and *Z*.

Thus, it is centered about the origin, and it has sides of length 2.0.

Polygons

Collections of polygons can be bounded in a manner similar to that for cubes. Each vertex of the polygon set is transformed and is then used to update the extrema values. If a list of unique spatial vertices is not maintained, a less efficient algorithm can perform the calculation for each vertex of every polygon in the set.

Cylinder

By transforming the circles at both ends of a cylinder and finding the extrema of those circles, a tight bounding volume can be determined. Here, a cylinder is defined as the extrusion of a circle of radius 1.0 along the Y axis from -1.0 to 1.0. The circle components are expressed functionally as

$$C_{top}(t) = (\cos(t),\ 1,\ \sin(t),\ 1), \qquad 0 \leq t \leq 2\pi,$$

$$C_{bot}(t) = (\cos(t),\ -1,\ \sin(t),\ 1).$$

The transformed circles T_{top} and T_{bot} are defined as

$$T_{top}(t) = C_{top}(t)M$$

$$= (M_{00}\cos(t) + M_{10} + M_{20}\sin(t) + M_{30},$$

$$M_{01}\cos(t) + M_{11} + M_{21}\sin(t) + M_{31},$$

$$M_{02}\cos(t) + M_{12} + M_{22}\sin(t) + M_{32});$$

$$T_{bot}(t) = C_{bot}(t)M$$

$$= (M_{00}\cos(t) - M_{10} + M_{20}\sin(t) + M_{30},$$

$$M_{01}\cos(t) - M_{11} + M_{21}\sin(t) + M_{31},$$

$$M_{02}\cos(t) - M_{12} + M_{22}\sin(t) + M_{32}).$$

The derivatives of T_{top} and T_{bot} are the same for all three dimensions. The extrema in each dimension (d) can be found by setting the derivative equal to 0.0 and solving for t:

$$0 = T_d = M_{2d}\cos(t) - M_{0d}\sin(t),$$

$$M_{2d} / M_{0d} = \sin(t) / \cos(t),$$

$$t = \arctan(M_{2d} / M_{0d})$$

By substituting these three values for t into the equations for T_{top} and T_{bot}, points that determine the extent of the bounding volume can be found.

Cone

The bounding volume of a cone is calculated using the circle component formed by its base and the point component formed by its apex. The presented algorithm effectively combines aspects of the cube and cylinder bounding volume routines. The canonical cone used here has a circular base of radius 1.0 around (0, –1, 0) in the $Y = -1.0$ plane, and an apex at (0, 1, 0).

Conic

The conic object class bounded by this routine has a circular base of radius 1.0 around (0, –1, 0) in the $Y = -1.0$ plane, and a circular top of radius r around (0,1, 0) in the $Y = 1.0$ plane. The parameter r may vary between 0.0 and 1.0. When $r = 0.0$, the conic forms a cone; where $r = 1.0$, it forms a cylinder. Separate extrema are found in each dimension for the base and the top, and they are compared to find the dimensional extrema for the entire object. While this algorithm could be used to calculate bounding volumes for cones and cylinders, the algorithms explicitly designed for those objects are more efficient.

Sphere

A bounding volume can be generated for a spheroid by finding the extrema of the surface in world coordinates. A unit sphere of radius 1.0 centered at the origin is defined as

$$S(u, \ v) \ = \ (\cos(u)\cos(v), \ \sin(v), \ \sin(u)\cos(v), \ 1),$$

$$0 \ \leq \ u \ \leq \ 2\pi \ , \ -\pi/2 \ \leq \ v \ \leq \ \pi/2.$$

Transforming this sphere by the matrix M yields T:

$$T(u, \ v) \ = \ S(u, \ v)M$$

$$= \ (M_{00}\cos(u)\cos(v) \ + \ M_{10}\sin(v) \ + \ M_{20}\sin(u)\cos(v) \ + \ M_{30},$$

$$M_{01}cos(u)cos(v) \ + \ M_{11}\sin(v) \ + \ M_{21}\sin(u)\cos(v) \ + \ M_{31},$$

$$M_{02}\cos(u)\cos(v) \ + \ M_{12}\sin(v) \ + \ M_{22}\sin(u)\cos(v) \ + \ M_{32}).$$

In order to find the u and v parameter values for the extrema on this surface, the partial derivatives of T are found:

$$T_{u} \ = \ (\cos(v)[M_{20}\cos(u) \ - \ M_{00}\sin(u)],$$

$$\cos(v)[M_{21}\cos(u) \ - \ M_{01}\sin(u)],$$

$$\cos(v)[M_{22}\cos(u) \ - \ M_{02}\sin(u)]);$$

$$T_{v} \ = \ (M_{10}\cos(v) \ - \ \sin(v)[M_{00}\cos(u) \ + \ M_{20}\sin(u)],$$

$$M_{11}\cos(v) \ - \ \sin(v)[M_{01}\cos(u) \ + \ M_{21}\sin(u)],$$

$$M_{12}\cos(v) \ - \ \sin(v)[M_{02}\cos(u) \ + \ M_{22}\sin(u)]).$$

The extrema in each dimension are then found by setting each pair of

partial derivatives equal to 0.0 and solving:

$$0 = T_{ud} = \cos(v)[M_{2d}\cos(u) - M_{0d}\sin(u)],$$

$$M_{2d}/M_{0d} = \sin(u)/\cos(u),$$

$$u = \arctan(M_{2d}/M_{0d});$$

$$0 = T_{vd} = M_{1d}\cos(v) - \sin(v)[M_{0d}\cos(u) + M_{2d}\sin(u)],$$

$$\sin(v)/\cos(v) = M_{1d}/(M_{0d}\cos(u) + M_{2d}\sin(u)),$$

$$v = \arctan(M_{1d}/(M_{0d}\cos(u) + M_{2d}\sin(u)).$$

Once one extremum is found, π is added to u and the negative of v is used to find the other extremum in this dimension. This procedure is repeated for all three dimensions. Each of these u, v locations is then substituted into T(u, v) to find the point used to generate the bounding volume.

Torus

The torus in this derivation is defined as a vertical circle of radius r that is revolved around the Y axis. The center of the circle sweeps out a perpendicular circle of radius q in the XZ plane. The equations of such a surface and its transformed counterpart are

$$S(u, v) = ((r + q\cos(v))\cos(u), \; q\sin(v), \; (r + q\cos(v))\sin(u), \; 1),$$

$$T(u, v) = S(u, v)M.$$

The u and v parameter values for the extrema in each dimension are found by setting the partial derivatives equal to 0.0 and solving simultaneously. In fact, the solution values for these partial derivatives are found using the same equations as were used for spheres. The parametric coordinates are then substituted into T to find the extrema in each dimension.

Alternatively, the *u, v* coordinates can be found using a circular component of the torus. To do this, transform the horizontal circle C of the torus by M to give T_c:

$$C(u) = (r\cos(u), \ 0, \ r\sin(u)),$$

$$T_c(u) = C(u)M.$$

At each of the dimensional extrema of the circle T_c, q can be added or subtracted to find points that will determine the rectangular bounding volume.

See also G1, 308; G1, 548; G3, F.6.

VI.6
A LINEAR-TIME SIMPLE BOUNDING VOLUME ALGORITHM

Xiaolin Wu
University of Western Ontario
London, Ontario, Canada

Three-dimensional bounding volume is a ubiquitous tool of accelerating ray tracing and visibility testing (Arvo and Kirk, 1989; Foley *et al.*, 1990). The two most common types of bounding volume are box and sphere. This is because the intersections between rays and boxes or rays and spheres involve very simple computations. To have the optimal performance of the bounding volumes we desire the smallest bounding volumes. Namely, given N points in 3-D space, we want to pack them into a box or a sphere of the smallest possible volume. The smallest bounding volume has received considerable attention in the computational geometry community (Megiddo, 1984; Preparata and Shamos, 1985). The $O(N)$ algorithm for smallest bounding sphere (Megiddo, 1984) and $O(N \log N)$ algorithm for the three-dimensional Voronoi diagram (Preparata and Shamos, 1985) exist, and they are both optimal. However, these algorithms require mathematical sophistication to understand, their implementation is not easy, and furthermore the constant before the O notation is considerable. In practice, near-optimal bounding boxes or spheres with slightly larger volumes will perform just as well as the optimal ones in terms of avoiding unnecessary ray-object intersection examinations. Therefore, simple and fast approximation algorithms for smallest bounding volumes are highly desirable for practitioners. This Gem introduces a simple algorithm to compute near-optimal bounding volumes enclosing N points in $O(N)$ time.

A naive bounding volume algorithm will be as follows. We scan the input point set S = {(x_i, y_i, z_i): $0 \leq i < N$}, to find their x, y, and z extents: $[x_{min}, x_{max}]$, $[y_{min}, y_{max}]$, and $[z_{min}, z_{max}]$. Immediately we have an orthogonal bounding box enclosed by six planes $x = x_{min}$, $x = x_{max}$,
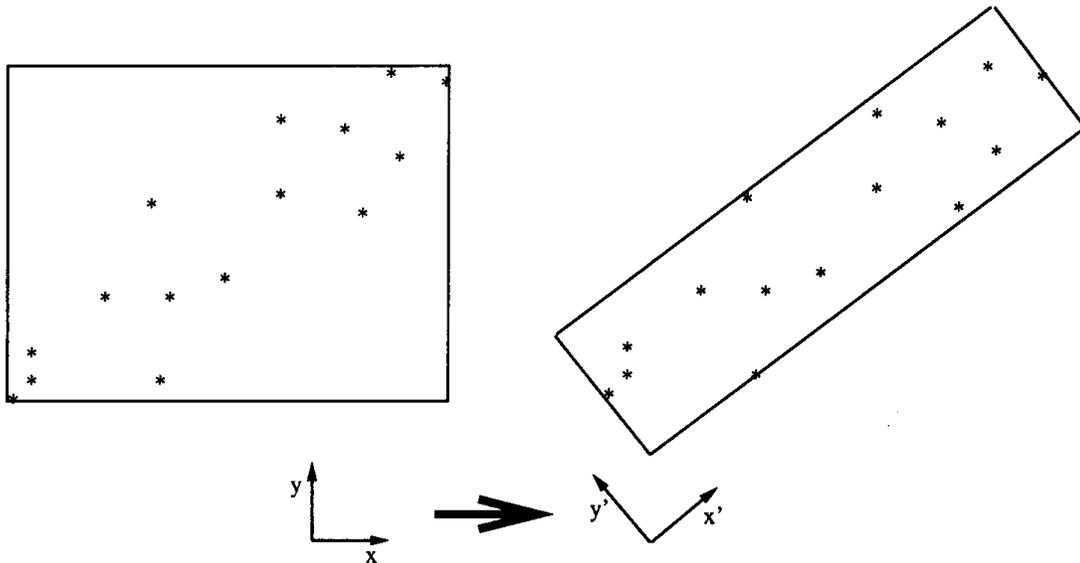
Figure 1. The worst case for the naive bounding volume algorithm.

$y = y_{min}$ $y = y_{max}$, $z = z_{min}$, $z = z_{max}$. A bounding sphere can also be constructed by setting the sphere center to be the center of the preceding box, and its radius to be the distance from the center to a vertex of the box. But this will not in general give us a good bounding box or sphere. The worst case for the preceding naïve approach is when the data points are stretched along the diagonal of the orthogonal bounding box. A two-dimensional illustration of this adverse situation is given in Fig. 1a. Intuitively, the cure to the problem is clear. The orientations of the bounding halfplanes should be made adaptive to, the data distribution. What we need is a linear transformation to rotate the old coordinate system in which the input data are given to a new coordinate system. The new coordinate system will have one of its axes coincide with the principal axis of the point set in which the data spread the most, or have the maximum variance in terms of multivariate analysis. In this new coordinate system it now makes sense to use the simple orthogonal bounding box as a good approximation to the smallest bounding box. The effect of the said linear transformation is demonstrated in Fig. 1b for two-dimensional space. It should not be difficult to visualize the same effect in three-dimensional space.

The remaining key question is how to compute the desired new coordinate system. Fortunately, the answer readily comes from a well-known technique called principal component analysis (Manly, 1986). The computational procedure for principal component analysis can be found in Manly (1986) or any other multivariate analysis books. But we will sketch it for completeness.

Step 1*.* Compute the covariance matrix of the data set S:

$$\mathbf{C} = \begin{pmatrix} C_{x,\,x} & C_{x,\,y} & C_{x,\,z} \\ C_{y,\,x} & C_{y,\,y} & C_{y,\,z} \\ C_{z,\,x} & C_{z,\,y} & C_{z,\,z} \end{pmatrix}, \tag{1}$$

in which the nine covariances are

$$c_{x,\,z} = c_{z,\,x} = \frac{\sum_{0 \le i < N} c_x c_z}{N} - \mu_x \mu_z,$$

$$c_{y,\,y} = \frac{\sum_{0 \le i < N} y_i^2}{N} - \mu_y^2,$$

$$c_{z,\,z} = \frac{\sum_{0 \le i < N} z_i^2}{N} - \mu_z^2,$$

$$c_{x,\,y} = c_{y,\,x} = \frac{\sum_{0 \le i < N} c_x c_y}{N} - \mu_x \mu_y,$$

$$c_{x,\,z} = c_{z,\,x} = \frac{\sum_{0 \le i < N} c_x c_z}{N} - \mu_x \mu_z,$$

$$c_{y,\,z} = c_{z,\,y} = \frac{\sum_{0 \le i < N} c_y c_z}{N} - \mu_y \mu_z, \tag{2}$$

where

$$\mu_x = \frac{\sum_{0 \leq i < N} x_i}{N}, \qquad \mu_y = \frac{\sum_{0 \leq i < N} y_i}{N}, \qquad \mu_z = \frac{\sum_{0 \leq i < N} z_i}{N}, \tag{3}$$

are the centroid coordinates of the data set S.

Step 2. Compute the eigenvalues λ_1, λ_2, λ_3, and the corresponding eigenvectors $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$, of the 3×3 matrix C. A C program for performing this eigenvector transform can be found in a reference book *Numerical Recipes in C* (Press *et al.*, 1988).

Step 3. The three orthogonal eigenvectors $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ define the desired new coordinate system. The relation between the old coordinates x_i, y_i, z_i) and the new coordinates (x_i', y_i', z_i') is given by the linear transform

$$\begin{pmatrix} x_i' \\ y_i' \\ z_i' \end{pmatrix} = \begin{pmatrix} u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \\ w_1 & w_2 & w_3 \end{pmatrix} \begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix}. \tag{4}$$

After the eigenvector transform we can compute the orthogonal bounding box of the points (x_i', y_i', z_i'), $0 \leq i < N$.

Now consider the problem of finding the smallest bounding sphere. In *Graphic Gems I*, Ritter (1990) proposed a simple approximation scheme. His technique will first scan the N input points $p_i = (x_i, y_i, z_i)$, $0 \leq i < N$, to find the extreme points in the x, y, and z directions: $p_{x\ min}$, $p_{x\ max}$, $p_{y\ min}$, $p_{y\ max}$, $p_{z\ min}$, $p_{z\ max}$. Then we tentatively construct a sphere with $\max\{d(p_{x\ min}, p_{x\ max}), d(p_{y\ min}, p_{y\ max}), d(p_{z\ min}, p_{z\ max})\}$ being its diameter, where $d(\cdot, \cdot)$ denotes the Euclidean distance. The N points will be scanned one more time to test if they are contained in the tentative sphere. If the test fails for the current point, the tentative sphere will be expanded to just cover this point.

Unfortunately, the preceding technique could fail to find a satisfactory approximation to the smallest bounding sphere when the data distribution is not orthogonal to the x, y, z axes, a common occurrence in practice. To see this defect, refer to Fig. 2. Sphere 1 is the initial tentative sphere, and sphere 2 is the bounding sphere by expanding the tentative sphere if
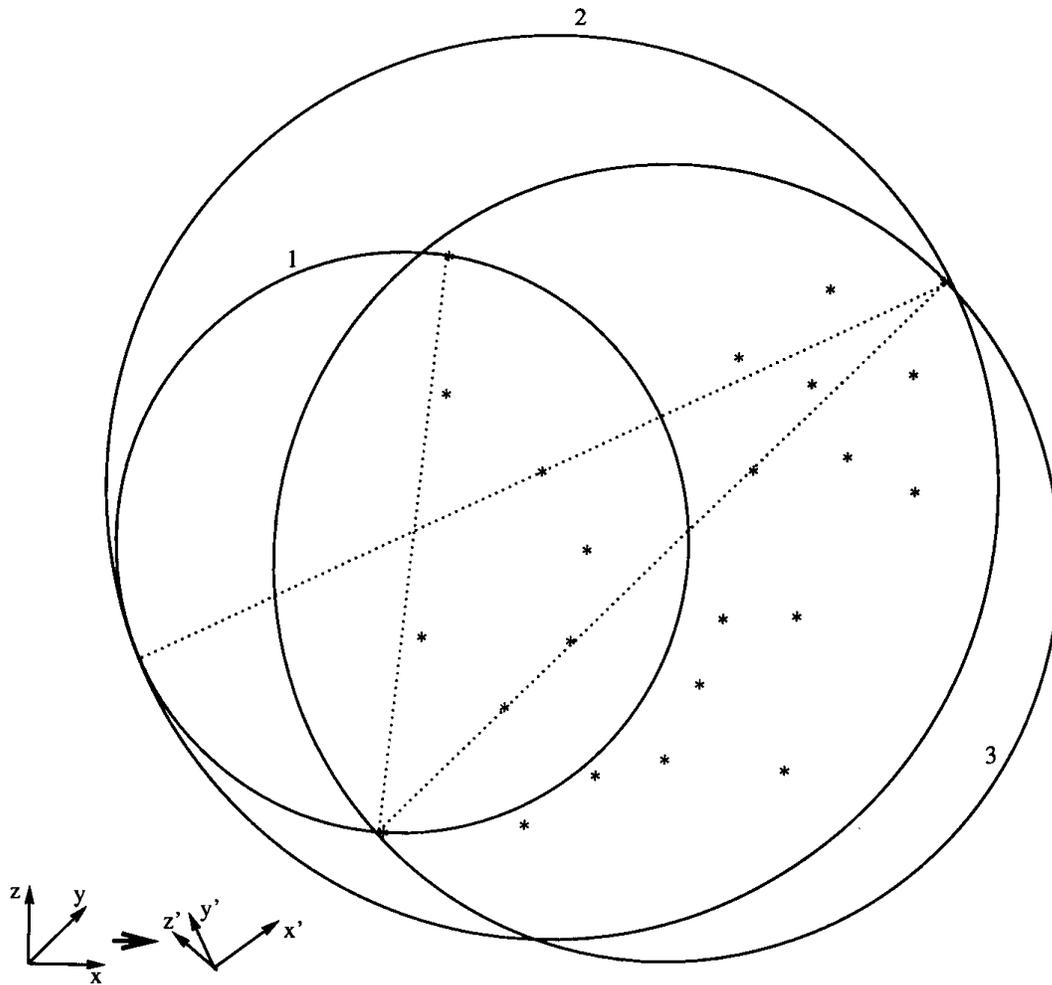
Figure 2. A situation in which the simple bounding sphere algorithm fails without principal component analysis.

Ritter's technique is used. As we can observe from the figure, the majority of input points fall outside of the tentative sphere, resulting in a poor approximation. Sphere 3 is the smallest bounding sphere. The differences between the approximation (sphere 2) and the true solution (sphere 3) in terms of their sizes and locations are significant. Note that the given point distribution is by no means degenerated. However, if the eigenvector transform is performed to find the principal orientation of the data points, then the problem associated with Ritter's simple bounding sphere technique can be fixed. Indeed, if Ritter's technique is applied in the new

x′y′z′ coordinate system rather than directly in the old *xyz* coordinate system, we will precisely obtain the smallest bounding sphere in the situation of Fig. 2. For a robust bounding sphere algorithm, we strongly recommend that the reader analyze the data first using eigenvector transform before using Ritter's technique.

Finally, the time complexity issue. How fast is the eigenvector transform? The cost of computing covariance matrix **C** at step 1 dominates that of the whole process. The computation clearly takes $O(N)$ time. Furthermore, if N is sufficiently large, we may only randomly sample a fraction of N points to approximate **C**, achieving even sublinear time. The numerical computation of the eigenvectors for the positive definite 3×3 matrix **C** is fast and robust.

See also G1, 301; G3, F.5.

VI.7
PHYSICALLY CORRECT DIRECT LIGHTING FOR DISTRIBUTION RAY TRACING

Changyaw Wang
Indiana University
Bloomington, Indiana

The direct lighting component in distribution ray tracing is calculated by performing a numerical integration accounting for all potentially visible luminaires.[1] In this Gem, a physically correct method for calculating the contribution from spherical and triangular luminaires is presented.

Suppose a point x is illuminated by a luminaire S and is viewed from direction ψ (see Fig. 1). The spectral radiance (essentially brightness) of the point at a particular wavelength is expressed by the rendering equation:

$$L(x,\ \psi)\ =\ \int_{x'\in S}\ \ g(x,\ x')\rho(x,\ \psi,\ \psi')L_e(x',\ \psi')\cos\theta\frac{dA'\cos\theta'}{\left\|x'-x\right\|^2},\tag{8}$$

where S is a luminaire; $g(x,\ x')$ is zero if there is an obstruction between x and x', and one otherwise; y' is the direction from x to x'; $r(x,\ y,\ y')$ is the BRDF; $L_e(x',\ y')$ is the radiance in power/area—solid angle; q is the angle between y' and the surface normal at x; q' is the angle between y' and the surface normal at x'; and dA' is the differential area of x'.

If a RGB color model is used, Eq. (1) should be evaluated for each color component. If a spectral model is used, then Eq. (1) should be evaluated at each wavelength.

For an environment with multiple luminaires, there is an integral like that of Eq. (1) for each luminaire, and the direct lighting is the sum of these integrals. For the rest of this discussion, one luminaire will be assumed.

---

[1]A luminaire is an object that produces light, e.g. the sun.
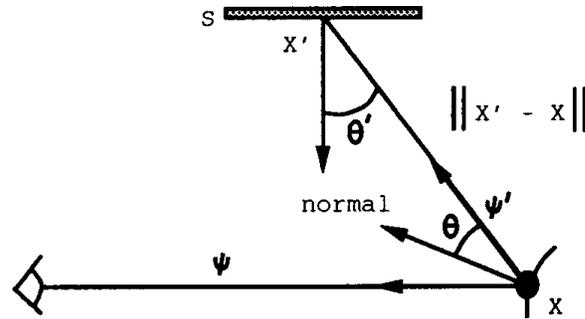
Figure 1. Direct lighting for *x*.

A simple and common way to estimate the integral in Eq. (1) is to first define L_{max} to be the approximate value of the integral if *g* is always 1 (i.e., there is no shadowing). Then we use the approximation

$$L(x, \psi) = L_{max} \times \text{(fraction of } S \text{ that is visible to } x)$$

To estimate the fraction, we send a shadow ray to a random point on *S*, and if it gets blocked before it reaches *S* we let $L(x, \psi) = 0$. If the shadow ray is not blocked, we get $L(x, \psi) = L_{max}$ This approximation can work well if solid angle is small, i.e., the area of *S* is small compared to *distance*(x, x′).

If the physical accuracy is required, then a more rigorous Monte Carlc integration can be used to get an unbiased estimate for Eq. (1). Given a set of *N* points x_i, we can estimate any integral with

$$\int_{x \in \Omega} f(x) d\mu(x) \approx \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)},$$

where $p(x)$ is any probability density function that is positive when $f(x)$ is nonzero; x_i is a random variable with density $p(x)$.

In classic distribution ray tracing, *N* is 1, so one shadow ray is sent to each luminaire, and Monte Carlo integration for (1) gives

$$L(x, \psi) \approx g(x, x') \rho(x, \psi, \psi') L_e(x', \psi') \cos\theta \frac{\cos\theta'}{p(x') \|x' - x\|^2}, \tag{2}$$

where *x′* is a point on *S* according to *p*.

   This means that we can carry out the calculation by the following procedure:

1.   Choose a probability density function p;

2.   find x' on S according to p;

3.   compute $g(x, x')$ by sending a shadow ray from x to x' ;

4.   if $g = 1$, then evaluate $L(x, \psi)$ by Eq. (2); otherwise, set $L(x, \psi) = 0$.

   The only problems left are how to choose p and how to choose x' with respect to p. The easiest way is to let $p(x') = 1/(\text{area of } S)$, i.e., uniformly choose a random point on S. To uniformly choose a point on a sphere, we can transform two random numbers r_1 and r_2 from 0 to 1 to a point on a unit sphere, $(1, \theta, \phi) = (1, \cos^{-1}[1 - 2r_1], 2\pi r_2)$ in spherical coordinates, then transfer the point to Cartesian coordinates: $(x, y, z) = (\sin \theta \cos \phi, \sin \theta \sin \phi, \cos \theta)$. Of course, this will generate sample points on the back of the sphere that do not contribute, and we will have a high variance in our estimate. We are better off intelligently choosing p by *importance sampling.*
   If the luminaire is diffuse $(L_e(x,\cdot y) = \text{constant})$, then the radiance from the luminaire coming into any direction inside the solid angle should have approximately the same contribution. According to the principle of importance sampling, choosing a sample point on the luminaire to the solid angle, $p(x'') = 1/(\text{solid angle})$, where x'' is a point on the unit sphere within the solid angle, should be the most efficient sampling method. Also, since the probability density function we need is for a point x' on S, after we find out x' with the help of x'', we need to scale $p(x'')$ to get $p(x')$. Suppose dA is the differential area. On surface S', the solid angle covered by dA with respect to x is $dA \cos \theta'/\text{distance}^2(x, x')$, where x' is a point inside dA on S' and θ' is the angle between the vector from x' to x and the normal on S' at x'. Similarly, on surface S'', the solid angle is $dA \cos \theta''/\text{distance}^2(x, x'')$. Since the probability of choosing a sample point is proportional to the covered solid angle, $p(x')/p(x'') = \text{solid angle of } dA \text{ on } S'/\text{solid angle of } dA \text{ on } S''$. Therefore, $p(x')$ can be expressed

$$p(x') = (x'')\frac{\text{distance}^2(x, x'')\cos\theta'}{\text{distance}^2(x, x')\cos\theta''}.$$

Sampling for Spherical Luminaire

As it turned out, in Fig. 2, we can sample according to the solid angle of a spherical luminaire. First, find out the solid angle, then uniformly pick a point on the unit sphere within the solid angle, i.e., a unit vector within the solid angle, and then scale the probability. All vectors are assulned lo be unit vectors.

1. Compute θ_{max}:

$$\theta_{max} = \sin^{-1} \frac{radius}{distance(x,\ center)}.$$

2. Suppose x is the origin and the center, as well as the solid angle, has been rotated to Z axis. Then we can uniformly pick a unit vector $(\psi = x'' - x),(1,\theta,\phi)$ in the spherical coordinate system, within the
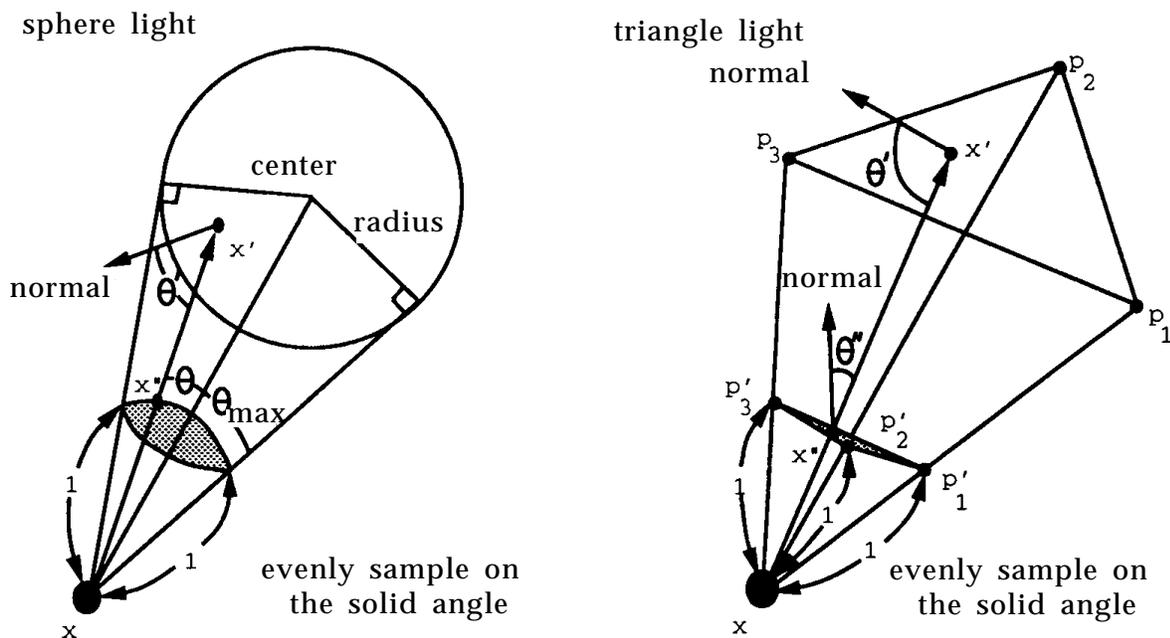
Figure 2. Sampling ideas for spherical luminaire and triangular luminaire.

rotated solid angle. r_1 and r_2 are random numbers from 0 to 1:

$$(\theta, \phi) = (\cos^{-1}(1 - r_1 + r_1\cos\theta_{max}), \ 2\pi r_2).$$

3.   Find ψ in the Cartesian coordinate system:

$$\psi = (\sin\theta \ \cos\phi, \ \sin\theta \ \sin\phi, \ \cos\theta).$$

4.   Find a new Cartesian coordinate system (X', Y', Z') in terms of the original Cartesian coordinate system (X, Y, Z), where Z' is the unit vector from x to center. v is an arbitrary vector that does not agree with $\pm Z'$. Notice we are using the right-handed coordinate system.

$$X' = Z' \times v,$$

$$Y' = Z' \times X'.$$

5.   Compute the rotation matrix M that rotates (X, Y, Z) to (X', Y', Z'):

$$M = \begin{bmatrix} X'.x & Y'.x & Z'.x \\ X'.y & Y'.y & Z'.y \\ X'.z & Y'.z & Z'.z \end{bmatrix}.$$

6.   Rotate ψ to the real solid angle from the Cartesian coordinates of step 2:

$$\psi = M\psi.$$

7.   Find x' on S. Refer to *Graphics Gems I*, page 388 (Glassner, 1990).

8.   Return ψ and $p(x')$:

$$p(x') = p(x'')\frac{\cos\theta'}{\text{distance}^2(x, \ x')} = \frac{\cos\theta'}{2\pi(1 - \cos\theta_{max})\text{distance}^2(x, \ x')}.$$

Sampling for Triangular Luminaire

Because of the difficulty in computing the solid angle of a triangle an finding a sample direction within the solid angle, we approximate the solid angle of a triangular luminaire with another triangle that is a approximation of the projection of the luminaire on the unit sphere centered at x (see Fig. 2).

1. Find p_1', p_2', p_3':

$$p'_i = x + \frac{p_i - x}{\text{distance}(x, p_i)}, \qquad i = 1, 2, 3.$$

2. Find x'', a random point on triangle p_1', p_2', p_3', in uv coordinates:

$$(u, v) = \left(1 - \sqrt{1 - r_1}, \; r_2\sqrt{1 - r_1}\right).$$

3. Find x'' in the Cartesian coordinate system:

$$x'' = p'_1 + u\left(p'_2 - p'_1\right) + v\left(p'_3 - p'_1\right).$$

4. Find ψ:

$$\psi = x'' - x.$$

5. Find x'. Refer to *Graphics Gems I,* pp. 390 and 394 (Glassner, 1990).

6. Return ψ and $p(x')$:

$$p(x') = p(x'')\frac{\text{distance}^2(x, x'')\cos\theta'}{\text{distance}^2(x, x')\cos\theta''} = \frac{\text{distance}^2(x, x'')\cos\theta'}{\text{area}_{p_1p_2p_3}\text{distance}^2(x, x')\cos\theta''}.$$

   For a rectangular luminaire, the solid angle is very difficult to compute, and the approximation method used for a triangular luminaire is not suitable. But to estimate direct lighting for a rectangular luminaire, we

can use the method that uniformly picks a random point on the luminaire, or divide the luminaire into two triangular luminaires.

For an environment with complicated lighting, such as a nondiffuse luminaire, the shadow rays should be sent out more toward the brighter region and the distribution of choosing shadow rays is not even within the solid angle. However, the Monte Carlo integration still guarantees an unbiased estimate, which means that the uniform sampling method and solid angle method will converge to a correct result, although many samples may be needed.

The definition of radiance can be found in an ANSI report (American National Standard Institute, 1986). Distribution ray tracing originated in Cook *et al.* (1984). The rendering equation was first presented in *Computer Graphics* in Kajiya (1986) and Immel *et al.* (1986). Monte Carlo integration is explored in depth in Screider (1966). A different way to sample a triangular luminaire can be found in Lee *et al.* (1985).

See also G3, F.8.

VI.8
HEMISPHERICAL PROJECTION
OF A TRIANGLE

Buming Bian
UT System Center for High Performance Computing
Austin, Texas

Form factor calculation is very important in order to obtain an accurate simulation of a real scene. The form factor is calculated analytically in terms of the area of a hemispherical projection of a planar patch; the triangle is used herein as an example. The intensity of the receiving triangle, due to a luminous planar triangle in space, is proportional to the area of the hemispherical projection of the luminous triangle. Given two 3-D triangles, homogeneous transformations are applied to move the coordinate system so that the origin is located on P, the center of the receiving triangle, and the xy plane is the triangle plane. The luminous triangle is transformed, and its vertices are located at P_1, P_2, and P_3. A hemisphere is placed with its center at P and its equatorial plane on the xy plane. To simplify the problem, consider only one line in the triangle. Points P_1 and P_2 and the line connecting them are selected to show the calculation procedures; the extension to the triangle is straightforward. P_1, and P_2 are projected on to the unit hemisphere; two unit vectors $\mathbf{P}_1$ and $\mathbf{P}_2$ are formed, pointing from P to P_1 and P_2. The endpoints of the two vectors are located on a great circle (since the plane containing them passes through the center of the hemisphere), and the normal $\mathbf{C}$ of this circles plane is

$$\mathbf{C} = \mathbf{P}_1 \times \mathbf{P}_2 = \{\mathbf{C}_x, \mathbf{C}_y, \mathbf{C}_z\}.$$

The great circle is projected onto the equatorial plane of the hemisphere forming a great ellipse; the arc terminated by $\mathbf{P}_1$ and $\mathbf{P}_2$ of the great circle has a corresponding arc of a great ellipse. The major axis of
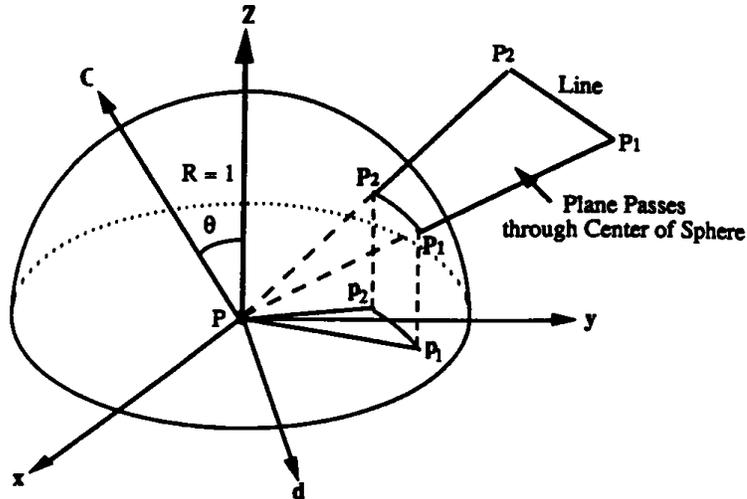
Figure 1. Projection of a line on a hemisphere.

the ellipse has unit length and points in the direction of the vector

$$\mathbf{d} = \mathbf{Z} \times \mathbf{C} = \{\mathbf{d}_x, \ \mathbf{d}_y, \ \mathbf{0}\},$$

which is perpendicular to the z direction; therefore, it is on the xy plane (the equatorial plane). The minor axis, $\mathbf{c}$, is the projection of C to the equatorial plane:

$$\mathbf{c} = (\mathbf{C}_x, \ \mathbf{C}_y, \ \mathbf{0}\}.$$

The vectors $\mathbf{C}$ and $\mathbf{d}$ define the plane of the ellipse. Using $\mathbf{p}_1$ and $\mathbf{p}_2$ to represent the projection of $\mathbf{P}_1$ and $\mathbf{P}_2$ on the equatorial plane, we have two 2-D vectors:

$$\mathbf{p}_1 = \{\mathbf{P}_{1x}, \ \mathbf{P}_{1y}\},$$

$$\mathbf{p}_2 = \{\mathbf{P}_{2x}, \ \mathbf{P}_{2y}\}.$$

A rotation matrix $\mathbf{M}$ is constructed to make the ellipse major axis $\mathbf{d}$ coincide with the x axis:

$$\mathbf{M} = \begin{bmatrix} \mathbf{d}_x & \mathbf{d}_y \\ -\mathbf{d}_y & \mathbf{d}_x \end{bmatrix}.$$
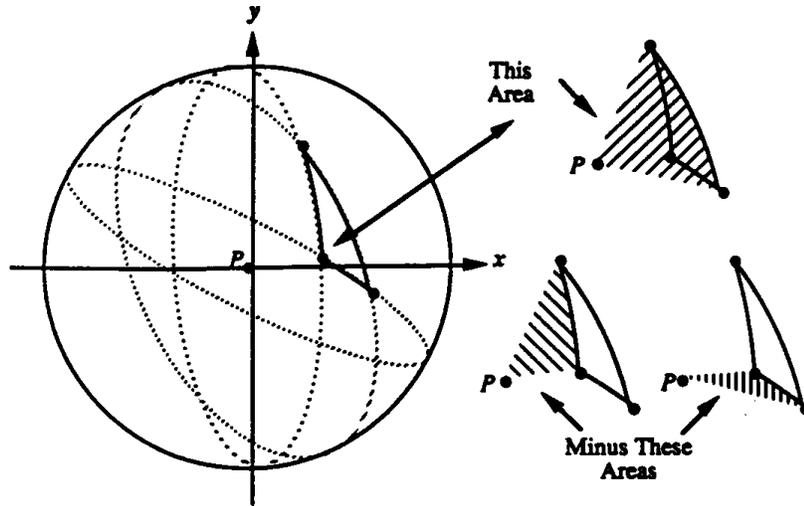
Figure 2. Projection of spherical triangle on equatorial plane.

Then, by applying $\mathbf{M}$ to $\mathbf{p}_1$ and $\mathbf{p}_2$, we have

$$\mathbf{p}'_1 = \mathbf{M}\mathbf{p}_1 = \{\mathbf{p}'_{1x}, \mathbf{p}'_{1y}\},$$

$$\mathbf{p}'_2 = \mathbf{M}\mathbf{p}_2 = \{\mathbf{p}'_{2x}, \mathbf{p}'_{2y}\},$$

These are the coordinates of the projected points in the $(\mathbf{d}, \mathbf{c})$ coordinate system. The area of the ellipse sector containing the points $\mathbf{p}'_1\mathbf{p}'_2$, and the origin P, is

$$A = \frac{C_z}{2|\mathbf{C}|}(a \; \cos(\mathbf{p}'_{2x}) - a \; \cos(\mathbf{p}'_{1x})).$$

The area of the projected triangle is obtained by adding or subtracting the three areas corresponding to the three sectors formed by the three sides, in their respective great ellipses. The form factor can be expressed in a closed form:

$$F = \frac{1}{\pi}\sum_{i=1}^{3}A_i.$$

When quadrilateral patches, or polygons with even more vertices, are used, the number of area calculations is equal to the number of vertices.
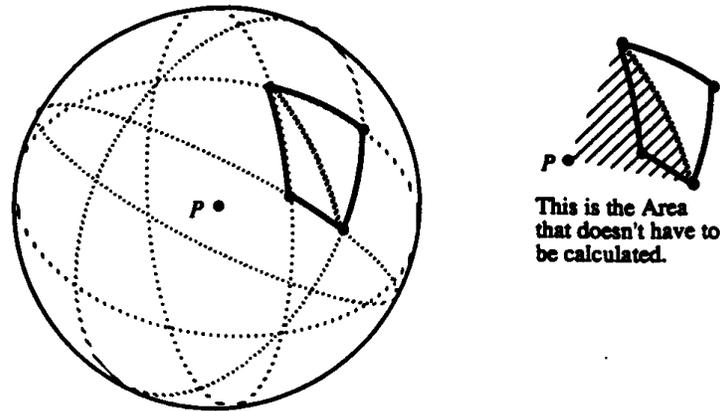
Figure 3. Equatorial projection of a spherical quadrilateral.

This can be seen by decomposing the polygon into triangles and noticing that some of the sector areas are both added and subtracted, canceling each other out. Note that this decomposition into triangles is not unique, but the number of ellipse sector area calculations is invariant with decomposition.

The equations allow exact calculation of the illuminance produced at any point by an arbitrary planar polygon with uniform luminance (Bian, 1990). Of course, since portions of a polygon lying below the equatorial plane cannot illuminate the plane (assuming it lies on an opaque surface), it might be necessary to clip the polygon to assure that it lies completely above the equatorial plane.

Compared with the hemicube projection (Cohen and Greenberg; 1985), which can perform visibility checks and estimate the form factor, hemisphere projection can not only check the visibility between patches, but also give an analytical calculation of the form factor. The analytical approach (Baum *et al.,* 1989) can calculate the form factor analytically, but it has to use some other methods such as hemicube projection to do the visibility check.

See also G.3, F.7; G.3, F.11.

VI.9
LINEAR RADIOSITY APPROXIMATION USING VERTEX-TO-VERTEX FORM FACTORS

Nelson L. Max and Michael J. Allison
Lawrence Livermore National Laboratory
Livermore, California

Most current radiosity algorithms assume that the radiosity is constant on each patch. In his U.C. Berkeley Ph.D. thesis, Paul Heckbert suggested approximating the radiosity variation with higher-order polynomial finite elements. He demonstrated in 2-D "flatland" that this gives better approximations with fewer elements, and less computation time. We have made a first step toward extending this approach to 3-D, by using linear radiosity variations across triangular patches. Such linear variation is usually assumed in the rendering phase after the solution has converged, but not in the radiosity solution itself.

Linear radiosity variation can be incorporated into both ray-tracing and hemicube methods. Ray-tracing methods for progressive radiosity (Wallace *et al.*, 1989) shoot rays from a patch to a vertex and can use linearly interpolated radiosities at sample points on the patch to find the energy transported along each ray. We used a hemicube algorithm (Cohen and Greenberg, 1985), taking advantage of hardware shading features to do the necessary interpolation.

In the finite element conceptual framework, the radiosity variation is approximated by a linear combination of basis functions or "shape functions". (See Burnett, 1987.) An optimization criterion is then used to specify a set of equations to solve for the coefficients in this linear combination which in our case are the vertex radiosities. We use the "point collocation" criterion (Burnett, 1987), requiring a consistent energy balance at the vertices.

All polygons are divided into triangles, so that the vertex radiosities specify a linear radiosity variation on each triangle. The basis function f_i corresponding to a vertex P_i is the piecewise linear function that is 1 on
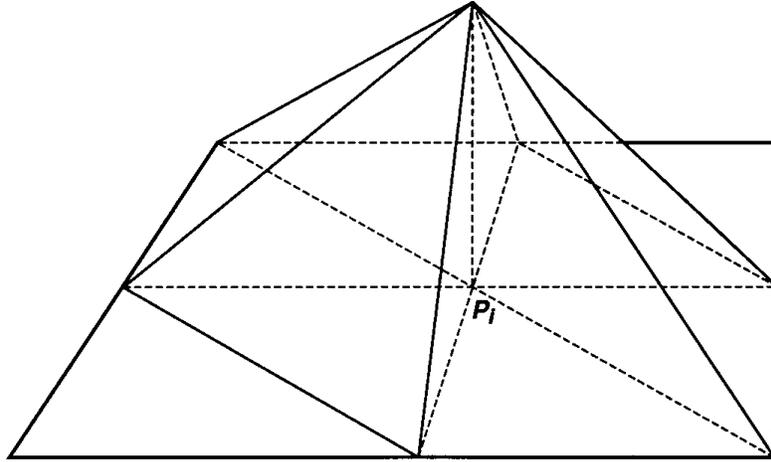
Figure 1. Graph of a basis function f_i taking the value 1 on the vertex P_i, and 0 on the other vertices. It slopes linearly from 1 to 0 along the six triangles having P_i as a vertex, and is 0 on all other triangles.

P_i and 0 on all other vertices: a "tent" with a single tent pole at P_i, as shown in Fig. 1. (If a vertex lies on the boundary between two different surface types or orientations, there must be a separate copy of the vertex for each surface type, to permit different radiosities, and parts of each "tent" will be absent.)

If B_i is the diffuse radiosity at P_i, then the linear combination $B(Q) = \Sigma_i B_i f_i(Q)$ *is* linear on each triangle and takes the value B_i at P_i. It will be our radiosity approximation.

The energy balance equation for vertex P_i states that

$$B_i = E_i + p_i I(P_i),$$

where E_i is the radiosity emitted by P_i (if it lies on a light source), P_i is the diffuse reflectivity of the surface at P_i, and $I(P_i)$ *is* the irradiance at P_i coming in from other surfaces in the scene. The values for E_i are specified in the input, and our formulation allows linear variation of intensity across light source triangles.

The irradiance $I(P_i)$ is defined by integration over a hemisphere H above the surface:

$$I(P_i) = (1/\pi)\int_H B(Q)\cos\,\theta\,d\omega$$

Here Q is the point visible in direction ω from P_i, and θ is the angle between the direction ω and the surface normal at P_i. The $1/\pi$ factor converts the radiosity $B(Q)$ to an intensity, and the Lambert factor $\cos\theta$ converts the intensity to irradiance.

Replacing $B(Q)$ by our linear combination,

$$
\begin{aligned}
I(P_i) &= (1/\pi)\int_H \sum_j B_i f_j(Q)\cos\theta \, d\omega \\
&= \sum_j B_j \mathbf{F}_{ij},
\end{aligned}
$$

where

$$
\mathbf{F}_{ij} = (1/\pi)\int_H f_j(Q)\cos\theta \, d\omega
$$

is a vertex-to-vertex form factor.

Thus the energy balance criterion gives the system of linear equations

$$
B_i = E_i + \rho_i \sum_j B_j \mathbf{F}_{ij},
$$

which can be solved for the unknown radiosities B_i using Gauss-Seidel iteration. If E_i and P_i vary with wavelength, these equations are solved separately for each wavelength band.

In the hemicube formulation, the integration is replaced by a summation over pixels k in a hemicube above the vertex P_i:

$$
\begin{aligned}
I(P_i) &= \sum_k B(Q_k)\Delta(k) \\
&= \sum_k \sum_j B_j f_j,(Q_k),\Delta(k).
\end{aligned}
$$

Here Q_k is the surface point visible in hemicube pixel k, and $\Delta(k)$ is the "Δ form factor" from Cohen and Greenberg (1985), which contains the $1/\pi$, the Lambert cosine factor, and the solid angle of pixel k. Thus,

the hemicube approximation to the vertex-to-vertex form factor is

$$\mathbf{F}_{ij} = \sum_k f_j(Q_k)\Delta(k).$$

By traversing the hemicube, the appropriate $\mathbf{F}_{ij}$ can be incremented for each pixel k. Note that $f_j(Q_k)$ will be nonzero only if j is one of the three vertex indices $I1$, $I2$, and $I3$ of the triangle T_l containing the point Q_k. Inside this triangle,

$$f_{I1}(Q) + f_{I2}(Q) + f_{I3}(Q) = 1,$$

since this sum is a linear function that is 1 on each vertex. The index l for the triangle T_l is stored in the ItemBuffer entry for pixel k, and the vertex indices $I1(l)$, $I2(l)$, and $I3(l)$ can be found from the vertex pointers for triangle T_l in the geometry model.

On our Personal Iris™, we used the gouraud shading hardware to do the linear interpolation of the $f_i(Q)$. In triangle T_l, we colored vertex $I1(l)$ pure maximum-intensity red, vertex $I2(l)$ pure green, and vertex $I3(l)$ pure black. We then combined the blue and "alpha" channels into one 16 bit ItemBuffer, to store the triangle index l. (We thank Paul Heckbert for suggesting that we use the identity $f_{I3} = 1 - f_{I1} - f_{I2}$ to free up the blue channel, so that we could do both the f_i interpolation and the ItemBuffer in one pass.)

For each vertex P_i, we started by setting $\mathbf{F}_{ij} = 0$ for all j. We then rendered all triangles in the model into the hemicube buffers, as described above, using the hardware Z-buffer and transformation pipeline. We read back the Red, Green, and ItemBuffers, respectively, into large contiguous memory arrays that included all five full and partial faces of the hemicube. (We had precomputed a corresponding large contiguous array of Δ form factors.) Then, we did the following loop:

```
For all pixels k in the large arrays
    l = ItemBuffer(k)
    j = I1(l)
    F_ij = F_ij + RedBuffer(k)* Δ(k)
    j = I2(l)
    F_ij = F_ij + GreenBuffer(k)* Δ(k)
    j = I3(l)
    F_ij = F_ij + (255 - RedBuffer(k) - GreenBuffer(k))* Δ(k)
```

When hemicubes are placed at a vertex instead of at a face center, problems arise at a concave corner, because faces adjacent to the vertex project onto the hemicube as lines, instead of as the area appropriate for computing the energy balance. For example, suppose V and W are the floor and wall copies of a vertex at the edge of the floor of a room. In order for the final gouraud shading on the floor to be correct near V, the radiosity at V should be the limit of the radiosities at points Q on the floor near V. Consider hemicubes centered at Q. As Q approaches V, the wall polygons cover half of the hemicube in the limit, although they project to a line at V itself. So to get the appropriate contribution for the wall, the center of the hemicube for V must be moved a small amount out into the interior of the room.

Also, as Q approaches V, smaller and smaller regions of the wall polygons near W contribute almost all of the solid angle to the wall's half of the hemicube. Thus, the radiosity at W itself should determine the wall's contribution to the input illumination at V; other vertices of triangles containing W should have no effect. So when wall triangles containing W are rendered into the hemicube for V, the color components of these other vertices are set to be the same as those of W.

This second fix is necessary because $f_j(Q)$ should actually be linearly interpolated in object space, while the hardware shader interpolates in image space. Hardware that can interpolate texture coordinates in object space would give better accuracy on all triangles. However, the error is only significant on ones that pass close to the hemicube center.

The preceding discussion was for the special case of the edge of a room, but by similar reasoning, these small displacement and recoloring fixes apply at any concave corner.

For large data bases, it may be impractical to store all the form factors, so they may be regenerated at each step in the Gauss-Seidel iteration. To update the radiosity B_i, one would place a hemicube at vertex P_i, find the corresponding $\mathbf{F}_{ij}$'s and, in each wavelength band, replace B_i by $\Sigma_j B_j \mathbf{F}_{ij}$. This is mathematically equivalent to rendering a colored, gouraud-shaded image into the hemicube, using the current radiosities, and then summing the image, weighted by the Δ form factors. When we tried this image summation method, we found that the 8-bit dynamic range for the image was not sufficient to accurately represent both the light sources and the diffuse surfaces. Therefore, we did a first iteration with just the light source emissivities to find the direct illumination. Then

we turned off the emissivities in subsequent Gauss-Seidel iterations for the indirect illumination.

The image summation method eliminates the need to access the Item-Buffer, the vertex pointers, and the $\mathbf{F}_{ij}$. In our experience it is faster than the vertex-to-vertex form factor method, for three or fewer iterations, possibly because the random accesses for the $\mathbf{F}_{ij}$ and vertex pointers frequently missed the memory cache on the Iris CPU. The brute force "render the scene from the point of view of every vertex," previously considered hopeless, actually wins with modern hardware!

Acknowledgment

See also G2, 303.

DELTA FORM-FACTOR CALCULATION FOR THE CUBIC TETRAHEDRAL ALGORITHM

Jeffrey C. Beran-Koehn and Mark J. Pavicic
North Dakota State University
Fargo, North Dakota

The *hemicube algorithm* has become the most popular method of calculating radiosity solutions for complex environments containing hidden surfaces and shadows (Cohen and Greenberg, 1985). The *cubic tetrahedral algorithm* replaces the hemicube with a cubic tetrahedron (Beran-Koehn and Pavicic, 1991). This adaptation increases performance by reducing the number of projection planes from five to three, while maintaining the simplicity of the required clipping and projection operations.

The cubic tetrahedral algorithm calculates the amount of light landing on a patch from every other patch by transforming the environment so that the receiving patch's center is at the origin and its surface normal coincides with the vector (1, 1, 1). An imaginary cubic tetrahedron is constructed around the center of the receiving patch by selecting the points (–2, 1, 1), (1, –2, 1), and (1, 1, –2) as the base, and the point (1, 1, 1) as the apex. This provides three symmetric faces onto which the environment is projected. The cubic tetrahedron is illustrated in Fig. 1.

Each face of the cubic tetrahedron is divided into square cells at a given resolution. Associated with each cell is a delta form-factor that specifies the fraction of the light passing through the cell and landing on the center of the patch.

The delta form-factor of a cell with an area of *a,* centered at the point *P* is computed as follows:

$$\Delta \mathrm{FF}(P,\ a)\ =\ \frac{\cos\ \alpha\ \cos\ \beta}{\pi r^2}a,$$

where α is the angle between $\mathbf{N}_p$, the surface normal of the receiving patch, and $\mathbf{R}$, the vector between the origin and the point P; β is the angle between $\mathbf{N}_c$, the cell's surface normal, and $-\mathbf{R}$; and r is the length of $\mathbf{R}$. These relationships are illustrated in Fig. 2.

The surface normal of the receiving patch is $(1, 1, 1)$ and therefore the $\cos \alpha$ term is

$$\cos \alpha = \frac{(P_x, P_y, P_z) \cdot (1, 1, 1)}{\left|(P_x, P_y, P_z)\right| \left|(1, 1, 1)\right|} = \frac{P_x + P_y + P_z}{r 3^{1/2}}.$$

For cells on the face of the cubic tetrahedron perpendicular to the y axis, the $\cos \beta$ term is

$$\cos \beta = \frac{(-P_x, -P_y, -P_z) \cdot (0, -1, 0)}{\left|(-P_x, -P_y, -P_z)\right| \left|(0, -1, 0)\right|} = \frac{P_y}{r} = \frac{1}{r}.$$

The same result is obtained for cells on the other two faces of the cubic tetrahedron.

Thus, the delta form-factor of a cell with area a, centered at the point P on any face of the cubic tetrahedron, is

$$\Delta\text{FF}(P, a) = \frac{P_x + P_y + P_z}{\pi \left(P_x^2 + P_y^2 + P_z^2\right)^2 3^{1/2}} a.$$

For each face, one of P_x, P_y, or P_z will always be 1, and thus ΔFF may be simplified to

$$\Delta\text{FF}(u, v, a) = \frac{u + v + 1}{\pi \left(u^2 + u^2 + 1\right)^2 3^{1/2}} a.$$

where u and v range from 1 to -2 and originate from the apex of the cubic tetrahedron.

Because of symmetry, the delta form-factor values need to be computed for only one-half of any face. Figure 3 illustrates a cubic tetrahedral face with a cell resolution of 8×8 for which the shaded cells have been selected for delta form-factor calculation. The cells located along the base
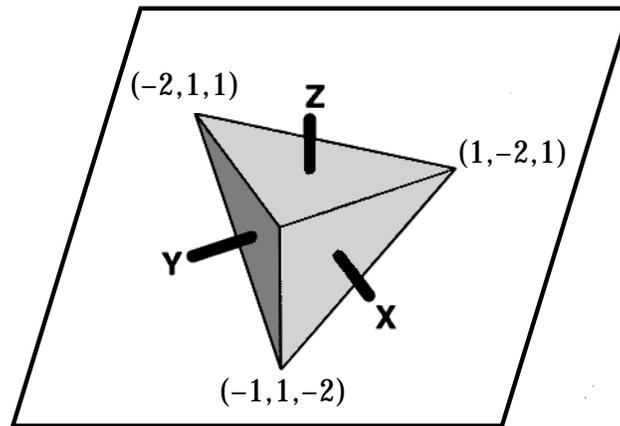
Figure 1. The cubic tetrahedron.
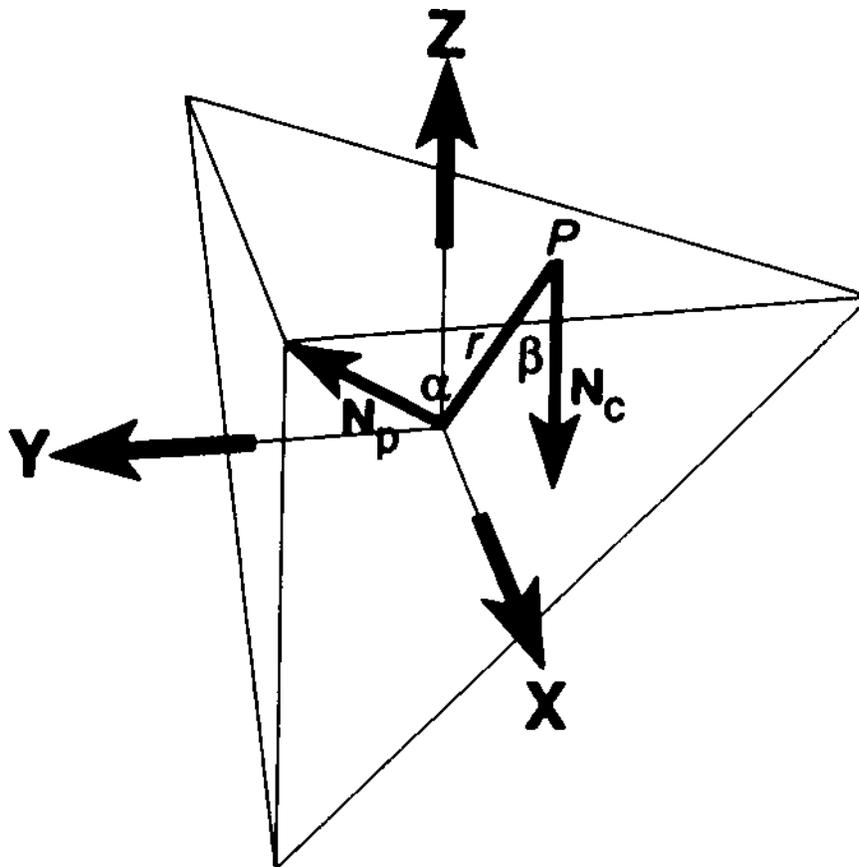
Figure 2. Delta form-factors.
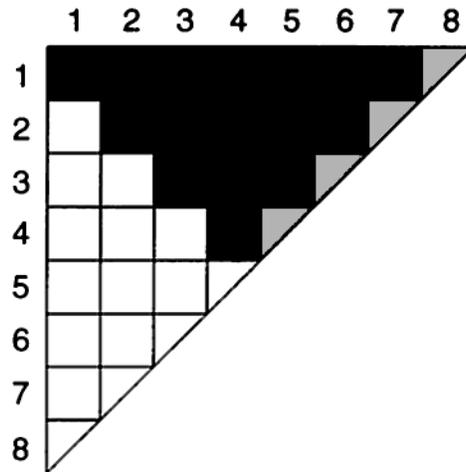
Figure 3. Delta form-factor symmetry.
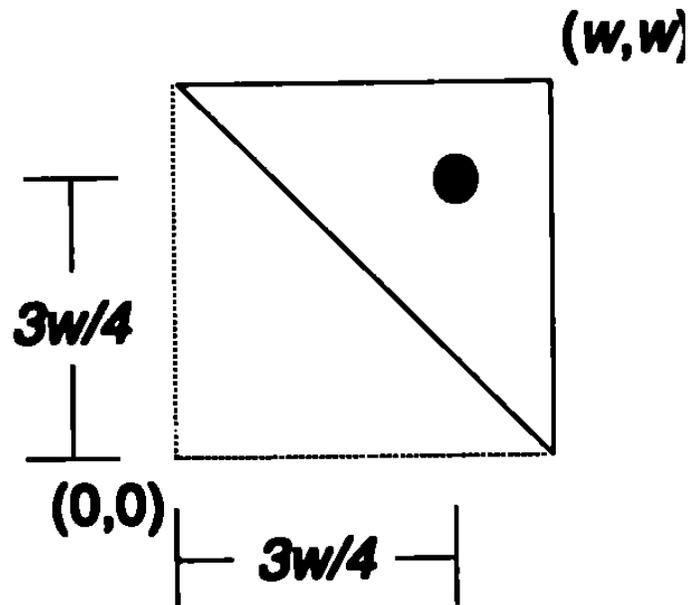
Figure 4. Center of a base cell.

of the cubic tetrahedron, those with a triangular shape, present a special case. These cells may be ignored, especially if the number of cells is large, since their delta form-factors will be very small. If these cells are included, the area is one-half that of the other cells and the center is calculated as shown in Fig. 4. See the appendix for a program that demonstrates the calculation of delta form-factors.

See also G2, 299.

VI.11

ACCURATE FORM-FACTOR COMPUTATION

Filippo Tampieri
Cornell University
Ithaca, New York

The radiosity method for diffuse environments (Goral *et al,* 1984; Cohen and Greenberg, 1985) expresses the energy transfer between a source patch *s* and a receiving patch *r* as

$$\Delta B_r A_r = r_r B_s A_s F_{As \to Ar},\tag{1}$$

where

ΔB_r is the unknown to be computed; it is the contribution of patch *s* to the radiosity of patch *r*, assumed constant across patch *r*;

A_r is the area of patch *r*;

ρ_r *is* the reflectivity of patch *r*;

B_s is the radiosity of patch *s*, assumed constant;

A_s is the area of patch *s*; and

$F_{As \to Ar}$ is the form-factor from patch *s* to patch *r* and represents the fraction of energy leaving patch *s* and arriving at patch *r*.

Lying at the core of the radiosity formulation, the accurate computation of form-factors plays an important role in determining the quality of the radiosity solution.
 This Gem presents a simple algorithm for the computation of accurate form-factors based on a number of ideas and techniques assembled from previously published work (Baum *et al.,* 1989; Wallace *et al.,* 1989; Hanrahan *et al.,* 1991). The algorithm computes the form-factor between

a finite-area polygonal source and a differential area on a receiving patch and can be used directly within the progressive refinement radiosity method (Cohen *et al.*, 1988).

If a differential area dA_r, centered about location x on patch r, is substituted for finite area A_r in Eq. (1), we obtain

$$\Delta B_r(x)\, dA_r(x) = \rho_r B_s A_s F_{A_s \to dA_r(x)}. \tag{2}$$

Using the reciprocity relationship for form-factors, $A_s F_{A_s \to dA_r} = dA_r F_{dA_r \to A_s}$, and dividing both sides of Eq. (2) by dA_r, leads to

$$\Delta B_r(x) = \rho_r B_s F_{dA_r(x) \to A_s} = \rho_r B_s \int_{A_s} v(x,\, y) \frac{\cos \theta_r\; \cos \theta_s}{\pi r^2}\, dA_s(y), \tag{3}$$

where

$\Delta B_r(x)$ is the radiosity contribution of patch s to a differential area dA_r centered about point x;

$v(x,\, y)$ is a visibility term, 1 if point y is visible from x, 0 otherwise;

r is the distance between x and y;

$\theta_r,\, \theta_s$ are the angles between the surface normals at x and y and the line connecting the two points; and

$dA_s(y)$ *is* the differential area centered about point y on patch s.

In the absence of occlusions, the visibility term is everywhere 1, and the area integral can be transformed into a contour integral using Stoke's theorem. This, in turn, can be evaluated analytically using the formula provided by Hottel and Sarofin (1967):

$$F_{dA_r(x) \to A_s} = \frac{1}{2\pi} \sum_{i=0}^{n-1} \frac{\cos^{-1}(R_i \cdot R_{i \oplus 1})}{\| R_i \times R_{i \oplus 1} \|}\, (R_i \times R_{i \oplus 1}) \cdot N_r, \tag{4}$$

where

$\oplus$ represents addition modulo n;

n *is* the number of vertices of source patch s;

R_i is the unit vector from point x on patch r to the ith vertex of patch s; and

N_r is the unit normal to patch r at point x.

Assuming the visibility term v were constant, it could be taken out from under the integral in Eq. (3), and the form-factor could be computed correctly using Eq. (4). Unfortunately, v is constant only when source patch s is either totally visible ($v(x, y) = 1$) or totally occluded ($v(x, y) = 0$) from the receiving differential area $dA_r(x)$. When partial occlusion occurs, however, approximating the visibility term with a constant function might result in a poor estimate of the form-factor.

A solution to this problem is not hard to find. By subdividing the source patch into smaller and smaller areas ΔA_j and using a different constant approximation to v for each ΔA_j, the visibility function can be approximated to any degree of accuracy, as can the integral in Eq. (3). Thus, the contribution of source s to receiving point x becomes

$$\Delta B_r(x) = \rho_r B_S \sum_{j=1}^{m} \left(V_j F_{dA_r(x) \to \Delta A_j} \right),$$

where V_j is the fraction of ΔA_j visible from x.

The following pseudo-code gives a recursive algorithm to compute ΔB_r:

```
            ΔB_r ← ComputeContribution(r(x), s, ε_B, ε_A);
            ComputeContribution (r (x), s, ε_B, ε_l)
  1.            if CanCull(r(x),s)
                    then return 0;
                    else begin
  2.                      ε_F ← ε_B/(ρ_r B_s);
                         return ρ_r B_s· ComputeFormFactor(r(x),s, ε_F, ε_A);
                         end;


            ComputeFormFactor(r(x), s, ε_F, ε_A)
                begin
  3.            V ← ComputeVisibilityFactor(r(x),s);
                if V ≤ 0
```

4.   **then return** 0;
         else if $V \geq 1$
                   then begin
5. $F_{dA_r \to A_s} \leftarrow$ ComputeUnoccludedFormFactor(r(*x*),s);
                         return $F_{dA_r \to A_s}$;
                         end:
         else begin
                   $F_{dA_r \to A_s} \leftarrow$ ComputeUnoccludedFormFactor(r(x),s);
6.   **if** $F_{dA_r \to A_s} \leq \varepsilon_F$ **or** $A_s \leq \varepsilon_A$
7.   **then return** $VF_{dA_r \to A_s}$;
                   else begin
8. $(s_1, s_2, \ldots, s_m) \leftarrow$ SplitPatch(*s*):
9.   **return** $\Sigma_{j=1}^{m}$ComputeFormFactor(r(x), s_j, ε_F, ε_A);
                         end;
                   end;
         end;

Subroutine *ComputeContribution* takes as input the description of a differential area centered about point *x* on patch *r*, the description of a source patch *s*, a radiosity tolerance ε_B, and an area ε_A, and uses recursive subdivision of the source patch *s* to compute the radiosity contribution within an accuracy controlled by ε_B. The area term ε_A is used to prevent excessive subdivision such as could occur for a receiving point very close to an edge of the source.

The subroutine starts by calling subroutine *CanCull* to check whether receiving point *x* is oriented away from patch *s* or is behind it (step 1), and returns a null contribution if either of the conditions hold. If point *x* cannot be culled out of consideration, a tolerance ε_F is derived (step 2) so that a form-factor computed within precision ε_F yields a radiosity contribution accurate within the required precision ε_B. Subroutine *ComputeFormFactor* relies on an external routine to get an estimate of the visibility factor *V* (step 3); *ComputeVisibilityFactor* should return 0 for total occlusion, 1 for total visibility, and the fraction of source *s* that is visible from point *x* in the case of partial occlusion The cases when *V* is either 0 or 1 are simple: The radiosity contribution is respectively equal to 0 (step 4) and to $\rho_r B_s$ times the unoccluded analytical form-factor (step 5), computed by subroutine *ComputeUnoccludedFormFactor* using Eq. (4).

The case of partial occlusion deserves more care: If the unoccluded analytical form-factor falls within the ε_F tolerance (step 6), then the radiosity contribution cannot possibly be larger than the desired accuracy ε_B and is estimated by the product of the visibility factor V and the computed unoccluded form-factor (step 7). If, instead, the unoccluded form-factor is larger than ε_F, the source is subdivided (step 8) and its contribution computed as the sum of the individual contributions of its parts (step 9). Also note that the recursive subdivision of the source is stopped if the area of the source falls below a specified threshold ε_A. This is done to prevent infinite recursion for receiving points along the corners and edges of the source, as suggested by Hanrahan *et al.* (1991).

The accurate computation of the visibility term V may not be a simple task. Such a computation could be carried out by projecting every polygon onto the source from the point of view of the receiving point and then computing the ratio of the sum of the unoccluded areas of the source to its total area. Unfortunately, this approach is prohibitively expensive, and faster methods must be used.

As an alternative to computing the visibility term V exactly, a reasonable approximation can be computed much more efficiently by tracing one or more shadow rays between the source and receiver and averaging their 0/1 associated visibilities (Hanrahan *et al.*, 1991.) The accuracy and efficiency of this technique is improved even further by testing for total occlusion or total visibility first, and using shadow rays only if neither case can be recognized with certainty.

Shaft culling (Haines and Wallace, 1991) provides a means of computing a list of potential occluders for a given source-receiver pair. If the list is empty, the source is totally visible; if an object in the list cuts the shaft and completely separates the source from the receiver, then the source is totally occluded; otherwise, the source is treated as partially occluded, and shadow rays are used as described above. Note that shaft culling also speeds up this last process by reducing the number of possible occluders that need to be tested against shadow rays.

See also G3, F.8.

VII

RENDERING

VII

RENDERING

This final chapter is a sort of "catch all" for many Gems that did not fit explicitly into other sections. As such, "rendering" means anything to do with making pictures. That is not to say that the section contains the dregs; in fact, some of my favorite Gems in this volume are in this very section.

The first Gem describes an optimization to create better images using a shadow depth buffer. The second Gem describes a clever trick for organizing the arithmetic in color interpolation across a scanline. The third Gem describes some shortcuts for anti-aliasing polygons. The fourth Gem describes a general programming trick for the C language, which has some nice side effects for span conversion. The fifth Gem presents a framework for progressive refinement display of a sampled image.

The sixth Gem is one of my personal favorites. It describes a way to think about how to decide which pixels to draw as part of a triangle to be rendered. The technique is accurate and consistent, and the exposition is very clear. The seventh Gem discusses a little trick for making interesting shadow effects using negative light sources, or "darklights." The eighth Gem asks the question "What if pixels were triangular?" and then provides some useful tools for rendering into triangular pixels.

The ninth Gem takes advantage of commonly available workstation graphics hardware to create higher quality images than the hardware was designed to make! This Gem describes how to use polygon drawing hardware to make motion blurred images for still or video recording. It also discusses some of the finer points of filtering for video. The final Gem describes a useful optimization for drawing polygons that are part of a mesh. This Gem applies the concept of cacheing to the shading process.

VII.1

THE SHADOW DEPTH MAP REVISITED

Andrew Woo
Alias Research
Toronto, Ontario

Introduction

The shadow depth map has proven to be an effective shadow determination approach in image synthesis (Williams, 1978; Hourcade, 1985; Reeves *et al.,* 1987). Not only does it handle any class of geometric primitives with equal ease, it is also the only shadow approach that requires a storage complexity independent of the number of objects in the scene (Woo *et al.,* 1990): an advantage when it comes to complex scenes. However, the depth map is prone to aliasing problems, of which some have been improved (Hourcade, 1985; Reeves *et al.,* 1987) using some filtering techniques. In this Gem, we attempt to reexamine the Moiré Pattern aliasing problem, and offer a superior solution.

The Shadow Depth Map

The basic shadow depth map approach :is very simple. Perform a Z-buffer scan-conversion and visibility determination of the scene from the perspective of a light source instead of the eye. However, the visibility is simplified in that only the depth information is kept in 2-D grid or map, with no information about the visible objects at all. During rendering of each pixel, the point or region to be shaded is projected onto the appropriate cell(s) in the depth map. A *SampleShadow routine* is called to compare the depth value for each applicable depth map cell with that for the current intersection point. If the intersection point is further away

from the light source than the distance stored in the depth map for that cell, then the point is in shadow; otherwise, it is not.

The Moiré Pattern Problem

In Fig. 1, when the closest surface to the left of the point marked (*) is to be shaded, it will improperly result in self-shadow because the depth map value is further away from the light source than the closest surface. Reeves *et al.* (1987) attempt to solve this problem by introducing a *bias* that is added to the depth value on the fly so the same surface will be in front of the depth map value. Note that if *bias* is too large to avoid the moiré pattern problems, then objects that should be shadowed might be fully lit. If the *bias* is not large enough, then improper self-shadowing occurs much more often—this usually results in the appearance of noisy surfaces or moiré patterns (noise from the randomness attached to the *bias* value)—see Fig. 2. Thus, no matter how carefully *bias* is chosen, the shadow results are usually unsatisfactory, and are especially noticeable for low depth map resolutions < 512×512.
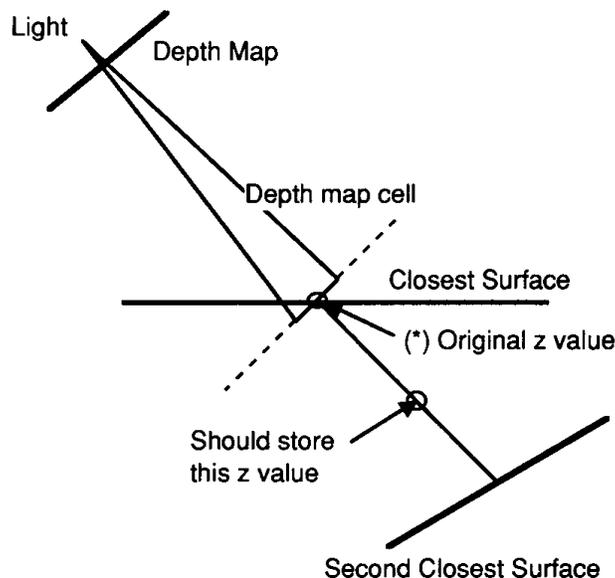
Figure 1.
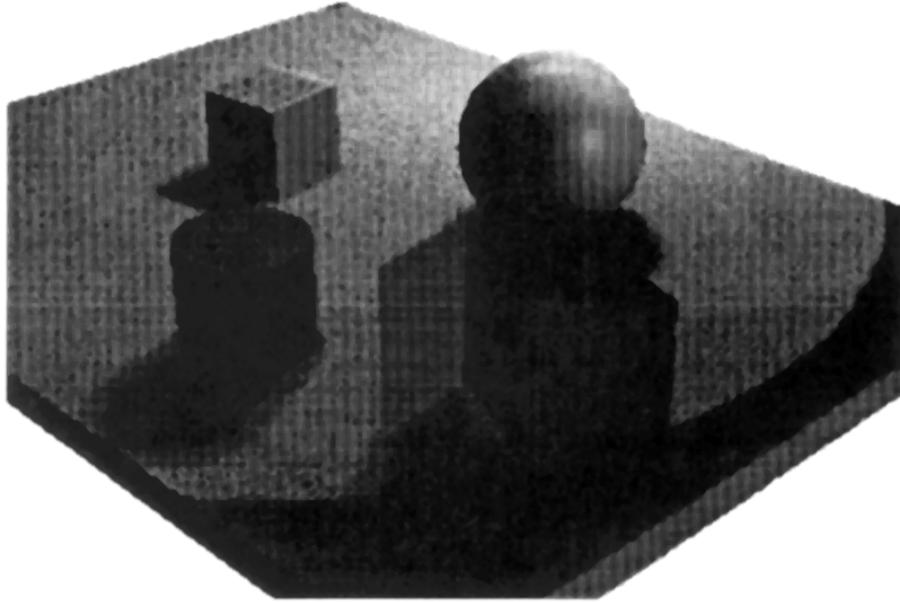
Figure 2. The old algorithm.

Another solution instead of *bias* is proposed here. If the depth map cell only encounters one surface, then the depth map value is set to the same value as if there were no surfaces in that cell, i.e., the z value is set to some large value. Such a surface will always be fully lit where it intersects that cell.

If there is more than one surface that hits the depth map cell, then the depth map cell value is assigned to be an average (halving the sum of the values is a good rule of thumb) between the first two visible surfaces with respect to the light source; see Fig. 1. In this way, the closest surface should always be lit and the second surface should always be in shadow. Thus, the *bias* value is not needed at all, and the results generated are far superior.

A Boundary Case

Special care is required at the outer limits of a spotlight's cone of illumination. When sampling occurs outside this cone, make sure that an

in shadow result is returned. Otherwise, spurious brightly lit lines may appear at the juncture between lit and shadowed regions. Reeves *et al.* (1987) provided source code that had this error.

Some Optimizations

The changes proposed above not only produce better quality shadows, but should also be faster computationally. For example, there is no need to compute bias values on the fly for each shadow sample (in the *SampleShadow* routine). Furthermore, because halving the depth value between visible surfaces results in large value differences during depth comparisons, depth values can be compared in integer first, then done in floating-point only if the integer comparison is inconclusive. This tends to help out on platforms where integer evaluations are much faster than floating-point. Available is some pseudo-code showing Reeves *et al.'s SampleShadow* routine vs. the new approach.
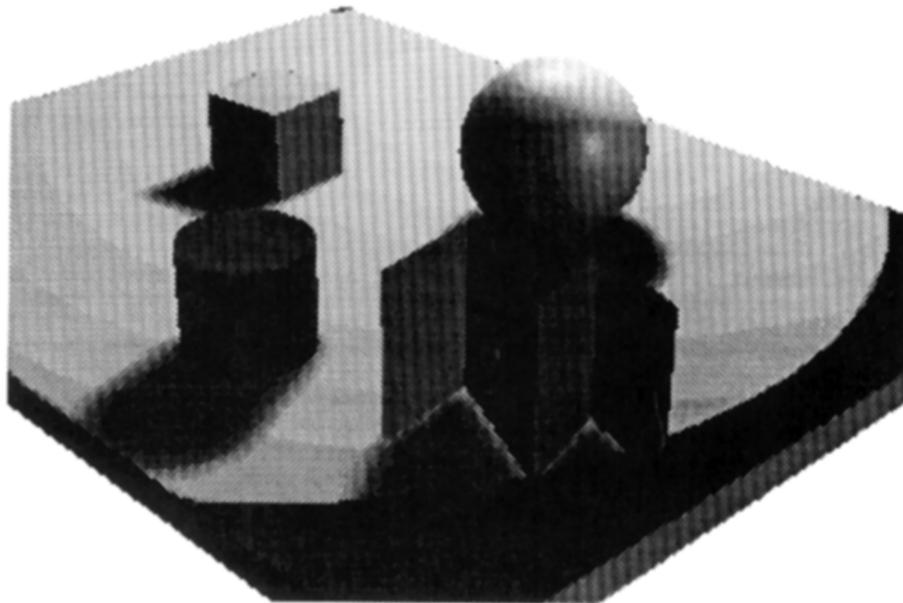
Figure 3. The new algorithm.

Conclusions

The changes made to the shadow depth map approach seem to work quite well, even at a low resolution of 256×256. This can be seen in Figs. 2 and 3, where a 256×256 depth map is used to render the scene: Notice the difference in the noise level of the surfaces. The spotlight frustum is 75 degrees, and the minimum and maximum bias values are 0.2 and 0.4, respectively. Thus, a $1,024 \times 1,024$ resolution depth map is not always necessary for realistic shadows. With this approach, a very reasonable job can be achieved at much lower resolutions and thus use up a lot less computation and memory.

See also G2, 273; G2, 275.

VII.2

FAST LINEAR COLOR RENDERING

Russell C. H. Cheng
University of Wales, College of Cardiff
Cardiff, United Kingdom

This Gem gives an algorithm for doing fast *z*-buffered linear color rendering of polygons.

Normally, linear shading is an expensive computation because separate interpolation is needed to calculate each of the attributes *r, g, b,* and *z* at each pixel in the polygon. Additionally, and still at the pixel level, the separate *r, g, b* values have to be merged to form the overall value representing the given color. (See, for example, Heckbert, 1990.) Hardware solutions for doing the interpolation and merging can overcome speed problems, but only at the expense of a less flexible format in the choice of color mode. A software solution is clearly desirable in this latter respect, or when a hardware implementation is not available.

When the shading is linear, the color-slopes, i.e., the rates of change of *rgb* values, remain constant over the entire polygon. There is actually no need to do the interpolation for the *r, g,* and *b* values separately for every pixel of the entire polygon; it is only necessary to make the interpolation along a single scanline. The algorithm below takes advantage of this fact to yield a renderer that is significantly more efficient, particularly for large polygons, than that which is conventionally described.

The rationale of the method is given for the case when the color value is held as the first 24 bits of a 32-bit integer, i.e., 8 bits each for *r, g,* and *b*—with *r* the first byte, *g* the second, and *b* the third. Thus, *r, g,* and *b* each lie in the range 0 through 255. The method is easily modified for other color representations.

For definiteness a floating-point *Z*-buffer is assumed, but this is not essential to the method.

Typically, linear shading is implemented by stepping through each scanline, y, that intersects the polygon. Suppose that

1.   xleft[y] and xright[y] are the leftmost and rightmost x-values of the pixels lying in the polygon on this scanline, and that the polygon covers all pixels between these two values;

2.   the r, g, b, and z values are initialized to their correct values at the leftmost pixel of this line-segment, i.e., at position xleft[y], y;

3.   floating point r, g, b color shade increments in the x-direction have been precalculated as dr_by_dx, dg_by_dx, db_by_dx, and the z-distance increment is dz_by_dx.

Then the pixels of a given scanline can be rendered with a loop calculation that, because of the linearity of the shading process, involves r, g, b, z increments in the x-direction only, the incrementation in the y-direction being only needed to set the r, g, b, z values in (2) above at the start of the scanline. The loop calculation for the scanline takes the form

```
for x ← xleft[y] to xright[y] do begin
    if z < zbuffer [x][y] then
        begin
        framebuffer[x][y] ← ((int)r) + (((int)g) << 8) +
                                (((int)b) << 16);              Step A

        zbuffer[x][y] ← z;
        end;
    r ← r + dr_by_dx;    Step B
    g ← g + dg_by_dx;    Step B
    b ← b + db_by_dx;    Step B
    z ← z + dz_by_dx;
    endloop;
```

This constitutes the innermost loop. It proceeds by incrementing the r, g, b values separately (Step B). The variables r, g, and b have to be floats to retain accuracy over the scanline. In Step A, these values are rounded into integers and merged to form the actual 24-bit colour value that is inserted into the frame-buffer. In the overall linear shading algo-

rithm, the greatest computational expense thus lies in these calculations at Steps A and B.

A significant improvement can be made to the preceding on noting that linearity of the shading process means that Steps A and B are in effect identical for each scanline. This is because, as far as the *x*-direction is concerned, the *same* merged 24-bit rgb *x*-increment is used for all pixels with the same *x* value. This means that, if the values of these 24-bit rgb increments are recorded for each value of *x* along any one scanline, then they can be used at Step A for every scanline, without the need to recalculate them. The separate incrementation of *r, g,* and *b* at Step B is only needed, of course, in the calculation of the merged 24-bit increments along the one scanline.

The pseudo-code for this fast linear renderer is as follows:

```
procedure FastLinearRend(
    x min: int;       leftmost x value amongst all pixels of the
                      polygon being rendered
    x max: int;       corresponding rightmost x value
    y min: int;       smallest y value amongst all pixels of the
                      polygon being rendered
    y max: int;       corresponding largest y value
    xleft: array[0. .HRES-1] of int; on entry xleft[y min], . . . ,
                      xleft[y max] hold the x value of the leftmost pixel
                      lying in the polygon
    xright: array[0. .VRES-1] of int; on entry
                      xright[ymin]. .xright[y max] hold the x value of
                      the rightmost pixel lying in the polygon, for
                      each scanline
    r0,g0,b0,z0:      float; ualue of r, g, b, and z at the base
                      pixel at position x0, y0. The base pixel need
                      not necessarily be a pixel of the polygon but
                      is typically one of its vertices.
                      Alternatiuely the base pixel can be taken to
                      be the origin (0, 0).
    dr_by_dx, dr_by_dy,
    dg_by_dx, dg_by_dy,
    db_by_dx, db_by_dy,
    dz_by_dx, dz_by_dy; float;
```

> *incremental slope information giving the rate of change of each attribute in the x and y directions.*

```
);
r, g, b, z, r1, g1, b1, z1, dx, dy: float; work variables
x, y: int; position of current pixel
z: float; z-value of pixel
screenbuff: array [0..HRES-1, 0. .VRES-1] of int;
                the screen-buffer, for clarity defined here as a
                two-dimensional array, but usually occupies
                linear memory. HRES and VRES are the
                horizontal and uertical resolutions of the
                frame.
zbuffer: array [0. .HRES-1, 0. .VRES-1] of float; z-buffer
col: int; work variable storing 24-bit rgb color
rgboffset: array [0. .HRES-1] of int; holds the merged
24-bit color x-offsets
begin
   Find the r, g, b, z value at position (x0, y min)
   dy ← (y min – y0);
   r1 ← r0 + dr_by_dy*dy;
   g1 ← g0 + dg_by_dy*dy;
   b1 ← b0 + db_by_dy*dy;
   z1 ← z0 + dz_by_dy*dy;
   Find the combined 24-bit rgb offset values along the scanline y min
   for x ← x min to x max do begin
      dx ← (x – x0);
      r ← dr_by_dx*dx;
      g ← dg_by_dx*dx;
      b ← db_by_dx*dx;
      rgboffset[x] ← ((int)r) + (((int)g) ≪ 8) + (((int)b) ≪ 16)
      endloop;
   for y ← y min to y max do begin
      find the 24-bit color value at (x0, y)
      col ← ((int)r1) + (((int)g1) ≪ 8) + (((int)b1) ≪ 16);
      find the z ualue at (xleft[y], y)
      dx ← xleft[y] – x0;
      z ← z1 + dz_by_dx*dx;
```

```
    do the scanline fill
    for x ← xleft[y] to xright[y] do begin
    if z < zbuffer[x][y] then begin
    screenbuff[x][y] ← col + rgboffset[x];
    zbuffer[x][y] ← z;
    end;
    z ← z + dz_by_dx;
    endloop;
  Increment the leftmost rgbz ualue ready for the next y
    r1 ← r1 + dr_by_dy;
    g1 ← g1 + dg_by_dy;
    b1 ← b1 + db_by_dy;
    z1 ← z1 + dz_by_dy;
    endloop:
  end;
```

There are several points to note in this algorithm.

1. The calculation does introduce an additional rounding error because the color intensities, $r1$, $g1$, $b1$, at the leftmost pixel of a scanline have to be rounded as well as the *rgb* offset values. Thus the overall rounding error can be +/– one unit rather than the usual +/– half unit. In my experience, with 8-bit color intensities and in real-time applications this effect is not noticeable.

2. None of the offset values of any color intensity have to lie in the range 0 to 255. So long as the final intensities of *r, g,* and *b* at any pixel lie properly in range, then any numerical "overflow" into a neighbouring byte at an intermediate calculation is automatically allowed for by the usual arithmetic operations of carrying and borrowing. Thus, intensity values are correct and positioned each in their own byte (apart from rounding) by the end of the calculation, and no special assumptions are needed to avoid colour-wrapping effects.

3. Colour overshoot because of rounding can occur at edges where an intensity is at or very near its maximum or minimum. This is a problem with any shading renderer and is most acute in small polygons, but the additional rounding error mentioned in 2 means that extra care is needed to avoid this effect. I usually adjust polygon

vertex intensities to be slightly away from their minimum and maximum values (taking care to preserve linearity) to prevent this.

4. The speed gain is dependent on the number, shape, and size of polygons. Typically I found an increase by factors of 2.3, 3.3, and 3.8 for 50, 500, and 5,000-pixel polygons, respectively.

5. Finally it is worth repeating the warning mentioned by Heckbert, that linear interpolation is not correct for all attributes. In particular the z-distance attribute is not affine, and in perspective-sensitive scenes use of linear approximation can be distracting. However, the reciprocal z-distance is affine, so that linear interpolation is correct if z-buffer calculations are done with reciprocal-z. The only change needed in the algorithm is to alter the z-test in the inner loop from (z < z_buffer[x][y]) to (z > z_buffer[x][y]).

See also G1, 75.

VII.3
EDGE AND BIT-MASK CALCULATIONS FOR ANTI-ALIASING

Russell C. H. Cheng
University of Wales, College of Cardiff
Cardiff, United Kingdom

Introduction

At the heart of many antialiasing algorithms for rendering polygons is a routine for calculating the overlap between a polygon and the (square) pixels at the edge of the polygon. (See for example Watt, 1989, for a discussion of such area sampling.) Anyone who has had to implement such an edge calculator will have encountered the problem of ensuring that the code

1.   handles polygons consistently (e.g., abutting faces are seam-free);

2.   does not become bogged down by numerous special cases (e.g., copes with a tiny face lying in a single pixel);

3.   remains accurate under difficult conditions (e.g., copes with long, thin polygons);

4.   does not contain redundant calculations (e.g., does not reset variables that are already clear).

   An algorithm follows for such an edge-calculator in which particular attention has been paid to the preceding points so that it operates in a clean, efficient, and fast manner. The algorithm has been used successfully in real-time applications. The basis of the method is a generalized Bresenham-type calculation that switches between not one, but two error variables.

The algorithm identifies those pixels intersected by an edge, and the nature of each intersection. This information can be used, for example, to calculate the bit-masks of an A-buffer anti-aliasing renderer as described by Carpenter (1984).

Alternating Bresenham Edge-Calculator

The routine works, scanline by scanline, up to the left edges of the face, then the right edges. (Here "up" means "in the direction of increasing y," though for many displays scanlines are labelled from top to bottom.) Four types of edge are used, and these are depicted in Fig. 1.

The form of the intersection of an edge with a pixel is determined by the two end points of the edge either within or at the boundary of the pixel square. At the core of the routine is the calculation of these end points for each pixel. The traversal from pixel to pixel along edgetypes 1 or 3 is illustrated in Fig. 2. Each square represents a pixel and the way an edge can intersect it. The arrows between pixels indicate the only possible ways of switching from one pixel lo another. For example, from a type "mx" it is only possible to go to another "mx," an "sy," or an "rx."

The switching between pixel types is controlled by two Bresenham error quantities: ex and ey, one in the x and the other in the y direction. Use of two error terms rather than one gives the calculation a simple
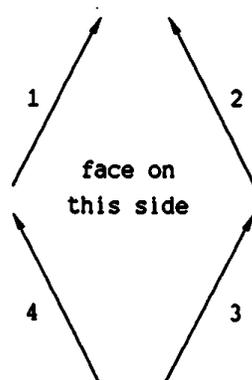
Figure 1. Edge types.
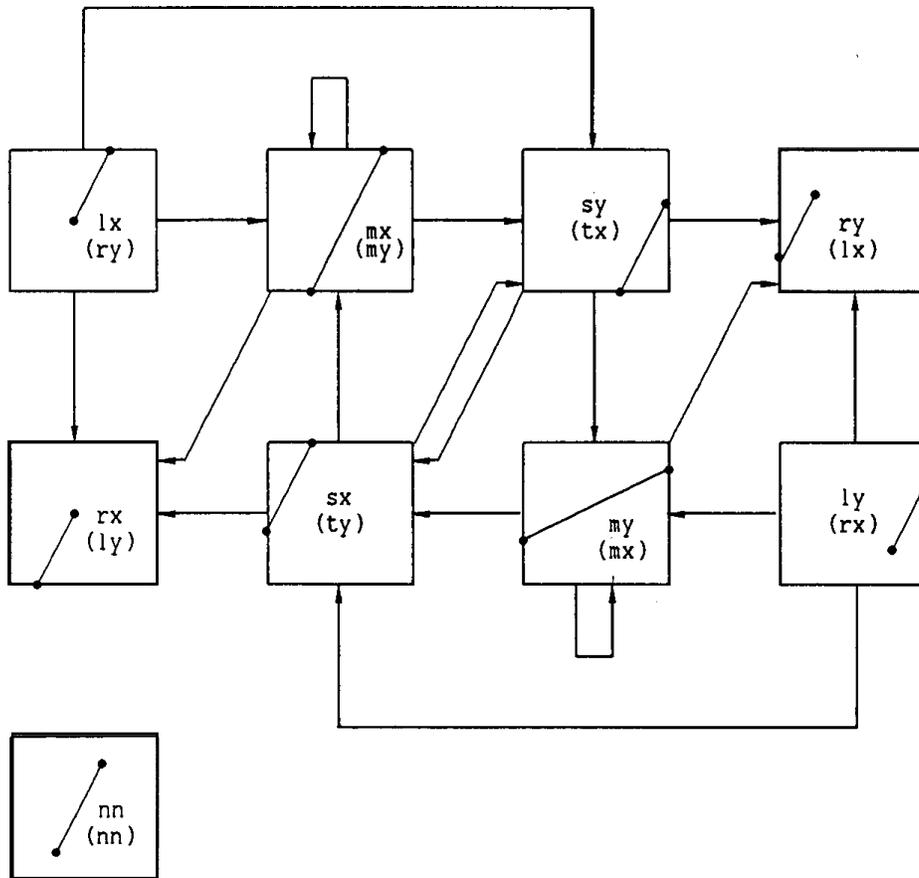
Figure 2. Possible edgetype 1 and 3 traversals. For edgetype 2 and 4 traversals, rotate the diagram 90° anticlockwise and use the labelling in brackets.

symmetry, and at the same time allows the coordinates of the end points of the edge-segment to be expressed immediately in terms of them.

Tests of *ex* alternate with tests of *ey* to find the next type of pixel encountered, with the end conditions of one block of tests forming the starting conditions of the next block. The alternating sets of computation dovetail to form a compact and efficient code. Details appear in the C-code version.

The edge-calculator also records, for each horizontal row of pixels intersected by the face, the leftmost and rightmost pixel affected: xleft[y], xright[y].

Bit-Mask Calculations

We illustrate how the edge-calculator might be utilized to set up bit-masks for use in an A-buffer anti-aliased renderer.

The method is not dependent on the precise size of bit-mask, but to be specific, suppose that each pixel is divided into $4 \times 4 = 16$ square subpixels so that the bit-mask of each pixel can be represented by a 16-bit integer. Suppose that array element frag[x][y] is used to record the bit-mask of the overlap of a face with the pixel at position *x, y*. Let each element of the array be initialized as ffff(hex) (i.e., the subpixels are all assumed covered—indicated by each bit being set equal to 1).

As each pixel overlapping the boundary of the face is obtained, the bit-mask corresponding to the particular overlap can be read-off precalculated arrays:

mask[1] [][][][], mask[2] [][][][],
mask[3] [][][][], mask[4] [][][][],

each array corresponding to one of the four edge-types. The last four arguments are the coordinates of the two endpoints defining the overlap. If the precalculated bit-masks are set as in Fig. 3, i.e., where the subpixels in the area marked 0 are set to zero, whilst the remainder are set to 1, then each overlap, when it is identified by edge-calculate(), can have its precalculated bit-mask bitwise ANDed with the existing bit-mask, frag[x][y], of that pixel and the result stored back in frag[x][y]. The effect of this is to cut out portions of the bit-mask of the pixel not covered by the face. The precalculated bit-masks will correctly handle pixels containing one or more vertices of a convex face (see, for example, Fig. 4a, b) except for pixels containing a vertex that is the intersection of an edge-type 1 with an edge-type 3 or of an edge-type 2 with an edge-type 4. For these cases an additional ANDing operation is needed to clip out a fragment that should be, but that would not otherwise, be removed. (See Fig. 4c.) Note that for a convex polygon these cases can only occur once each at the bottom or top of a thin polygon, so that this correction is not at all time-consuming.

Once a face has been processed in this way, the elements of frag[][] lying between xleft[y] and xright[y] on each scanline will contain their
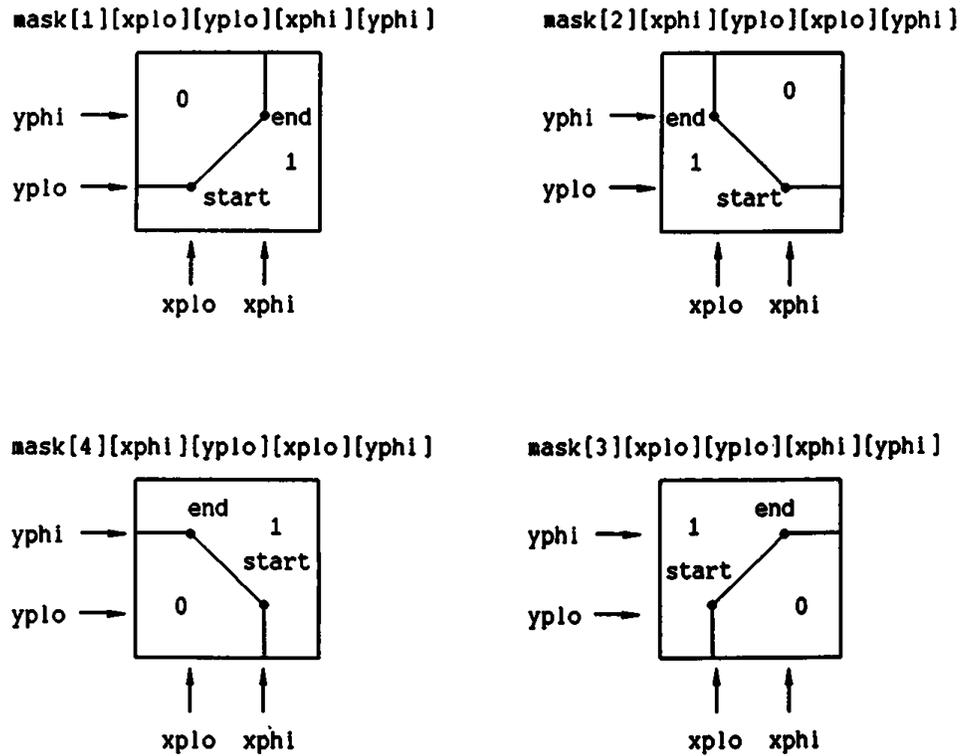
Figure 3. Suggested bit-mask definitions.
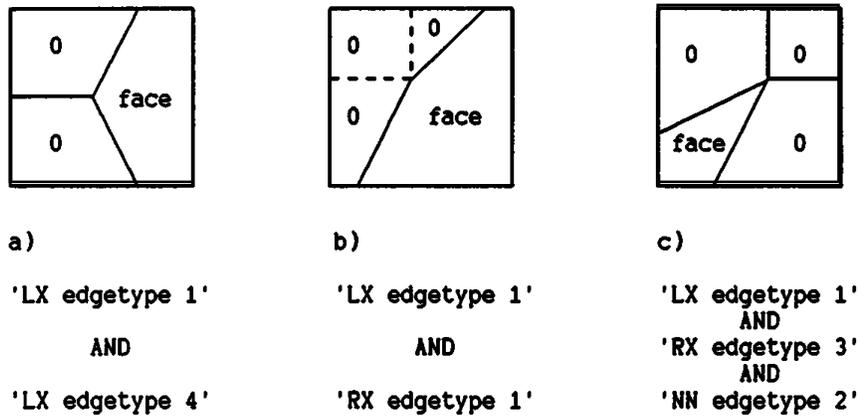
Figure 4. Examples of bit-masks at vertices formed by ANDing basic mask types.

correct bit-mask value; outside this segment the bit-mask will still be at their initialized ffff(hex) value.

Implementation Details

The implemented version contains all the calculations for ensuring that the bit-masks are correctly calculated, assuming the bit-mask-structure described above.

For reasons of space, the C-code version is written for compactness rather than full computational efficiency. An efficient version (about three times faster) is readily obtained by making the following two modifications:

1. For clarity, the main Bresenham-type calculation has been separated off, starting at label "gamma." This handles all four edge-types by using switches, the main ones being:

    a. toggle, which selects between edge-types 1 and 3 and edge-types 2 and 4,

    b. the lookup table rl[][], which is used in resultproc() to select the appropriate combinations of pixel and edge-types at which the *xl* and *xr* arrays are set.

    This code should place "inline" using the appropriate version of the calculation for each of the four cases, so that all switches and their associated tests are removed.

2. For clarity the error quantities *ex, ey,* and their associated variables are given using pixel-width as the unit of measure. This means that they have to be rescaled to subpixel units in resultproc() to access the bit-mask arrays. At the expense of more opaque-looking code, it is much more efficient to do all the calculations directly in subpixel units in the main Bresenham loop. Rescaling is then unnecessary when the bit-mask arrays are accessed.

See also G1, 76.

FAST SPAN CONVERSION: UNROLLING SHORT LOOPS

Thom Grace
Illinois Institute of Technology
Chicago, Illinois

Introduction

A common operation in many graphics applications is the illumination of a "span" of pixels: Several consecutive pixels are to be set to the same value. This occurs naturally in polygon scan-conversion, area filling, drop-shadowing, and the like. A closely related operation is the illumination of several consecutive pixels by values that are determined by differences (forward differencing is most common) from their immediate predecessors. This is essentially the situation in many interpolated shading techniques (e.g., Gouraud, Warnock, and the normal vector interpolation in Phong shading).

If we assume that consecutive pixels occupy contiguous memory locations, and that they are to be assigned common values, then the implementation of the preceding idea is most naturally expressed as a loop:

```
for (i = FirstPixel; i < = LastPixel; i + + )
    setpixel(i, val);
```

This compact implementation has, however, a serious drawback: Each pass through the loop requires the update and test of the loop control variable (*i* here), as well as a potential jump back to the start of the loop. These operations can easily equal the time used to do the useful work, namely the pixel illumination. This means that somewhat close to half of the time needed to illuminate a span of pixels may be spent "housekeeping."

There is a very popular technique by which the overhead introduced by the use of a loop (as described above) may be avoided. We speak, of course, about loop "unrolling." This is best illustrated by replacing

```
for i ← 1, 3 do
    whatever();
    endloop;
```

with the following:

```
whatever();
whatever();
whatever();
```

This Gem details programming techniques whereby this idea may be conveniently implemented.

Implementation

The C function movespan() shown in the Appendix will implement a complete unrolling of the pixel illumination loop for spans of up to 16 pixels. The idea, of course, is for control to pass into the appropriate section of the switch to move data into one pixel, then fall through the rest of the cases so that the rest of the pixel data is also moved. This relies on a peculiarity of the semantics of the switch in C; other languages (e.g., Pascal and the Graphics Gems pseudocode) may not be amenable to this trick.

Longer spans may be accommodated at the cost of additional memory consumption by the obvious insertion of more cases in the switch. Arbitrarily long spans may be handled by including a loop before the switch, as illustrated by the C function movelongspan(). Note that the constants in the while condition and the subtraction should be the same as the number of cases in the switch. Although this only partially unrolls a loop, it is a useful technique, especially if memory is scarce.

The operations carried out at each pixel are not limited to the movement of a single byte of data. The function shadespan() shown in the Appendix moves linearly interpolated values to the pixels in a span, where the interpolation is computed incrementally at each pixel. Several other variations are possible (e.g., moving more than one byte per pixel, compositing images, etc.), but the basic technique is the same. Note, however, that the effective time savings (as a percentage of the total time used) decrease when the amount of processing per pixel increases.

See also G1, 75.

VII.5
PROGRESSIVE IMAGE REFINEMENT VIA GRIDDED SAMPLING

Steve Hollasch
Kubota Pacific Computer, Inc.
Santa Clara, California

The most common method employed to display a raster image is to sample the image data along scanlines, usually from left to right, top to bottom When the pixel-painting process is fast enough, the particular pixel ordering chosen is unimportant as far as the viewer is concerned. However, if the pixel sampling is slow, or if the resolution is sufficiently high, the time to display the image can take minutes, hours, or days.

This Gem addresses the display of images by sampling at progressively finer grids over the entire image area. The advantage over scanline sampling is that for the equivalent expense of a few scanlines, the viewer quickly gets a good impression of the overall image. The image is displayed with an initially coarse resolution that eventually refines down to the final display resolution. Each pixel is sampled only once (as for scanline methods). This approach does assume, however, that displaying square regions of a solid color is not much more expensive than sampling a pixel.

The main idea is this:

1. Display an initial coarse sampling of the image data. For each initial picture region, sample a corner pixel and use this color to display the region.

2. Subdivide each of the picture regions into four quadrants of equal size. Sample each new quadrant at a corner pixel and assign the sampled pixel's color to the quadrant. For each region, there will be
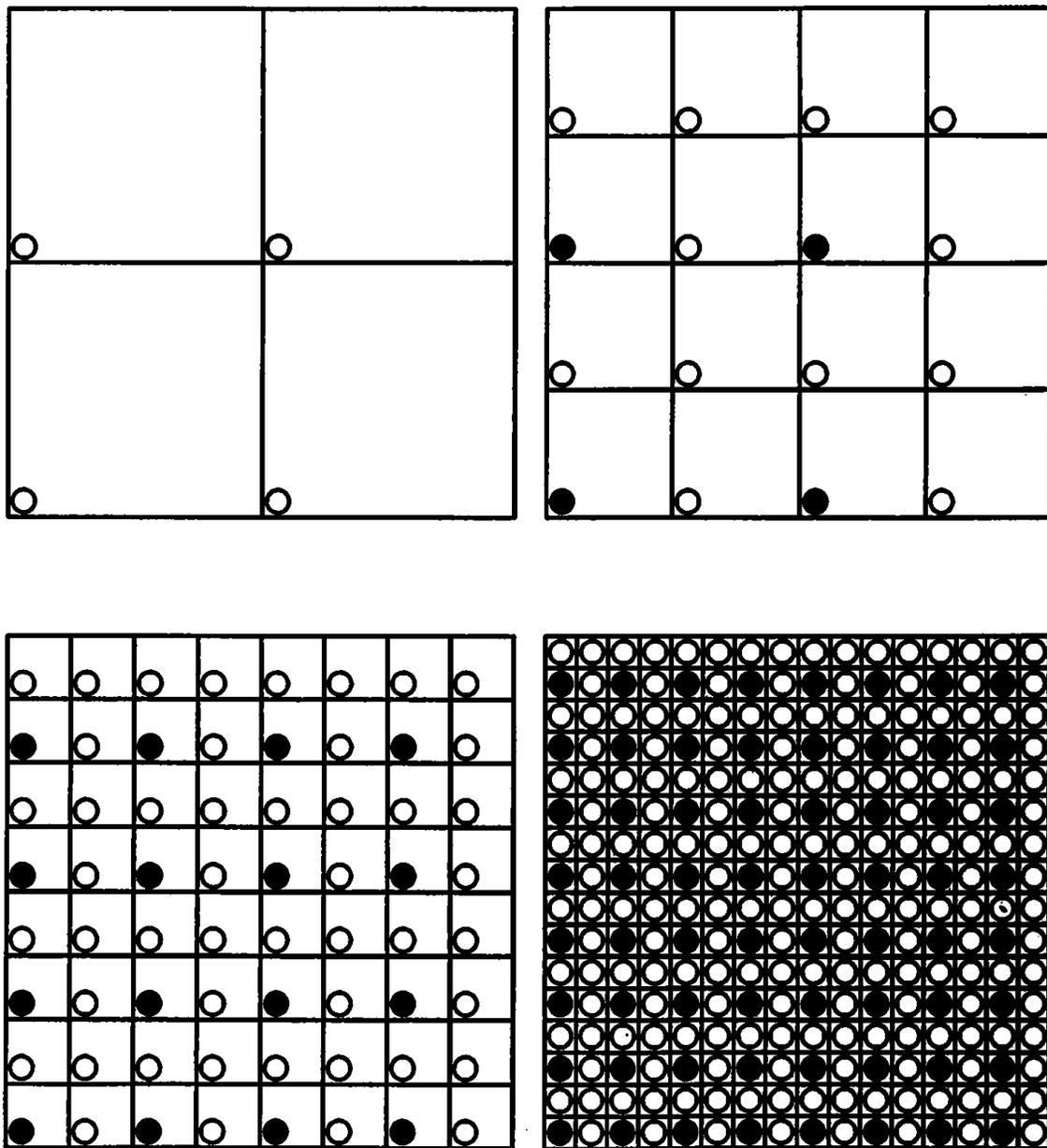
Figure 1. Progressive refinement of the picture elements. White circles denote newly-sampled pixels; black circles denote previously sampled pixels.

three new quadrants and one whose corner pixel has already been sampled.

3. Repeat step 2 until the regions are subdivided to one pixel per region.

See Fig. 1 for an illustration of the subdivision process. The white circles denote newly sampled pixels, and the black circles denote previously sampled pixels.

Note that, for the purposes of illustration, the [0,0] pixel is in the lower left corner. If the display device uses inverted Y coordinates, the algorithm is unchanged.

The algorithm's first step is to lay down the initial regions over the image area. Since the regions will be subdivided down to a single pixel, the initial region dimensions must be a power of two. In addition, the initial regions must evenly tile the display surface. For example, if the display resolution is 768×1024, then the largest possible initial region size will be 256×256 pixels (this may not be a good choice, however, as will be discussed later).

When the resolution doesn't allow for an initial coarse gridding, e.g., $1,152 \times 900$, then there are two approaches you can take. One possibility is to just use the largest region that will fit evenly (4×4 pixels in this case). Another approach would be to tile the image with larger regions, and then to draw the remaining scanlines using regular scanline sampling.

When selecting the size of the initial tiling, you should consider several points. First of all, you want to select an initial display that will present a reasonable preview of the scene. For example, if you selected an initial region size of 256 pixels for the 768×1024 display, the initial resolution would be 3×4 picture elements—not much of an image preview. In this case an initial region size of 32×32 pixels would be much better (an initial resolution of 24×32 picture regions).

You also want to choose an initial resolution that's large enough to display a coarse screen in a "reasonable" amount of time. Obviously this is up to the user, but if an image takes two hours to generate, then it would take about half an hour to display the first screen if the initial region size were 2×2 pixels. Selecting a larger initial region size, such as 8×8 pixels, would bring up an initial screen in less than two minutes.

One last consideration is an artifact caused by sampling the regions at the corners. Because the color of a given region appears to "shrink

down" to its corner pixel, the image can appear to creep toward the origin for successive passes. Fortunately this effect is largely overwhelmed by other artifacts of coarse sampling, such as aliasing, and is mostly noticeable when the picture regions are large (more than about 16 pixels on a side) and when successive passes occur quickly (in several seconds or less). If you wish to ameliorate this effect, keep the size of the initial regions down to around 8×8 or less. For images that take longer to display, the effect won't be noticeable.

See also G1, 265; G2, 295.

VII.6
ACCURATE POLYGON SCAN CONVERSION USING HALF-OPEN INTERVALS

Kurt Fleischer
California Institute of Technology
Pasadena, California

and

David Salesin
Cornell University
Ithaca, New York

Introduction

Here is a simple, efficient polygon scan-conversion algorithm that allows a mesh of polygons to be scan-converted, one polygon at a time, without drawing any pixel twice, and without leaving any holes.

The algorithm uses a slightly modified Bresenham algorithm (Bresenham, 1965) to scan-convert the polygon edges, and uses half-open intervals[1] to disambiguate the pixels lying on edges shared by adjacent polygons. For clarity, we exhibit here a scanline algorithm for triangles; however, the technique is readily extended to arbitrary simple polygons.

Derivation

Suppose we are given a planar subdivision S—that is, a division of the plane into (polygonal) faces, edges, and vertices, with no edge crossings. We would like to find a consistent way of mapping points in the plane to the faces of the subdivision so that every point (including those on vertices and edges of S) maps to a single face of S.

We define the edges and faces of S to be open sets. By choosing a Cartesian coordinate system in the plane, we can map every element of S

---

[1]The half-open interval $(a, b]$ is the set of points $\{x \mid a < x \leq b\}$.

to a single face of *S* as follows:

• Map every face to itself.

• Map every horizontal edge to the face above it; map every other edge to the face on its left.

• Map every vertex that is the right endpoint of a horizontal edge to the same face as that edge; map every other vertex to the face on its left.

Since the plane is the union of all faces, edges, and vertices of S, these rules define a mapping from every point in the plane to a particular polygonal face *P* of *S*. Thus, a scan conversion algorithm that paints exactly those pixels whose centers are mapped to a given polygon *P* will be able to render all the polygons of a polygonal mesh, one at a time, without leaving any holes or duplicating any pixels.

Algorithm

The following algorithm scan-converts a polygon *P*, painting exactly those pixels whose centers are mapped to *P*.

Let *EdgeScan*(*e*) be a function that, given an edge *e*, returns the *x*-coordinate of the rightmost pixel whose center lies on or to the left of *e* for a particular scanline. Let *LeftEdge* and *RightEdge* be left and right edges of a polygon *P*. At each scanline we compute

$$xmin \leftarrow EdgeScan\ (LeftEdge)$$
$$xmax \leftarrow EdgeScan\ (RightEdge)$$

and draw every pixel on the half-open interval (*xmin, xmax*].

Clearly, the algorithm paints exactly those pixels on scanline *y* whose centers are mapped to polygon *P*. Furthermore, if we repeat this step for every scanline on the half-open interval (*ymin, ymax*], where *ymin* and *ymax* are the *y*-coordinates of the topmost and bottommost pixels of *P*, then we will paint exactly those pixels whose centers are mapped to *P*.

Here is pseudo-code for scanning two edges *ab* and *ac,* starting at the apex *a:*

```
LeftEdge ← EdgeSetup(a.x, a.y, b.x, b.y)
RightEdge ← EdgeSetup(a.x, a.y, c.x, c.y)
for y = a.y + 1 to min{b.y, c.y} do
    xmin ← EdgeScan(LeftEdge)
    xmax ← EdgeScan(RightEdge)
    for x = xmin + 1 to xmax do
        DrawPoint(x, y)
    end for
end for
```

For efficiency, *EdgeScan* is computed incrementally using a Bresenham-like algorithm. The *EdgeSetup* routine initializes a structure that contains the increments, and the *EdgeScan* routine uses these increments to compute the intersection of the edge with each scanline.

Note that a necessary consequence of painting only those pixels whose centers map to a polygon is that the algorithm may draw disconnected sets of pixels for very thin polygons, and draws no pixels at all for polygons that are degenerate such as line segments or single points.

Scan-Converting the Edges

All we need now is a good implementation of the *EdgeSetup* and *EdgeScan* routines. We use a slight modification of Bresenham's classic line-drawing algorithm.

Recall that for a line *l* with slope greater than 1, the Bresenham algorithm scans in *y*; paints the nearest pixel (b_x, y), such that $-\frac{1}{2} < b_x - l_x \leq \frac{1}{2}$, where l_x is the actual intersection of line *l* with scanline *y*; and increments b_x by either 0 or 1. The classic Bresenham algorithm treats lines with slope less than 1 in the same fashion, with the roles of *x* and *y* reversed.

Our *EdgeSetup* and *EdgeScan* routines implement a Bresenham algorithm with the following changes.

First, our algorithm treats every edge alike—regardless of slope—by scanning only in *y*. Thus, just a single pixel is generated for each edge at each scanline.

Second, to handle edges with slope less than 1, the algorithm must compute an increment i in the *EdgeSetup* routine, and increment b_x by either i or $i + 1$ as each pixel is drawn (instead of incrementing by 0 or 1). The increment i can also be used for edges with negative slopes.

Finally, since our algorithm must compute pixels whose x-coordinates e_x satisfy $0 < e_x - l_x \leq 1$, we must use the conversion $e_x = b_x - \frac{1}{2}$. Thus, the pixels drawn by *EdgeScan* for a given edge (of slope greater than 1) are the same as those drawn by the Bresenham algorithm for the same edge, shifted left by half a pixel.

The appendix gives code for scan-converting triangles with integer coordinates, as well as an extension for triangles whose vertices are defined to sub-pixel resolution (Fleischer, 1991). The two versions of the algorithm use different *EdgeSetup* routines, but share the same *EdgeScan.*

See also G1, 76; G1, 84; G1, 87; G1, 92.

VII.7

DARKLIGHTS

Andrew S. Glassner
Xerox PARC
Palo Alto, California

Every image designer knows that appropriate illumination is an important part of an effective image. The play of light on the surfaces of the scene gives the image depth and mood, indicates position and character, and reveals both objective visual information and suggests subjective emotional impressions. Even objective physical rendering, when used in fields such as scientific visualization, requires sensitive judgement for the placement and control of illumination.

The interplay of light and surface is a subject; that rewards close study. Our purpose in this note is to make better known a tool that is used subconsciously by many painters, and quite explicitly by many graphics designers.

The idea is actually something that people have wanted for a long time. I recall reading a comic book in the late 1960s, in which a superhero caught a bank robber in the act of emptying a vault in the middle of the day. To thwart the hero, the villian turned on an "anti-flashlight" and pointed it at the hero, immersing the good guy in a cone of darkness. The villian got away because the hero couldn't see. In real life it is unclear how to build such an anti-flashlight, but in computer graphics we work with a more general physics that provides the facility naturally.

The mechanism is quite simple. First, note that to create a brightly illuminated piece of surface, one directs onto that surface a light source. This source sends illumination energy to that surface, in addition to any light arriving from other sources.

Alternatively, suppose you wish some piece of surface to appear darker. One approach is to add lights everywhere else in the scene so that the desired surface is relatively darker. A simpler approach is to illuminate

the surface with *negative light.* This is easy: Simply create a light source whose color is defined by negative numbers. I call such lights *darklights.* Usually there is nothing in a rendering program to prevent you from assigning any numbers you want to the color of a light source—if a program does prevent you from using negative numbers, it's a bug, not a feature: Negative sources make syntactic and semantic sense, and should be available in every rendering system.

As an example application, suppose you have a kitchen scene with a complex lighting arrangement. But there's one spot on a countertop that appears too bright—it distracts the eye from where you really want the viewer to look. You could change all the lighting, but suppose that you've spent some time getting the illumination correct in the area of interest, and it's just bad luck that there happens to be a bright spot in the wrong place. Simply create a spotlight source with negative coefficients and direct it to the surface. The rendering program will add the light from all other sources, and subtract the light from the darklight, thereby darkening the patch of surface.

To see how this gets implemented, suppose you have a standard Phong lighting equation (for illustration, we'll just write the diffuse component, ignoring transparency, Fresnel effects, etc.):

$$I(\lambda) \,=\, k_{\rm d}(\lambda) \sum_{n=0}^{\text{lights}} I_n(\lambda)(\mathbf{L}_n \cdot \mathbf{N}).$$

To create a darklight, simply create a light with negative energy; that is, some (or all) of the components of $I_n(\lambda) < 0$. If the colors are represented simply by three components (typically near monitor red, green, and blue phosphors), these end up less than zero.

Darklights can cause the final pixel values to dip below zero. This is a problem, but it's nothing new. It is well known that the dynamic range of brightness that can be displayed on monitors is far less than that which can be computed for real scenes. Typically the colors of every image must be compressed in some way to fit onto a monitor. The simplest approach is to simply clamp color components greater than one to one; alternatively, one may simply compress or expand the synthesized range of colors into the displayable range. A slightly more sophisticated solu-

tion desaturates the offending colors until they are within gamut. Good color gamut mapping is still an art. Whatever techniques are applied to map pixel values greater than one into gamut should also be applied to pixel values that dip below zero.

Just as with all lights, darklights must be designed and positioned with care to achieve an effect. They are useful for simulating fuzzy shadows, darkening up corners of rooms, and changing the relative brightness of objects without affecting the basic lighting scheme. They are naturally available in virtually all rendering systems. Used with care and sensitivity, darklights can extend your expressive power with the medium of image synthesis.

VII.8

ANTI-ALIASING IN TRIANGULAR PIXELS

Andrew S. Glassner
Xerox PARC
Palo Alto, California

Most graphics rendering today takes place in a square pixel grid. This is because the boundaries of the square grid are all vertical and horizontal lines, which eases the computational burden of sampling, filtering, and reconstruction.

But it is well known that the square grid is not the most uniform in terms of sampling densities. In two dimensions, the triangular lattice shown in Fig. 1 is more isotropic. This lattice permits two tilings by regular polygons: hexagonal and triangular. Both unit cells have been used for image processing, and to a lesser extent, computer graphics (note that the arrangement of phosphor triples on the inside of a CRT monitor is a lattice like that of Fig. 1). In this note we present an inexpensive anti-aliasing technique for triangular pixels. Figure 2 shows a tiling of the plane by equilateral triangles, and a polygon passing through each triangle. We can parameterize the prefiltered contribution of the polygon to each element of the grid by building a look-up table based on the vertices of the polygon and the polygon's intersections with the edges of the triangle. A simple box filter simply returns the amount of area within the triangle; higher-order filters may apply a weighting mask (e.g., a Gaussian) first.

Our analysis is inspired by the square-pixel anti-aliasing technique of Abram, Westover, and Whitted (1985).

There are two cases to consider: no vertices in the triangle (the $V0$ case), and one vertex in the triangle (the $V1$ case). The side length of the triangle is 1. We take the $V0$ case first.

Figure 3 shows the two possible $V0$ configurations. In 3a, the area to be counted is the shaded triangle. We identify the two intersection points
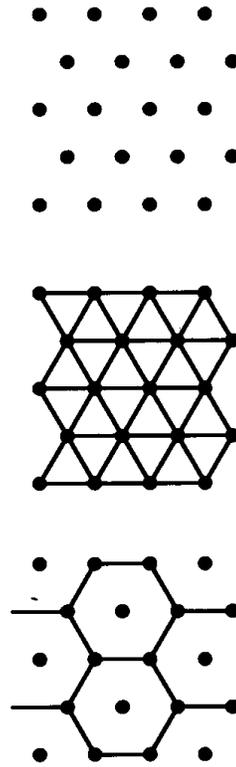
Figure 1.

by their distances α and β from the surrounded vertex. Without loss of generality, we label the coordinates $(\alpha, 0)$ and $(\beta/2, \beta\sqrt{3}/2)$. Since $A = \frac{1}{2}bh$, we find

$$A = \alpha\beta\frac{\sqrt{3}}{4} \quad \text{(Case } V0, \text{ Convex)}$$
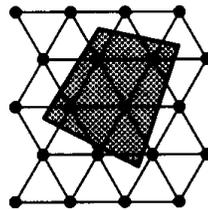
Figure 2.
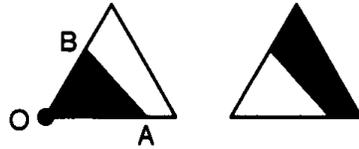
Figure 3.

The area in Fig. 3b is just the area of the triangle ($A = \sqrt{3}/4$) minus this amount:

$$A = \frac{\sqrt{3}}{4}(1 - \alpha\beta) \quad \text{(Case } V0, \text{ Concave)}.$$

This concludes type $V0$.

The $V1$ case comes in three types, called Types 1, 2, and 3, illustrated in Figs. 4, 5, and 6. Note that Types 2 and 3 each have the two configurations we saw for the $V0$ case.
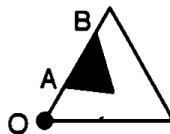
Figure 4.
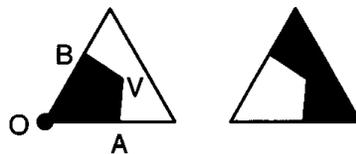
Figure 5.
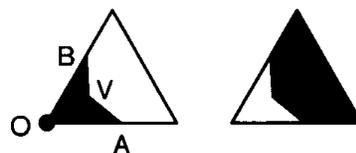
Figure 6.

We start with Type 1. In Fig. 4, the two edges of the polygon intersect the triangle on the same side, at distances α and β from the origin. These points have coordinates

$$\left(\frac{\alpha}{2}, \frac{\alpha\sqrt{3}}{2}\right), \qquad \left(\frac{\beta}{2}, \frac{\beta\sqrt{3}}{2}\right), \qquad \text{and} \qquad \left(V_x, V_y\right).$$

We apply the more general triangular area formula

$$A = \tfrac{1}{2}(x_1 y_2 + x_2 y_3 + x_3 y_1 - y_1 x_2 - y_2 x_3 - y_3 x_1).$$

Setting the point at α to point 1, the point at β to point 2, and V to point 3, and simplifying, we get

$$A = \frac{\alpha - \beta}{4}\left(V_x\sqrt{3} - V_y\right) \qquad \text{(Case V1, Type 1)}.$$

This concludes Type 1.

Type 2 is illustrated in Fig. 5. This type is identified by a single vertex V in the triangle (thus case V1), the two polygon edges pass through two triangle edges, and the vertex V is outside the triangle formed by the enclosed triangle vertex and the points at $\cdot\alpha$ and $\cdot\beta$. We find the area by first finding the area of the triangle (V, α, β), and then adding the area of triangle (α, V, β). As we saw from case V0, the area of the "inner" triangle (V, α, β) is

$$A_i = \frac{\sqrt{3}}{4}\alpha\beta.$$

Using the triangle area formula above, we find that the area of "outer" triangle (α, V, β) is

$$A_O = \tfrac{1}{4}(\alpha\beta - \beta V_y\sqrt{3} - \beta V_x - 2V_y\alpha).$$

The area of the complete quadrilaterial is thus

$$A = A_i + A_O \qquad \text{(Case V1, Type 2, Convex)}.$$

Note that for the concave case, we subtract this area from the triangle:

$$A = \frac{\sqrt{3}}{4} - (A_i + A_o) \qquad \text{(Case } V1, \text{ Type 2, Concave).}$$

This ends Type 2.

Type 3 follows much the same argument, only we subtract the triangle (V, α, β) from the other:

$$A = A_i - A_o \qquad \text{(Case } V1, \text{ Type 3, Convex),}$$

$$A = \frac{\sqrt{3}}{4} - (A_i - A_o) \qquad \text{(Case } V1, \text{ Type 3, Concave).}$$

This concludes the case analysis. Note that in Case V1, Types 2 and 3 Concave find the area of pentagonal regions by finding the complementary quadrilateral within a triangle, and decomposing that quadrilateral into triangles. This decomposition is a general result for this situation.

We consider polygons that include more than one vertex in the triangle to be sufficiently rare, and sufficiently hard, that traditional (and more expensive) area-estimation techniques such as low-resolution scan conversion and analytical area computation should be considered, rather than building many more case analyses.

These techniques are also applicable to text filtering, drawing thick lines, and other scan-conversion operations with triangular pixels.

This note uses included areas as an anti-aliasing measure. This implies that we are using a box filter. A box filter is better than no filter at all, but it is not ideal. One could extend this technique by applying the ideas of Abram, Westover, and Whitted to build precomputed anti-aliasing tables based on parameterized intersections with the triangle edges.

VII.9
MOTION BLUR ON GRAPHICS WORKSTATIONS

John Snyder and Ronen Barzel
California Institute of Technology
Pasadena, California

and

Steve Gabriel
Arvada, Colorado

Lively and dynamic motion can make exciting computer-generating images and animations. Typically, each individual image is rendered at a single instant of time. For a single image of a moving subject, this conveys no sense of motion. For an animation, this can result in extremely objectionable strobing (time-aliasing) artifacts.

Aliasing in time can be reduced by motion blur, that is, by averaging or filtering images at many instants of time to create a result in which moving objects are blurred. Motion blur techniques for ray tracing are relatively well understood, involving the association of different time values for each ray cast (Cook *et al.,* 1984). This paper presents a motion blur technique suitable for graphics workstations having fast z-buffer rendering. The essential result is that fast, high-quality motion blur is achieved simply by

1. supersampling in time using a temporal box filter, and

2. computing images on fields, rather than frames, for video animations.

We will explain these concepts further in the next sections, along with some "tricks" to improve speed and quality on typical graphics workstations. Plate 1 (see color insert) shows the results of the techniques described in this paper.

Supersampling in Time

To create an image with motion blur, the basic method is to average n images, each rendered at a single instant of time using the z-buffer hardware. The images are rendered at equally spaced times, t_i,

$$t_i = t_0 + \frac{i\Delta}{n}, \qquad i = 0, 1, ..., n - 1,$$

where t_0 is the start time of the image, and Δ is the interval during which images are averaged, analogous to the interval during which the shutter is open in a camera. In our animation experience, good results are achieved with Δ equal to the entire time interval between frames. The final frame, I, is the average of the n time-sampled images $I(t_i)$:[1]

$$I = \frac{1}{n}\sum_{i=0}^{n-1} I(t_i).$$

We will use the term *subframe* for a time-sampled image, $I(t_i)$.

The technique of rendering subframes at equally spaced time intervals, called uniform sampling, caused strobing artifacts if n is too small, or the temporal frequencies too high. Some of these artifacts can be made less objectionable, without increasing n, by using stochastic sampling. A simple technique that works well is to compute each subframe at time

$$t_i = t_0 + \frac{(i + \delta)\Delta}{n},$$

where δ is a random variable with uniform distribution over [0, 1]. By stochastically perturbing the rendering time for each subframe, we effectively transform aliasing into less objectionable noise (Cook, 1986). For example, if a body rotates an integral number of times between each subframe, then uniform sampling yields no motion blur, whereas stochastic sampling yields a more acceptable, blurred result.

Computing on Fields

On video displays, all scan lines of a single frame are not presented simultaneously. Instead, scan lines are displayed in an interleaved fash-

---

[1]One might consider more sophisticated filtering than averaging (box filtering), but we have found that box filtering works well, and at a small computational cost.

ion, called *interlacing*: first, lines 1, 3, 5, and so on (called the first *field),* followed by lines 0, 2, 4, and so on (called the second field). Therefore, temporal antialiasing is much enhanced by "computing on fields"—i.e., by computing n subframes, the first $n/2$ of which are averaged to create the scan lines of the first field, and the second $n/2$ of which are averaged to create the scan lines of the second field.[2]

Thus, if F_1 and F_2 are the first and second field images, respectively, then

$$F_1 = \frac{2}{n} \sum_{i=0}^{n/2-1} I\left(t_0 + \frac{(i+\delta)\Delta}{n}\right),$$

$$F_2 = \frac{2}{n} \sum_{i=n/2}^{n-1} I\left(t_0 + \frac{(i+\delta)\Delta}{n}\right).$$

The scan lines of the computed field images, F_1 and F_2, are then interleaved to create the resulting frame. That is, the kth scan line of the resulting frame, I_k, is given by scan lines F_{1k} and F_{2k} via

$$I_k = \begin{cases} F_{1k} & \text{if } k \text{ is odd} \\ F_{2k} & \text{if } k \text{ is even} \end{cases}$$

Note that the even scan lines of F_1 and the odd scan lines of F_2 are never used and need not be computed. Computing on fields can therefore save computation time on graphics workstations because we average only half the scan lines of each subframe. Of course, the same total number of subframes must still be rendered.

Combining Spatial and Temporal Anti-aliasing

Most workstation z-buffer hardware lacks spatial anti-aliasing capability; rendering images have "jaggies." One way to reduce aliasing is to render

---

[2]We assume n is even.

Table 1.   Pixel shifts yielding a box filter in x and y.

Time	Field 1				Field 2			
	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
x shift	$+\frac{1}{4}$	$+\frac{1}{4}$	$-\frac{1}{4}$	$-\frac{1}{4}$	$+\frac{1}{4}$	$+\frac{1}{4}$	$-\frac{1}{4}$	$-\frac{1}{4}$
y shift	$-\frac{1}{4}$	$+\frac{1}{4}$	$+\frac{1}{4}$	$-\frac{1}{4}$	$-\frac{1}{4}$	$+\frac{1}{4}$	$+\frac{1}{4}$	$-\frac{1}{4}$

each image at high resolution and filter it down to lower resolution. This can work well, but requires costly processing to do the filtering.

Given that the image is to be motion-blurred anyway, we can combine temporal anti-aliasing with spatial anti-aliasing of static parts of the image, at no extra computational cost. The idea is to slightly displace each of the subframes spatially before they are averaged. For example, consider an image in which four subframes are rendered for each frame (two for each field). If we displace subframe 0 by $-\frac{1}{4}$ pixel in x, and subframe 1 by $+\frac{1}{4}$ pixel in x before averaging, and likewise for subframes 2 and 3 of the second field, then static parts of the image will be spatially anti-aliased in x as though they had been computed at a higher x resolution and averaged, but without the cost of high-resolution to low-resolution filtering.[3, 4, 5] With eight subframes (four per field), we can achieve a box filter in both x and y, using the shifts from Table I. With 32 subframes, we can achieve a triangle filter in x and y, using the shifts from Table II.

The pixel displacements need not be tied to a rectangular grid as in the tables. Better results can be obtained using a Poisson distribution of displacements (see Barkans, 1991).

---

[3]It is important to note that only static parts of the image are spatially anti-aliased, not moving parts. But moving parts are motion-blurred anyway, and hence require less spatial anti-aliasing.

[4]Correlating spatial and temporal sampling can produce sampling artifacts if the animation contains high spatial and temporal frequencies. We have not encountered difficulties in practice.

[5]Many graphics workstations do not directly support subpixel displacement of an image. The section on implementation tricks describes a trick to implement this technique anyway.

Table 2. Pixel shifts yielding a triangle filter in x and y. The shifts for the second field are identical.

							Field 1									
Time	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}
x shift	$-\frac{1}{2}$	$-\frac{1}{2}$	0	0	0	0	$+\frac{1}{2}$	$+\frac{1}{2}$	$-\frac{1}{2}$	$-\frac{1}{2}$	0	0	0	0	$+\frac{1}{2}$	$+\frac{1}{2}$
y shift	$-\frac{1}{2}$	0	0	$-\frac{1}{2}$	$-\frac{1}{2}$	0	0	$+\frac{1}{2}$	$+\frac{1}{2}$	0	0	$+\frac{1}{2}$	$+\frac{1}{2}$	0	0	$..$

Reducing Interlace Artifacts

Flicker is a well-known problem with interlaced video. In a still image, a one-pixel wide horizontal line colored differently than adjacent scan lines will flicker. For a moving image computed on fields, any narrow, high-contrast, quickly moving lines or edges will appear to break apart and look jaggy or blocky.

This problem can be solved by filtering each field image (F_1 or F_2 from earlier) in y before discarding alternate scan lines. We have had good results with a five-pixel wide filter having the coefficients

$$-\frac{1}{8}, \ \frac{2}{8}, \ \frac{6}{8}, \ \frac{2}{8}, \ -\frac{1}{8}.$$

That is, the kth scan line of the result field image, F'_k, is computed from five scan lines of the original field image, F_{k-2} through F_{k+2}, via[6]

$$F'_k \ = \ \tfrac{1}{8}F_{k-2} \ + \ \tfrac{2}{8}F_{k-1} \ + \ \tfrac{6}{8}F_k \ + \ \tfrac{2}{8}F_{k+1} \ - \ \tfrac{1}{8}F_{k+2}.$$

The two resulting field images should be interleaved to create a final frame as discussed in the section on computing on fields. This filter allows scan lines that would be discarded to contribute to the resulting field scan lines, without undue blur. Interlace problems are much reduced.

As in the earlier section, we need only compute the even or odd scan lines of the result field images, F'. However, since the filter requires all

[6]Scan lines off the top (bottom) of the original image can be replicated from the top (bottom) scan line.

the original field image's scan lines to compute the result field image, the motion blur accumulation must be performed on all scan lines. Thus, the time savings for computing on fields does not apply when using this anti-flicker filter.

Other Effects

Motion blur can be thought of as the result of integrating a time-dependent image over a range of time values; our method of averaging subframes computes a simple approximation of this integral. Other rendering effects, such as diffuse shadows and depth of field, similarly depend on integrating the image over a range of parameter values (see Cook *et al.,* 1984). We can achieve such effects by choosing different rendering parameters for each subframe, allowing the subframe averaging to perform the integration.[7]

For example, by varying the position of the light source slightly with each subframe, we achieve diffuse shadows and lighting, simulating an area light source. In addition, since our graphics hardware lacks the ability to render transparent shadows,[8] we render some fraction of subframes with shadows and the rest without, in order to achieve transparent shadows rather than completely black ones. Plate 1 shows diffuse and transparent shadows computed this way. Depth-of-field effects can be achieved similarly by varying the camera model parameters with each subframe.

To choose the parameter values (e.g., the position of a light source) for each subframe of an animation, it is best to precompute a table of values so that the same sequence of parameter values is used in each field (or frame). This produces better results than stochastically generating a different sequence of parameter values for each field (or frame), which would make the animation flicker from frame to frame.

---

[7]As stated in Section 3, correlating time and rendering parameters can cause artifacts. In most cases, increasing the number of subframes reduces these artifacts.

[8]We render shadows by projecting objects onto a floor or ground plane with an appropriate 4×4 matrix, drawing them in black. Because our graphics hardware lacks the ability to render transparent polygons, these shadows are completely opaque.

Implementation Tricks

We will describe three "tricks" that can be used in implementing motion blur:

- achieving spatial antialiasing on graphics hardware that lacks subpixel positioning
- using pixel displacement arrays to increase spatial antialiasing as the number of subframes increases
- using double buffering or video-resolution rendering to speed accumulation of each subframe

The discussion in the section on combining temporal and spatial antialiasing assumes the workstation graphics can do subpixel positioning —i.e., shifting an image by a fraction of a pixel results in a different image. Not all workstations support subpixel positioning. For instance, the Starbase graphics library on the HP9000 Series 800 workstation with the HP 98731 (Turbo SRX) graphics accelerator and z-buffer, and the GL graphics library on the Silicon Graphics 4D 80GT workstation do not support subpixel positioning. These graphics systems convert polygon vertices to pixel coordinates before tiling; only shifting by entire pixel amounts results in a change in the resulting image.

Fortunately, we can mimic subpixel positioning by taking advantage of our workstation's high-resolution rendering (i.e., approximately 1,280 by 1,024 pixels). Although we can't shift by half pixels, we can render at twice video resolution, so that a shift by one high-resolution pixel corresponds to a shift by one-half of a video-resolution pixel. We can then achieve the image displacements required by Table II. In converting the high-resolution image to video resolution, we simply ignore samples not falling on a video resolution sampling grid. That is, we sample only every other pixel of every other scan line, for computation on frames, and every other pixel of every fourth scan line, for computation on fields (see Fig. 1). Since we examine only as many pixels as needed for video resolution, there is no extra averaging or filtering cost compared with rendering directly at video resolution without antialiasing.[9] Thus, given that we are

[9]The cost due to rendering at high resolution rather than video resolution is minimal on most z-buffer hardware, where rendering time depends mostly on the number of polygons rendered, not their screen size.

frame pixel grid   field pixel grid
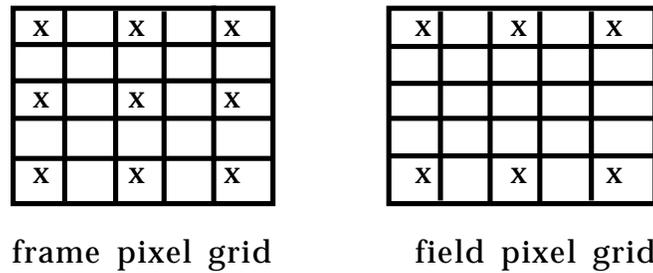
Figure 1. Sampling grid for spatial anti-aliasing. The x's indicate pixels that are sampled; the pixels without x's are ignored.

supersampling to produce motion blur, we can spatially antialias almost for free.

The pixel shifts can be implemented by creating an array of offsets for x and y, which are indexed by the subframe number. A 2-D image translation transformation using the resulting x and y offset is then applied before the subframe is rendered (see the mb_frame routine in the included code). The offsets are easily arranged so that computing more subframes yields better spatial antialiasing. Table III shows the relation between the number of subframes per field and spatial antialiasing produced by the offset arrays in the mb_frame routine. We note that this procedure can cause a global shift of the entire image by a small fraction of a pixel. For example, the eight subframes per field case shifts the entire image left by one-fourth of a pixel. This is of little consequence if the number of subframes per field is constant through the animation.

Cached memory access on most workstations means that two passes of a simple computation over the frame buffer is often slower than a single pass of a significantly more complex computation. On workstations that implement double buffering by dividing each pixel of the frame buffer into low- and high-order halves (such as the HP9000 Series 800 Turbo SRX),

Table 3. Relationship between number of subframes per field and resulting kind of spatial antialiasing filter, in the mb_frame routine

Subframes/Field	Spatial Anti-aliasing
1	none
4	2 sample box filler in x and y
8	3 sample triangle filler in x, 2 sample box filter in y
16	3 sample triangle filter in both x and y

we can take advantage of this behavior. We render two subframes into the frame buffer, the first in the low order half, the second in the upper. This sacrifices half the color resolution, but allows both subframes to be accumulated into the final image with a single frame buffer pass, by using a few extra shifts and adds. On the HP Series 800 workstation, this technique speeds up the accumulation process by nearly a factor of two.

If we are willing to sacrifice spatial anti-aliasing, we can get an additional speed-up. Since the hardware frame buffer has twice the resolution of video both horizontally and vertically, we can render four subframes at video resolution into the four quadrants of the framebuffer, and accumulate them all in a single pass. Combining video-resolution rendering with double-buffering allows us to accumulate eight subframes in a single pass. This technique sacrifices half the color resolution and all spatial anti-aliasing, but it is our fastest motion-blur method. We often use these quick and dirty motion-blur methods for motion tests, and the slower methods for final results.

The technique of squeezing as much computation as possible into a single pass over the data can be applied to other processing that may need to be done to the image before it is recorded, such a conversion from RGB to YUV, filtering, or indirection through the color map. It is most efficient to do all such processing during the last accumulation pass, rather than making further passes over the frame buffer or accumulation buffer memory. Unfortunately, this requires special-purpose code for every combination of postprocessing procedures.

Examples

Plate 1 (color insert) shows images rendered using the motion blur technique. See the Appendix for a sample implementation.
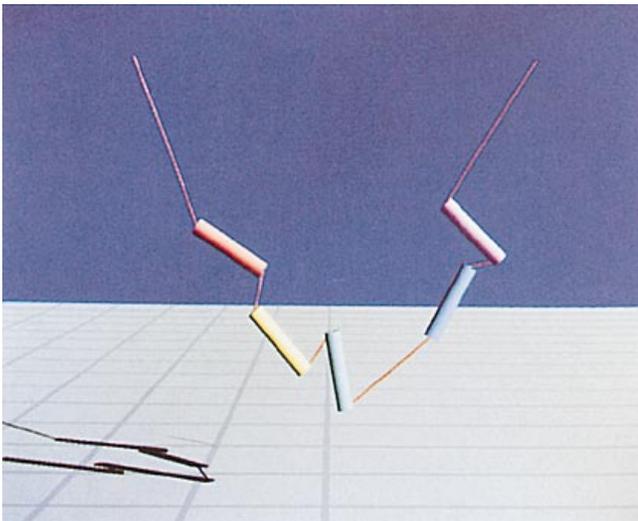
A. Cylinders without Blur
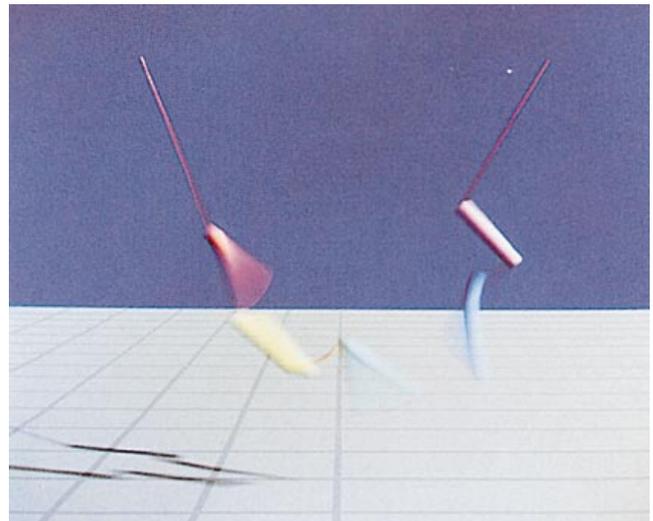
B. Cylinders with Blur
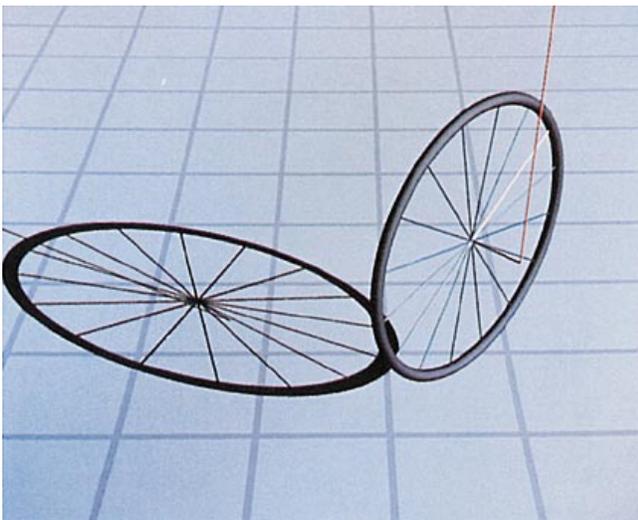
C. Wheels without Blur
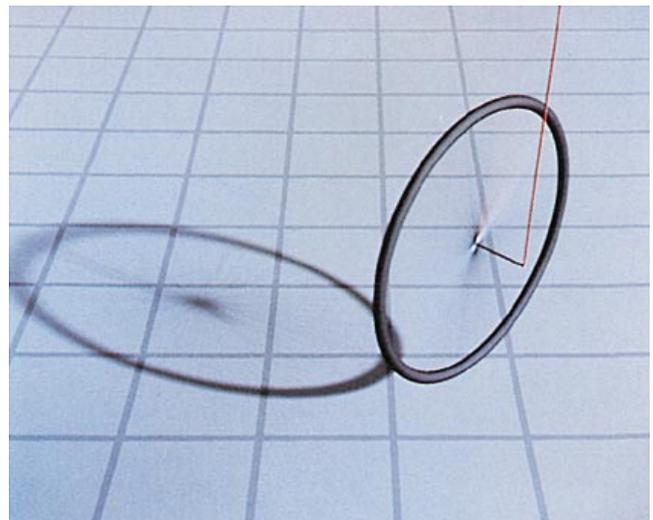
D. Wheels with Blur

7.9 Plate 1. This plate illustrates motion blur as described in "Motion Blur on Graphics Workstations." The pictures on the left are individual time samples rendered using the z-buffer hardware; the pictures on the right are the result of averaging many time samples. Note the motion Blur as well as the diffuse, transparent shadows in the pictures on the right. On the top, a chain of cylinders linked by rubber bands swings from two anchor points. On the right bottom, a bicycle wheel suspended from the ceiling rotates roughly 45° during the shutter open interval. The cylinder image was computed by averaging 64 time samples; the bicycle wheel image used 256 samples.

VII.10
THE SHADER CACHE:
A RENDERING PIPELINE
ACCELERATOR

James Arvo   Cary Scofield
Cornell University   and   Hewlett-Packard
Ithaca, New York   Chelmsford, Massachusetts

Introduction

This Gem describes a caching strategy to accelerate shading calculations in any traditional polygon rendering pipeline, such as that defined by PHIGS + . The purpose of the cache is to reduce the number of redundant shading calculations performed on behalf of adjacent polygons sharing at least one vertex and associated surface normal. Such polygons frequently result from applications that tesselate trimmed or implicit surfaces. Because implementations of these algorithms often do not perform the necessary bookkeeping to record polygon adjacency, large numbers of autonomous polygons may be emitted in lieu of more efficient organizations such as a triangle strips or quad meshes. Given that some applications will always choose to generate streams of independent polygons, we wish to do what we can "on the fly" to mitigate the cost of such redundancy without affecting the performance in other circumstances.

  Our approach is to augment the shading module with a cache to store the results of recently performed shading calculations. Then, with reasonable probability, these results can be reused for adjacent polygons that share one or more vertices with identical data and attributes. If we view the shader as a function that computes a color based on the position P, normal N, and color index k of a polygon vertex, that is

$$\text{color} \leftarrow \text{Shader}(P, N, k), \qquad\qquad (1)$$

then the role of the cache is to mimic this behavior based on a table
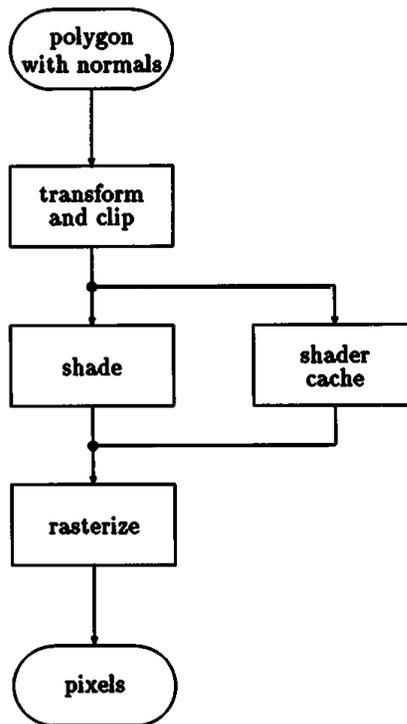
Figure 1. A flow chart showing where the shader cache fits into the rendering pipeline.

lookup, deferring to the actual shader only when the appropriate entry is not found. Thus, each cache entry associates a color with a complete set of shader input arguments. These entries must be indexed efficiently so that lookups can be done with a minimum of overhead.

The logical arrangement of the cache is shown in Fig. 1. A shader request is intercepted and serviced by the cache ii' this same request has been issued recently on behalf of another polygon and the result is still resident in the cache. The cost of each lookup can be reduced by means of a hash function and multilevel hit validation as described later in the section on implementation. This caching scheme does not require any reorganization of drawing elements in the display list. It automatically takes advantage of a common form of redundancy whenever it is present in the data, and it can be employed with no modification to the rest of the rendering pipeline.

Shading Calculations

In this section we examine the cost of a shading calculation to get an idea of what might be gained by economizing on the number of shader calls. In the very simplest case, the scene may be illuminated by a single "directional" light source, which is equivalent to an infinitely distant source. This requires a calculation of the form

$$C\big[A(N \cdot L) + B(E \cdot R)^s\big], \tag{2}$$

where *A, B,* and *C* are the colors of the polygon, specular highlight, and light source respectively. Even this simple case requires several dot products and an exponent. For spotlights the calculations can be much more costly, involving another exponent for the light concentration (making the light diminish with angle), and the calculation of attenuation (making the light diminish with distance) (van Dam, 1988). Given that there are potentially many light sources, we are faced with a computation of the form

$$\sum_{i=1}^{N} \frac{C_i(D_i \cdot L_i)^k\big[A(N \cdot L_i) + B(E \cdot R_i)^s\big]}{\alpha_1 + \alpha_2\|P - Q_i\|} \tag{3}$$

for each vertex of each polygon. This can add up to a significant amount of work and become the dominant computation. In contrast, the work involved in traversing the display list, transforming vertices, and clipping polygons remains nearly constant. It is easy to see how cases can arise in which saving a few shading calculations will increase the overall throughput of the rendering pipeline.

Implementation

Figure 2 shows how to incorporate the caching strategy into an existing shader. The new shader simply consults the cache before resorting to the more costly computation carried out by the general-purpose shader.

```
procedure NewShader( X, Color )
begin
      First consult the shader cache.
      hit ← Cache_Lookup( X, tag, Color);
      if not hit then begin

            Invoke full shader to compute color
            Shader( x, Color);
            Update the shader cache with the new info
            index ← mask( tag );
            Table[index].tag ← tag;
            Table[index].X ← X
            Table[index].Color ← Color
            end;
      end;
```

Figure 2. A shader that incorporates the caching mechanism.

When a *cache miss* occurs, the newly computed color and associated data is inserted into the cache so that it is potentially available to adjacent polygons. If there are no such polygons, or if this color is overwritten before an adjacent polygon is encountered, then no work has been saved. In no case, however, is an incorrect result generated.

Figure 3 shows how the cache works. It first takes all of the arguments passed to the shader and combines them into a single *hash tag*. A simple yet effective method for doing this is to XOR all the components of all the arguments together. From this hash tag we extract enough bits to index into the cache; for instance, we may use the low-order byte to index into a 256-entry cache.

Next, we need to determine whether or not the given cache entry already contains the result we are looking for. If so, this is a *cache hit*. Hit validation is done in two phases: First we compare the entire hash tag

```
boolean function Cache_Lookup( X, tag, Color)
begin
    Compute the "tag" by xor-ing all the data
    tag ← X₁ xor X₂xor . . . xor Xₙ;
    Compute table index from the low-order bits.
    index ← mask( tag );
    Compare the tags as a quick validation check.
    if Table[index].tag ≠ tag then return False
    Now do a complete cascaded validation check.
    if Table[index].X₁ ≠ X₁ then return False;
    if Table[index].X₂ ≠ X₂ then return False;

        ⋮

    if Table[index].Xₙ ≠ Xₙ then return False;
    A hit, return the previously computed color.
    Color ← Table[index].Color;
    return True;
    end;
```

Figure 3. The shader cache look-up algorithm. The input data, X, will typically include a 3-D point and a normal vector.

to the one generated when the current cache entry was computed. If the cache entry contains the result of a different request, this tag will very likely differ and immediately indicate a cache miss. If the tags match, we then proceed to compare all the actual input arguments in order to verify a hit. This is a multilevel validation scheme, with each level increasing the total cost but also increasing the probability of a hit. If all entries match, we can immediately return the desired color with no further computation.

Note that it is necessary to invalidate the contents of the entire cache when there is a state change that is not reflected in the input arguments to the shader. In particular, this is necessary when any of the light

sources change or the view changes (because specular highlights are view-dependent). The former may occur in mid-traversal if light sources are turned on or off via attributes in the display list.

Effectiveness

When is this strategy likely to be effective? It's easy to analyze expected performance in terms of the dominant costs and the *hit rate* of the cache, although the latter can be very difficult to characterize in advance. Let σ be the total cost of an un-assisted shading calculation and let $\hat{\sigma}$ be the cost of this operation when the shader cache is added. We can think of $\hat{\sigma}$ as a random variable that depends upon the characteristics of the cache and of the display list being processed. Our goal is to make $\hat{\sigma} < \sigma$ on average; otherwise we're better off without the cache. In other words, we want $E(\hat{\sigma}) < \sigma$, where $E(\hat{\sigma})$ is the expected vahle of $\hat{\sigma}$. It is easy to see that

$$E(\hat{\sigma}) = (1 - h)\sigma + c, \tag{4}$$

where *h is* the probability of a cache hit and c is the cost of consulting the cache. Therefore, the cache is advantageous only when $c < h\sigma$. This quantifies the trade-off between the cost of a cache look-up and the likelihood of a cache hit. The quantity h depends upon the size of the cache, the hash function, and the organization of the polygons processed by the pipeline. Clearly, if there are no shared vertices whatsoever, $h = 0$, and the cache can only hurt performance by adding a cost of c to each shading calculation. This cost is quite small, however, so even a very modest hit rate can more than compensate for the time spent consulting the cache. Furthermore, the larger σ is, the smaller the hit rate required to break even.

Results

We have measured the performance of a software rendering pipeline with and without a shader cache on a number of test models. The models were generated by tesselating various surfaces into regular meshes of several

thousand triangles. We have found that with a 256-entry cache using an XOR hash function, even with as few as two directional light sources, the shader cache resulted in a 20% increase in throughput, and with four spotlights this figure increased to 60%. In fairly extreme cases involving 15 spotlights the shader cache increased performance by 250%. In our tests we have found that increasing the cache size beyond 256 entries was of little benefit and, in fact, could even degrade performance on large display lists because of increased paging.

REFERENCES

Abrams, Greg, Westover, Lee and Whitted, Turner (1985). "Efficient Alias-Free Rendering Using Bit-Masks and Lookup Tables," *Computer Graphics* **19**(3), pp. 53–60. (Vll.8 Anti-Aliasing in Triangular Pixels)

Abremowitz, Milton, and Stegun, Irene (1970). *Handbook of Mathematical Functions,* 9th printing. Dover, New York. (III.8 Rigid Physically Based Superquadrics)

Amanatides, John, and Woo, Andrew (1987). "A Fast Voxel Traversal Algorithm for Ray Tracing," *Proceedings of Eurographics '87,* pp. 3–10. (VI.1 Ray Tracing with the BSP Tree)

American National Standard Institute (1986). "Nomenclature and Definitions for Illumination Engineering," ANSI Report, ANSI/IES RP-16-1986. (VI.7 Physically Correct Direct Lighting for Distribution Ray Tracing)

Arnaldi, Bruno, Priol, Thierry, and Bouatouch, Kadi (1987). "A New Space Subdivision Method for Ray Tracing CSG Modelled Scenes," *Visual Computer* **3**(2), 98–108. (VI.1 Ray Tracing with the BSP Tree; VI.3 Use of Residency Masks and Object Space Partitioning to Eliminate Ray-Object Intersection Calculations)

Arvo, James (1988). "Linear-Time Voxel Walking for Octrees," *Ray Tracing News* **1**(2). E-mail edition, available under anonymous flp from weedeater.math.yale.edu. (VI.1 Ray Tracing with the BSP Tree)

Arvo, James (1991). "Random Rotation Matrices," *Graphics Gems II* (James Arvo, ed.), pp. 355–356. Academic Press, Boston. (III.4 Fast Random Rotation Matrices; III.6 Uniform Random Rotations)

Arvo, J., and Kirk, D. (1989). "A Survey of Ray Tracing Acceleration Techniques," *An Introduction to Ray Tracing* (A. Glassner, ed.), pp. 201–262. Academic Press, Boston. (VI.6 A Linear-Time Simple Bounding Volume Algorithm)

Bagby, Dave (1984). "Parameterization of Elliptical Elements," letter to ANSI X3H3 Committee (August 16, 1984), p. 17. (IV.1 A Parametric Elliptical Arc Algorithm)

Barkans, Anthony (1991). "Hardware-Assisted Polygon Antialiasing," *IEEE Computer Graphics and Applications* **11**(1), pp. 80–88. (VII.9 Motion Blur on Graphics Workstation)

Barr, A. H. (1981). "Superquadrics and Angle-Preserving Transformations," *IEEE Computer Graphics and Applications* **1**(1). (III.8 Rigid Physically Based Superquadrics)

REFERENCES

Barr, A. H. (1984). "Local and Global Deformations of Solid Primitives," *Computer Graphics* **18**(3). (111.8 Rigid Physically Based Superquadrics)

Barzel, R., and Barr, A. H. (1988). "A Modeling System Based on Dynamic Constraints," *Computer Graphics* **22**(4), pp. 179–188. (III.8 Rigid Physically Based Superquadrics)

Barzel, R. (1992). *Physically-Based Modeling for Computer Graphics,* Academic Press, Boston. (III.8 Rigid Physically Based Superquadrics)

Baum, Daniel, R., Rushmeier, Holly E., and Winget, James M. (1989). "Improving Radiosity Solutions through the Use of Analytically Determined Form-Factors," *Computer Graphics (SIGGRAPH '89 Proceedings)* **23**(3), 325–334. (VI.8 Hemispherical Projection of a Triangle; VI.11 Accurate Form-Factor Computation)

Baum, Daniel R., Mann, Stephen, Smith, Kevin P., and Winget, James M. (1991). "Making Radiosity Usable: Automatic Preprocessing and Meshing Techniques for the Generation of Accurate Radiosity Solutions," *Computer Graphics* **25**(4), 51–60. (V.4 Grouping Nearly Coplanar Polygons into Coplanar Sets)

Baumgart, B. G. (1975). "A Polyhedron Representation for Computer Vision," *Proc. NCC*. (V.2 Partitioning a 3–D Convex Polygon with an Arbitrary Plane)

Beran-Koehn, Jeffrey, and Pavicic, Mark (1991). "A Cubic Tetrahedral Adaptation of the Hemi-cube Algorithm," *Graphics Gems II* (James Arvo, ed.). Academic Press, Boston. (VI.10 Delta Form-Factor Calculation for the Cubic Tetrahedral Algorithm)

Bian, Buming (1990). "Accurate Simulation of Scene Luminances," Ph.D. Dissertation, Worcester Polytechnic Institute, Worcester, Massachusetts. (VI.8 Hemispherical Projection of a Triangle)

Birkhoff, G., and MacLane, S. (1965). *A Survey of Modern Algebra* 3rd Ed., Exercise 15, Section IX-3, p. 240; also corollary, Section IX-14, pp. 277–278. MacMillan, New York. (II.5 Fast Generation of Cyclic Sequences)

Blinn, James (1984). "Homogeneous Properties of Second Order Surfaces," course notes, ACM SIGGRAPH '87, Vol. 12, July 1984. (VI.2 Intersecting a Ray with a Quadric Surface)

Blinn, James (1987). "How Many Ways Can You Draw a Circle?" *Computer Graphics and Applications* **7**(8), 74–85. (IV.1 A Parametric Elliptical Arc Algorithm)

Bohlender, G., Böhm, H., Kaucher, E., Kirchner, R., Kulisch, U., Rump, S., Ullrich, Ch., and von Gudenberg, W. (1981). "PASCAL–SC: A Pascal for Contemporary Scientific Computation," IBM Report RC 9009. (II.4 Interval Arithmetic)

Booth, K. S., Forsey, D. R., and Paeth, A. W. (1986). "Fast Z-Buffer Clearing," Graphics Interface '86, Vancouver; also CG&A '87. (II.5 Fast Generation of Cyclic Sequences)

Bowyer, Adrian, and Woodwark, John (1983). A *Programmer's Geometry.* Butterworths, London. (I.9 A Fast Boundary Generator for Composited Regions)

Bresenham, J. E. (1965). "Algorithm for Computer Control of a Digital Plotter," *IBM Systems Journal* **4**(1), 25–30. (VII.6 Accurate Polygon Scan Conversion Using Half-Open Intervals)

REFERENCES

Burnett, David S. (1987). *Finite Element Analysis from Concepts to Applications.* Addison-Wesley, Reading, Massachusetts. (VI.9 Linear Radiosity Approximations Using Vertex-to-Vertex Form Factors)

Campbell, A. T. III, and Fussell, Donald S. (1990). "Adaptive Mesh Generation for Global Diffuse Illumination," *Computer Graphics* **24**(4), 155–164. (V.4 Grouping Nearly Coplanar Polygons into Coplanar Sets)

Carpenter, L. C. (1984). "The A-Buffer, an Anti-Aliased Hidden Surface Method," *Computer Graphics* **18**(3), 103–108. (VII.3 Edge and Bit-Mask Calculations for Anti-Aliasing)

CGI (1989). International Standards Organization, *Information Processing Systems— Computer Graphics—Interfacing Techniques for Dialogues with Graphical Devices* (*Computer Graphics Interface*), ISO/DIS 9636 (December 7, 1989). American National Standards Institute, New York. (IV.1 A Parametric Elliptical Arc Algorithm)

Chen, Michael, Mountford, S. Joy, and Sellen, Abigail (1988). "A Study in Interactive 3-D Rotation Using 2-D Control Devices," *Proc. SIGGRAPH 1988, Computer Graphics* **22**, 121–130. (II.3 The Rolling Ball)

Chin, Norman, and Feiner, Steven (1989). "Near Real-Time Shadow Generation Using BSP Trees," *Computer Graphics* (*SIGGRAPH '89 Proceedings*) **23**(3), 99–106. (V.2 Partitioning a 3-D Convex Polygon with an Arbitrary Plane; V.4 Grouping Nearly Coplanar Polygons into Coplanar Sets)

Cleary, J. G., Wyvill, B., Birtwistle, G. M., and Vatti, R. (1983). "Multiprocessor Ray Tracing," Technical Report No. 83/128/17, October 1983. Department of Computer Science, University of Calgary. (VI.3 Use of Residency Masks and Object Space Partitioning to Eliminate Ray–Object Intersection Calculations)

Cohen, Dan (1969). "Incremental Methods for Computer Graphics," ESD-TR-69–193 (April 1969), p. 42. Department of Engineering and Applied Physics, Harvard University. (IV.1 A Parametric Elliptical Arc Algorithm)

Cohen, Michael F., and Greenberg, Donald P. (1985). "The Hemi-cube: A Radiosity Solution for Complex Environments," *Computer Graphics* (*SIGGRAPH '85 Proceedings*) **19**(3), 31–40. (V.2 Partitioning a 3-D Convex Polygon with an Arbitrary Plane; VI.8 Hemispherical Projection of a Triangle; VI.9 Linear Radiosity Approximations Using Vertex-to-Vertex Form Factors; VI.10 Delta Form-Factor Calculation for the Cubic Tetrahedral Algorithm; VI.11 Accurate Form-Factor Computation)

Cohen, Michael F., Chen, Shengchang Eric, Wallace, John R., and Greenberg, Donald P. (1988). "A Progressive Refinement Approach to Fast Radiosity Image Generation," *Computer Graphics* **22**(4), 75–84. (VI.11 Accurate Form-Factor Computation)

Conte, Samuel Daniel, and de Boor, Carl (1980). *Elementary Numerical Analysis,* pp. 284–286. McGraw-Hill, New York. (V.12 Curve Tesselation Criteria through Sampling)

Cook, R. L. (1986). "Stochastic Sampling in Computer Graphics," *ACM Transactions on Graphics* **5**(1), 51–72. (VII.9 Motion Blur on Graphics Workstation)

REFERENCES

Cook, Robert L., Porter, Thomas, and Carpenter, Loren (1984). "Distributed Ray Tracing," *Computer Graphics (Proc. SIGGRAPH 1984)* **18**(3), 137–146. (I.8 $2\frac{1}{2}$-D Depth-of-Field Simulation for Computer Animation; VII.9 Motion Blur on Graphics Workstation)

Cook, Robert L., Porter, Thomas, and Carpenter, Loren (1984). " Distributed Ray Tracing," *Computer Graphics* **18**(4), 165–174. (II.7 Nonuniform Random Point Sets via Warping; VI.7 Physically Correct Direct Lighting for Distribution Ray Tracing)

Coxeter, H. S. M. (1969). *Introduction to Geometry,* 2nd Ed. Wiley, New York. (V.1 Triangles Revisited)

Cychosz, J. M. (1986). "Vectorized Ray Tracing of Polygonal Models," Unpublished results. (VI.3 Use of Residency Masks and Object Space Partitioning to Eliminate Ray-Object Intersection Calculations)

Cychosz, Joseph M. (1990). "Efficient Generation of Sampling Jitter Using Lookup Tables," Graphics Gems (Andrew Glassner, ed.). Academic Press, Boston. (I.4 A Simple Color Reduction Filter)

Dahmen, Wolfgang A., and Micchelli, Charles A. (1982). "On the Linear Independence or Multivariate B-Splines. I. Triangulations of Simploids," *SIAM Journal of Numerical Analysis* **19**(5), 993–1012. (V.10 Understanding Simploids)

Dana, Alicia C. (1991). *Chronos-3D Key Frame Animator User's Manual.* Lexicor Software, Fairfax, California. (III.5 Issues and Techniques for Keyframing Transformations)

Dalany, H. C. (1988). "Ray Tracing on a Connection Machine," Technical Paper VZ88-3, April 1988. Thinking Machines Corp. (VI.3 Use of Residency Masks and Object Space Partitioning to Eliminate Ray-Object Intersection Calculations)

Diaconis, P., and Shahshahani, M. (1986). "The Subgroup Algorithm for Generating Uniform Random Variables," Technical Report No. 257, September 1986. Department of Statistics, Stanford University. (III.6 Uniform Random Rotations)

Dippé, M. E., and Swensen, J. (1984). "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis," *Computer Graphics (SIGGRAPH '84 Proceedings* **18**(3), 149–158. (VI.3 Use of Residency Masks and Object Space Partitioning to Eliminate Ray-Object Intersection Calculations)

Duff, Tom (1985). "Compositing 3-d Rendered Images," *Proc. SIGGRAPH 1985,* 41–44. (I.8 $2\frac{1}{2}$-D Depth-of-Field Simulation for Computer Animation)

Dürst, M. J. (1988). "Additional Reference to Marching Cubes," *Computer Graphics* **22**(2), 72–73. (I.5 Compact Isocontours from Sampled Data)

Edmonds, A. R. (1957). *Angular Momentum in Quantum Mechanics.* Princeton University Press, Princeton, New Jersey. (II.3 The Rolling Ball)

Farin, Gerald E. (1990). *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide.* Academic Press, Boston. (V.11 Converting Bézier Triangles into Rectangular Patches)

Faux, I. D., and Pratt, M. J. (1979). *Computational Geometry for Design and Manufacture.* Halsted Press, New York. (III.8 Rigid Physically Based Superquadrics)

REFERENCES

Fleischer, Kurt (1991). "Polygon Scan Conversion Derivations," Caltech Technical Report CS-TR-91-12. California Institute of Technology, Pasadena, California. (VII.6 Accurate Polygon Scan Conversion Using Half-Open Intervals)

Foley, J. D., and van Dam, A. (1982). *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, Reading, Massachusetts. (VI.2 Intersecting a Ray with a Quadric Surface)

Foley, James D., van Dam, Andries, Feiner, Steven K., and Hughes, John F. (1990). *Computer Graphics Principles and Practice.* Addison-Wesley, Reading, Massachusetts. (I.2 General Filtered Image Rescaling; I.9 A Fast Boundary Generator for Composited Regions; p. 951–961, IV.1 A Parametric Elliptical Arc Algorithm; V.2 Partitioning a 3-D Convex Polygon with an Arbitrary Plane; VI.6 A Linear-Time Simple Bounding Volume Algorithm)

Forsythe, G. E., and Moler, C. B. (1967). *Computer Solution of Linear Algebraic Systems.* Prentice-Hall, Englewood Cliffs, New Jersey. (II.4 Interval Arithmetic)

Freeman, H. (1961). "On the Encoding of Arbitrary Geometric Configurations," *IRE Transactions on Electronic Computers* **EC**-**10**(2), 260–268. (I.6 Generating Isovalue Contours from a Pixmap)

Fuchs, H., Kedem, A., and Naylor, B. (1980). "On Visible Surface Generation by *A Priori* Tree Structures," *Computer Graphics* (*SIGGRAPH '80 Proceedings*) **14**(3), 124–133. (V.2 Partitioning a 3-D Convex Polygon with an Arbitrary Plane; V.4 Grouping Nearly Coplanar Polygons into Coplanar Sets)

Fujimoto, A., Tanaka, A., and Iwata, K. (1986). "ARTS: Accelerated Ray-Tracing System," *IEEE Computer Graphics & Applications* **6**(4), 10–26. (VI.3 Use of Residency Masks and Object Space Partitioning to Eliminate Ray-Object Intersection Calculations)

Fussell, Donald, and Subramanian, K. R. (1988). "Fast Ray Tracing Using K-D Trees," Technical Report No. TR-88-07. Department of Computer Science, The University of Texas at Austin. (VI.1 Ray Tracing with the BSP Tree)

Gardner, G. (1984). "Simulation of Natural Scenes Using Textured Quadric Surfaces," *Computer Graphics* **18**(3), pp. 11–20. (VI.2 Intersecting a Ray with a Quadric Surface)

Garloff, J. (1985). "Interval Mathematics. A Bibliography," *Freiburger Interval-Berichte* (*6*), 1–122. (II.4 Interval Arithmetic)

Garloff, J. (1987). "Bibliography on Interval Mathematics. Continuation," *Freiburger Interval-Berichte* (*2*), 1–50. (II.4 Interval Arithmetic)

Gasson, Peter C. (1983). *Geometry of Spatial Forms.* Ellis Horwood Ltd., Chichester, West Sussex, United Kingdom. (VI.2 Intersecting a Ray with a Quadric Surface)

Gervautz, Michael, and Purgathofer, Werner (1990). "A Simple Method for Color Quantization: Octree Quantization," *Graphics Gems* (Andrew Glassner, ed.). Academic Press, Boston. (I.4 A Simple Color Reduction Filter)

Glassner, Andrew (1984). "Space Subdivision for Fast Ray Tracing," *IEEE Computer Graphics and Applications* **4**(10), 15–22. (VI.1 Ray Tracing with the BSP Tree; VI.3 Use of Residency Masks and Object Space Partitioning to Eliminate Ray-Object Intersection Calculations)

Glassner, Andrew (1990). Notes for Course 24, "Implementation Notes for Ray Tracers," ACM SIGGRAPH '90. (VI.1 Ray Tracing with the BSP Tree)

Glassner, Andrew S., ed. (1990). *Graphics Gems.* Academic Press, Boston. (II.3 The Rolling Ball; II.6 A Generic Pixel Selection Mechanism; V.1 Triangles Revisited)

Glassner, Andrew (1990). "Useful 3D Geometry," *Graphics* Gem. (Andrew S. (Glassner, ed.). Academic Press, Boston. (V.3 Signed Distance from Point to Plane)

Glassner, Andrew S., ed. (1989). *An Introduction to Ray Tracing.* Academic Press, London. (V.2 Partitioning a 3-D Convex Polygon with an Arbitrary Plane; VI.2 Intersecting a Ray with a Quadric Surface)

Goldman, R. N. (1990). "Matrices and Transformations," *Graphics Gems* (Andrew S. Glassner, ed.), pp. 472–475. Academic Press, Boston. (II.8 Cross Product in Four Dimensions and Beyond; III.2 Decomposing Projective Transformations; III.3 Decomposing Linear and Affine Transformations)

Goldman, R. N. (1990). "Some Properties of Bézier Curves." *Graphics Gems* (Andrew S. Glassner, ed.). Academic Press, Boston. (III.7 Interpolation Using Bézier Curves)

Goldman, Ronald (1990). "Intersection of Two Lines in Three-Space," *Graphics Gems* (Andrew S. Glassner, ed.). Academic Press, Boston. (IV.6 Faster Line Segment Intersection)

Goldman, Ronald (1990). "Triangles," *Graphics Gems* (Andrew *S.* Glassner, ed.), pp. 20–23. Academic Press, Boston. (V.1 Triangles Revisited)

Goldman, R. N. (1991a). "Recovering the Data from the Transformation Matrix," *Graphics Gems II* (James Arvo, ed.), pp. 324–331. Academic Press, Boston. (III.2 Decomposing Projective Transformations; III.3 Decomposing Linear and Affine Transformations)

Goldman, R. N. (1991b). "More Matrices and Transformations: Shear and Pseudo-Perspective," *Graphics Gems II* (James Arvo, ed.), pp 338–341. Academic Press, Boston. (II.8 Cross Product in Four Dimensions and Beyond. III.3 Decomposing Linear and Affine Transformations)

Goldstein, Herbert (1980). *Classical Mechanics,* 2nd Ed. Addison-Wesley, Reading, Massachusetts. (111.8 Rigid Physically Based Superquadrics)

Golub, Gene H., and Van Loan, Charles F. (1985). *Matrix Computations.* The Johns Hopkins University Press, Baltimore, Maryland. (III.4 Fast Random Rotation Matrices)

Gonzalez, Rafael C., and Wintz, Paul (1987). *Digital Image Processing.* Addison-Wesley, Reading, Massachusetts. (I.9 A Fast Boundary Generator for Composited Regions)

Goral, Cindy M., Torrance, Kenneth E., Greenberg, Donald P., and Battaile, Bennett (1984). "Modeling the Interaction of Light between Diffuse Surfaces," *Computer Graphics* **18**(3), 213–222. (VI.11 Accurate Form-Factor Computation)

Haines, Eric A. (1991). "Ronchamp: A Case Study for Radiosity," course notes, "Frontiers in Rendering," SIGGRAPH '91. (V.4 Grouping Nearly Coplanar Polygons into Coplanar Sets)

Haines, Eric A., and Wallace, John R. (1991). "Shaft Culling for Efficient Ray-Traced Radiosity," *SIGGRAPH '91 Radiosity Course Notes.* (VI.11 Accurate Form-Factor Computation)

Hall, Mark (1990). "Defining Surfaces from Sampled Data," *Graphics Gems* (Andrew Glassner, ed.), pp. 552–557. Academic Press, Boston. (I.5 Compact Isocontours from Sampled Data)

Hanrahan, P. (1989). "A Survey of Ray-Surface Intersection Algorithms," *An Introduction to Ray Tracing* (A. Glassner, ed.). Academic Press, New York. (II.4 Interval Arithmetic)

Hanrahan, Pat, Salzman, David, and Aupperle, Larry (1991). "A Rapid Hierarchical Radiosity Algorithm," *Computer Graphics* **25**(4), 197–206. (VI.11 Accurate Form-Factor Computation)

Heckbert, P. (1990). "Generic Convex Polygon Scan Conversion and Clipping," *Graphics Gems* (Andrew S. Glassner, ed.), pp. 84–86. Academic Press, Boston. (V.2 Partitioning a 3-D Convex Polygon with an Arbitrary Plane; VII.2 Fast Linear Color Rendering)

Higgins, T. M., and Booth, K. S. (1986). "A Cel-Based Model for Paint Systems," *Proceedings, Graphics Interface '86, (Vancouver)*, pp. 82–90. (II.6 A Generic Pixel Selection Mechanism)

Hoffman, C. M. (1989). *Geometric & Solid Modeling: An Introduction.* Morgan Kaufmann, San Mateo, California. (IV.4 Exact Computation of 2-D Intersections)

Hottel, Hoyt C., and Sarofin, Adel F. (1967). *Radiative Transfer.* McGraw-Hill, New York. (VI.11 Accurate Form-Factor Computation)

Hourcade, J., and Nicolas, A. (1985). "Algorithms for Anti-Aliased Cast Shadows," *Computer and Graphics* **9**(3), 259–265. (VII.1 The Shadow Depth Map Revisited)

Immel, David S., Cohen, Michael F., and Greenberg, Donald P. (1986). "A Radiosity Method for Non-diffuse Environments," *Computer Graphics* **20**(4), 133–142. (VI.7 Physically Correct Direct Lighting for Distribution Ray Tracing) Institute for New Technologies (1991). *Interval Computations,* Vol. 1, Leningrad. (II.4 Interval Arithmetic)

Jansen, Frederik (1986). *Data Structures for Raster Graphics,* "Data Structures for Ray Tracing," pp. 57–73. Springer-Verlag, The Netherlands. (VI.1 Ray Tracing with the BSP Tree)

Kajiya, James T. (1986). "The Rendering Equation," *Computer Graphics* (*ACM SIGGRAPH '86 Conference Proceedings*) **20**(4), 143–150. (VI.7 Physically Correct Direct Lighting for Distribution Ray Tracing)

Kalra, D., and Barr, A. H. (1989). "Guaranteed Ray Intersections with Implicit Surfaces," *Computer Graphics* **23**, 297–306. (II.4 Interval Arithmetic)

Kaplan, M. R. (1985). "The Uses of Spatial Coherence in Ray Tracing," course notes, "State of the Art in Image Synthesis," *SIGGRAPH '85 Course Notes No. 11, July 1985,* pp. 22–26. (VI.3 Use of Residency Masks and Object Space Partitioning to Eliminate Ray-Object Intersection Calculations)

Kaplan, Michael (1986). Course notes, "Space-Tracing, A Constant Time Ray-Tracer," ACM SIGGRAPH '85. (VI.1 Ray Tracing with the BSP Tree)

Kay, Timothy, and Kajiya, James (1986). "Ray Tracing Complex Scenes," *Computer Graphics* (*SIGGRAPH '86 Proceedings*) **20**(4), 269–278. (VI.1 Ray Tracing with the BSP Tree; VI.3 Use of Residency Masks and Object Space Partitioning to Eliminate Ray-Object Intersection Calculations)

Knuth, D. E. (1981). *The Art of Computer Programming,* Vol.2, "Seminumerical Algorithms." Addison-Wesley, Reading, Massachusetts. (II.5 Fast Generation of Cyclic Sequences; III.6 Uniform Random Rotations)

Koparkar, P. A., and Mudur, S. P. (1985). "Subdivision Techniques for Processing Geometric Objects," *Fundamental Algorithms for Computer Graphics* (Bresenham et al., eds.), NATO ASI Series. Springer-Verlag, New York. (II.4 Interval Arithmetic)

Kuhn, H. W. (1960). "Some Combinatorial Lemmas in Topology," *IBM Journal of Research and Development* **45***,* 518–524. (V.10 Understanding Simploids)

Lathrop, Olin, Kirk, David, and Voorhies, Doug (1990). "Accurate Rendering by Subpixel Addressing," *IEEE Computer Graphics and Applications* **10**(5), 45–53. (VII.6 Accurate Polygon Scan Conversion Using Half-Open Intervals)

Lee, Mark E., Redner, Richard A., and Uselton, Samuel p. (1985). *Computer Graphics* (*ACM SIGGRAPH '85 Conference Proceedings*) **19**(3), 61–68. (Vl.7 Physically Correct Direct Lighting for Distribution Ray Tracing)

Lorenson, W., and Cline, H. (1987). "Marching Cubes: A High Resolution 3d Surface Construction Algorithm," *Compuler Graphics* **21**(4)*,* 163–169. (I.5 Compact Isocontours from Sampled Data)

MacDonald, David, and Booth, Kellogg (1989). "Heuristics for Ray Tracing Using Space Subdivision," *Proceedings of Graphics Interface '89,* 152–163. (VI. 1 Ray Tracing with the BSP Tree)

Manly, B. F. J. (1986). *Multivariate Statistical Methods.* Chapman and Hall, London. (VI.6 A Linear-Time Simple Bounding Volume Algorithm)

Max, Nelson L., and Lerner, Douglas M. (1985). "A Two-and-a-Half-D Motion-Blur Algorithm," *Proc. SIGGRAPH 1985,* 85–93. (I.8 $2\frac{1}{2}$-D Depth-of-Field Simulation for Computer Animation)

Megiddo, N. (1984). "Linear Programming in Linear Time When the Dimension Is Fixed," *J. of ACM* **31**, 114–127. (VI.6 A Linear-Time Simple Bounding Volume Algorithm)

Mitchell, Don P. (1987). "Generating Antialiased Images at Low Sampling Densities," *Computer Graphics* **21**(4), 65–72. (VI.4 A Panoramic Virtual Screen for Ray Tracing)

Mitchell, Don P., and Netravali, Arun N. (1988). "Reconstruction Filters in Computer Graphics," *Computer Graphics* **22**(4), 221–228. (I.2 General Filtered Image Rescaling)

Moore, Doug, and Warren, Joe (1991). "Mesh Displacement: An Improved Contouring Method for Trivariate Data," Technical Report TR 91–166. Rice University, Department of Computer Science. (I.5 Compact Isocontours from Sampled Data)

Moore, R. E. (1966). *Interval Analysis.* Prentice-Hall, Englewood Cliffs, New Jersey. (II.4 Interval Arithmetic)

Morton, Mike (1990). "A Digital 'Dissolve' Effect," *Graphics Gems* (Andrew D. Glassner, ed.). Academic Press, Boston. (II.5 Fast Generation of Cyclic Sequences)

Mudur, S. P., and Koparkar, P. A. (1984). "Interval Methods for Processing Geometric Objects," *IEEE Computer Graphics and Applications* **4**(2)*,* 7–17. (II.4 Interval Arithmetic)

Musgrave, F. Kenton (1990). "About the Cover: Natura ex Machina II," *IEEE Computer Graphics and Applications* **10**(6), 5–7. (VI.4 A Panoramic Virtual Screen for Ray Tracing)

Musgrave, F. Kenton (1991). "Use of Fractional Brownian Motion in Modelling Nature," Course 14, "Fractal Modelling in 3D Computer Graphics and Imaging," SIGGRAPH '91, July 1991. (VI.4 A Panoramic Virtual Screen for Ray Tracing)

Musgrave, F. Kenton, and Mandelbrot, B. B. (1989). "Natura ex Machina," *IEEE Computer Graphics and Applications* **9**(1), 4–7. (VI.4 A Panoramic Virtual Screen for Ray Tracing)

Naylor, Bruce, Amanatides, John, and Thibault, William (1990). "Merging BSP Trees Yields Polyhedral Set Operations," *Computer Graphics* **24**(4), 115–124. (V.4 Grouping Nearly Coplanar Polygons into Coplanar Sets)

Neumaier, A. (1990). *Interval Methods for Systems of Equations.* Cambridge University Press, Cambridge, United Kingdom. (II.4 Interval Arithmetic)

Newman, W., and Sproull, R. (1979). *Introduction to Computer Graphics,* 2nd Ed., Vol. 1, pp. 256–257, exercises 17–8 through 17–10, McGraw-Hill, New York. (II.5 Fast Generation of Cyclic Sequences; II.6 A Generic Pixel Selection Mechanism)

Newman, William, and Sproull, Robert (1979), *Principles of Interactive Computer Graphics,* pp. 27–28. McGraw-Hill, New York. (IV.1 A Parametric Elliptical Arc Algorithm)

Nishimura, H., Ohno, H., Kawata, T., Shirakawa, I., and Omura, K. (1983). "LINKS-1: A Parallel Pipelined Multimicrocomputer System for Image Creation," *Conference Proceedings of the 10th Annual International Symposium on Computer Architecture, SIGGARCH,* 387–394. (VI.3 Use of Residency Masks and Object Space Partitioning to Eliminate Ray-Object Intersection Calculations)

Oppenheim, A. V., and Schaeffer, R. W. (1975). *Digital Signal Processing.* Prentice-Hall, Englewood Cliffs, New Jersey. (I.2 General Filtered Image Rescaling)

Paeth, A. W. (1986). "Fast Raster rotation." Graphics Interface '86, Vancouver. (II.5 Fast Generation of Cyclic Sequences)

Paeth, A. W. (1990a). "Fast Median Finding on a 3×3 Grid," *Graphics Gems* (Andrew S. Glassner, ed.). Academic Press, Boston. (II.5 Fast Generation of Cyclic Sequences)

Paeth, A. W. (1990b). "A Fast Algorithm for General Raster Rotation," *Graphics Gems* (Andrew S. Glassner, ed.), pp. 179–195. Academic Press, Boston. (II.5 Fast Generation of Cyclic Sequences; IV.1 A Parametric Elliptical Arc Algorithm)

Paeth, A. W. (1990). "Digital Cartography for Computer Graphics," *Graphics Gems* (Andrew S. Glassner, ed.), pp. 307–320. Academic Press, Boston. (VI.4 A Panoramic Virtual Screen for Ray Tracing)

Paeth, A. (1991). "Mapping RGB Triples onto 16 Distict Values," *Graphics Gems II* (James Arvo, ed.). Academic Press, Boston. (II.6 A Generic Pixel Selection Mechanism)

Paeth, Alan W., and Schilling, David (1991). "Of Integers, Fields, and Bit Counting," *Graphics Gems II* (James Arvo, ed.), p. 371. Academic Press, Boston. (II.5 Fast Generation of Cyclic Sequences)

Pavlidis, Theo (1982). *Algorithms for Graphics and Image Processing,* Chapter 14. Computer Science Press. (VII.6 Accurate Polygon Scan conversion Using Half-Open Intervals)

Pentland, Alex, and Williams, J. (1989). "Good Vibrations: Modal Dynamics for Graphics and Animation," *Computer Graphics* **23**(3), pp. 223–232. (III.8 Rigid Physically Based Superquadrics)

Perlin, Ken (1985). Course notes, "Seminar on State-of-the-Art in Image Synthesis," *SIGGRAPH 1985, July 1985.* (I.8 $2\frac{1}{2}$-D Depth-of-Field .Simulation for Computer Animation)

Plunkett, D. J. (1984). "A Vectorized Ray Tracing Algorithm," MSME Thesis, Purdue University, West Lafayette, Indiana.

Plunkett, D. .J., and Bailey, M. .J. (1985). "The Vectorization of a Ray-Tracing Algorithm for Improved Execution Speed," *IEEE Computer Graphics and Applications* **5**(8), 52–60. (VI.3 Use of Residency Masks and Object Space Partitioning to Eliminate Ray-Object Intersection Calculations)

Porter, Thomas, and Duff, Tom (1984). "Compositing Digital Images," *Proc. SIGGRAPH 1984*, 258–259. (I.8 $2\frac{1}{2}$-D Depth-of-Field Simulation for Computer Animation)

Porter, Thomas, and Duff, Tom (1984). "Compositing Digital Images," *Computer Graphics* **18**(3), 253–260. (1.7 Compositing Black-and-White Bitmaps)

Potmesil, Michael, and Chakravarty, Indranil (1982). "Synthetic Image Generation with a Lens and Aperture Camera Model," *ACM Trans. on Graphics* **1**(2), 85–108. (1.8 $2\frac{1}{2}$-D Depth-of-Field Simulation for Computer Animation)

Prasad, Mukesh (1991). "Intersection of Line Segments," *Graphics Gems II* (James Arvo, ed.), pp. 7–9. Academic Press, Boston, (IV.4 Exact Computation of 2-D Intersections; IV.6 Faster Line Segment Intersection)

Pratt, William K. (1991). *Digital Image Processing.* Wiley-Interscience, New York. (I.2 General Filtered Image Rescaling)

Preparata, F. P., and Shamos, M. I. (1985). *Computational Geometry.* Springer-Verlag, New York. (VI.6 A Linear-Time Simple Bounding Volume Algorithm)

Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T. (:1988). *Numerical Recipes in C.* Cambridge University Press, Cambridge, United Kingdom. (VI.6 A Linear-Time Simple Bounding Volume Algorithm)

Rabiner, Lawrence R., and Gold, Bernard (1975). *Theory and Application of Digital Signal Processing.* Prentice-Hall, Englewood Cliffs, New Jersey. (I.2 General Filtered Image Rescaling)

Reduce (]987). *Reduce User's Manual, Ver. 3.3*, July 1987. The RAND Corporation, Santa Monica, California. (III.7 Interpolation Using Bézier Curves)

Reeves, W., Salesin, D., and Cook, R. (1987). "Rendering Anti-aliased Shadows with Depth Maps," *Computer Graphics* **21**(4), 283–291. (VII.1 The Shadow Depth Map Revisited)

Ritter, J. (1990). "An Efficient Bounding Sphere," *Graphics Gems* (Andrew S. Glassner, ed.), pp. 801–303. Academic Press, Boston. (VI.6 A Linear-Time Simple Bounding Volume Algorithm

Ritter, Jack (1991). "Fast Sign of Cross Product Calculation," *Graphics Gems II* (James Arvo, ed.). Academic Press, Boston.(IV.6 Faster Line Segment Intersection)

Rokne, Jon (1991). *Graphics Gems (Andrew* Glassner, ed.), Academic Press, Boston. (IV.7 Solving the Problems of Appolonius and Other Related Problems)

Roth, S. (1982). "Ray Casting for Modeling Solids," *Computer Graphics and Image Processing* **18**, 109–144 (VI.2 Intersecting a Ray with a Quadric Surface)

Rubin, S. M., and Whitted, T. (1980). "A 3-Dimensional Representation for Fast Rendering of Complex Scenes," *Computer Graphics* (*SIGGRAPH '80 Proceedings*) **14**(3), 110–116. (VI.3 Use of Residency Masks and Object Space Partitioning to Eliminate Ray-Object Intersection Calculations)

Salesin, David (1991). "Epsilon Geometry: Building Robust Algorithms from Imprecise Computations." Ph.D. Thesis, Stanford University. (V.4 Grouping Nearly Coplanar Polygons into Coplanar Sets)

Salesin, David, and Barzel, Ronen (1986). "Two-Bit Graphics," *IEEE Computer Graphics and Applications* **6**(6), 36–42. (I.7 Compositing Black-and-White Bitmaps)

Screider, Y. A. (1966). *The Monte Carlo Method.* Pergamon Press, New York. (II.7 Nonuniform Random Point Sets via Warping; VI.7 Physically Correct Direct Lighting for Distribution Ray Tracing)

Segal, Mark (1990). "Using Tolerances to Guarantee Valid Polyhedral Modeling Results," *Computer Graphics* **24**(4), 105–114. (V.4 Grouping Nearly Coplanar Polygons into Copolanar Sets)

Shoemake, K. (1985). "Animating Rotation with Quaternion Curves," *Proc. SIGGRAPH 1985, Computer Graphics* **19**(3), 245–254. (II.3 The Rolling Ball; III.1 Quaternion Interpolation with Extra Spins; III.6 Uniform Random Rotations)

Shoemake, K. (1989). "Quaternion Calculus for Animation," SIGGRAPH Course #23, Math for SIGGRAPH, presented at SIGGRAPH '89, Boston. Also presented in Course #2 at SIGGRAPH '91, Las Vegas. (III.6 Uniform Random Rotations)

Shoemake, K. (1991). "Quaternions and 4 ¥ 4 Matrices," *Graphics Gems II* (James Arvo, ed.), pp. 355–356. Academic Press, Boston. (III.6 Uniform Random Rotations)

Smith, Alvy Ray (1981). "Digital Filtering Tutorial for Computer Graphics," *Technical Memo No. 27.* Lucasfilm Ltd. (I.2 General Filtered Image Rescaling)

Smith, Alvy Ray (1982). "Digital Filtering Tutorial for Computer Graphics, Part II," *Technical Memo No. 44,* Lucasfilm Ltd. (I.2 General Filtered Image Rescaling).

Subramanian, K. R., and Fussell, Donald (1991). "Automatic Termination Criteria for Ray Tracing Hierarchies," *Proceedings of Graphics Interface '91,* 93–100. (VI. 1 Ray Tracing with the BSP Tree)

Suffren, K. G., and Fackerell, E. D. (1991). "Interval Methods in Computer Graphics," *Computers and Graphics* **15**, 331–340. (II.4 Interval Arithmetic)

Sung, Kelvin (1991). "A DDA Octree Traversal Algorithrn for Ray Tracing," *Proceedings of Eurographics '91*, 73–85. (VI. 1 Ray Tracing with the BSP Tree)

Sutherland, I. E., and Hodgman, G. W. (1974) "Re-entrant Polygon Clipping," *CACM* **17**(1), 32–42. (V.2 Partitioning a 3-D Convex Polygon with an Arbitrary Plane)

Sutherland, I. E., Sproull, R. F., and Schumaker, R. A. (1974). "A Characterization of Ten Hidden-Surface Algorithms," *ACM Computing Surveys* **6**(1), 1–55. (V.4 Grouping Nearly Coplanar Polygons into Coplanar Sets; V.5 Newell's Methods for Computing the Plane Equation of a Polygon)

Terzopoulos, D., Platt, J., Barr, A. H., and Fleischer, K. (1987). "Elastically Deformable Models," *Computer Graphics* **21**(4), 205–214. (III.8 Rigid Physically Based Superquadrics)

Thibault, W., and Naylor, B. (1987). "Set Operations on Polyhedra Using Binary Space Partitioning Trees," *Computer Graphics* (*SIGRAPH '87 Proceedings*) **21**(4), 153–162. (V.2 Partitioning a 3-D Convex Polygon with an Arbitrary Plane)

Thomas, Frank, and Johnston Ollie (1981). Disney Animation. Abbeville Press, NewYork. (I.8 $2\frac{1}{2}$-D Depth-of-Field Simulation for Computer Animation)

Thomas, Spencer W. (1991). "Decomposing a Matrix into Simple Transformations," *Graphics Gems II (*James Arvo, ed.), pp. 320–323. Academic Press, Boston. (III.2 Decomposing Projective Transformations; III.:3 Decomposing Linear and Affine Transformations; III.5 Issues and Techniques for Keyframing Transformations)

Tran-Thong (1983). "Ellipse, Arc of Ellipse, and Elliptic Spline," *Computers & Graphics* **7**(2), 169–175. (IV. 1 A Parametric Elliptical Arc Algorithm)

Turkowski, Ken (1990). "Filters for Common Resampling Tasks," *Graphics Gems* (Andrew S. Glassner, ed.). Academic Press, Boston. (I.2 General Filtered Image Rescaling)

Van Aken, .Jerry, and Simar Ray (1988). "A Conic Spline Algorithm," *TMS34010 Application Guide* (January 1988). Tl lit. no. SPVA007A, pp. 255–278. Texas Instruments Inc., Dallas. (IV.1 A Parametric Elliptical Arc Algorithm)

van Dam, Andries (1988). "PHIGS + Functional Description Revision 3.0," *Computer Graphics* **22**(3), 125–218. (VII.10 The Shader Cache: A Rendering Pipeline Accelerator)

Wallace, .John R., Elmquist, Kells A., and Haines, Eric A. (1989). "A Ray Tracing Algorithm for Progressive Radiosity," *Computer Graphics* (*SIGGRAPH '89 Proceedings*) **23**(3), 335–314. (VI.9 Linear Radiosity Approximations Using Vertex-to-Vertex Form Factors; VI.11 Accurate Form-Factor Computation)

Watt, A. (1989). *Fundamentals of Three-Dimensional Computer Graphics.* Addison-Wesley, Wokinham, United Kingdom. (VII.3 Edge and Bit-Mask Calculations for Anti-Aliasing)

Weghorst, H., Hopper, G., and Greenberg, D. P. (1934). "Improved Computational Models for Ray Tracing," *ACM Transactions on Graphics* **3**(1)*,* 52–69. (VI.3 Use of Residency Masks and Object Space Partitioning to Eliminate Ray-Object Intersection Calculations)

REFERENCES

Whittaker, E. T. (1944). *A Treatise on the Analytical Dynamics of Particles and Rigid Bodies.* Dover, New York. (II.3 The Rolling Ball)

Williams, L. (1978). "Casting Curved Shadows on Curved Surfaces," *Computer Graphics* **21**(4), 283–291. (VII.1 The Shadow Depth Map Revisited)

Wirth, Niklaus (1973). *Systematic Programming*: *An Introduction.* Prentice-Hall, Englewood Cliffs, New Jersey. (II.5 Fast Generation of Cyclic Sequences)

Wolfram, Stephen (1991). *Mathematica*: *A System for Doing Mathematics by Computer,* 2nd Ed. Addison-Wesley, Reading, Massachusetts. (III.8 Rigid Physically Based Superquadrics)

Woo, A., Poulin, P., and Fournier, A. (1990). "A Survey of Shadow Algorithms," *IEEE Computer Graphics and Applications* **10**(6), 13–22. (VII.1 The Shadow Depth Map Revisited)

Wyvil, G., McPheeters, C., and Wyvil, B. (1986). "Data Structure for *Soft* Objects," *The Visual Computer* **2**, 227–234. (I.5 Compact Isocontours from Sampled Data)

Wyvill, Brian (1990). "Storage-Free Register Swapping," *Graphics Gems* (Andrew S. Glassner, ed.). Academic Press, Boston. (II.5 Fast Generation of Cyclic Sequences)

Yale (1975). *The Contest Book* (*1972 Examination*)*,* question 30, p. 45. Mathematical Association of America. (II.5 Fast Generation of Cyclic Sequences)

Yap, Sue-Ken (1991). "A Fast 90-Degree Bitmap Rotator," (I.3 Optimization of Bitmap Scaling Operations)

INDEX

D

E

F

Q

R

U

V

W