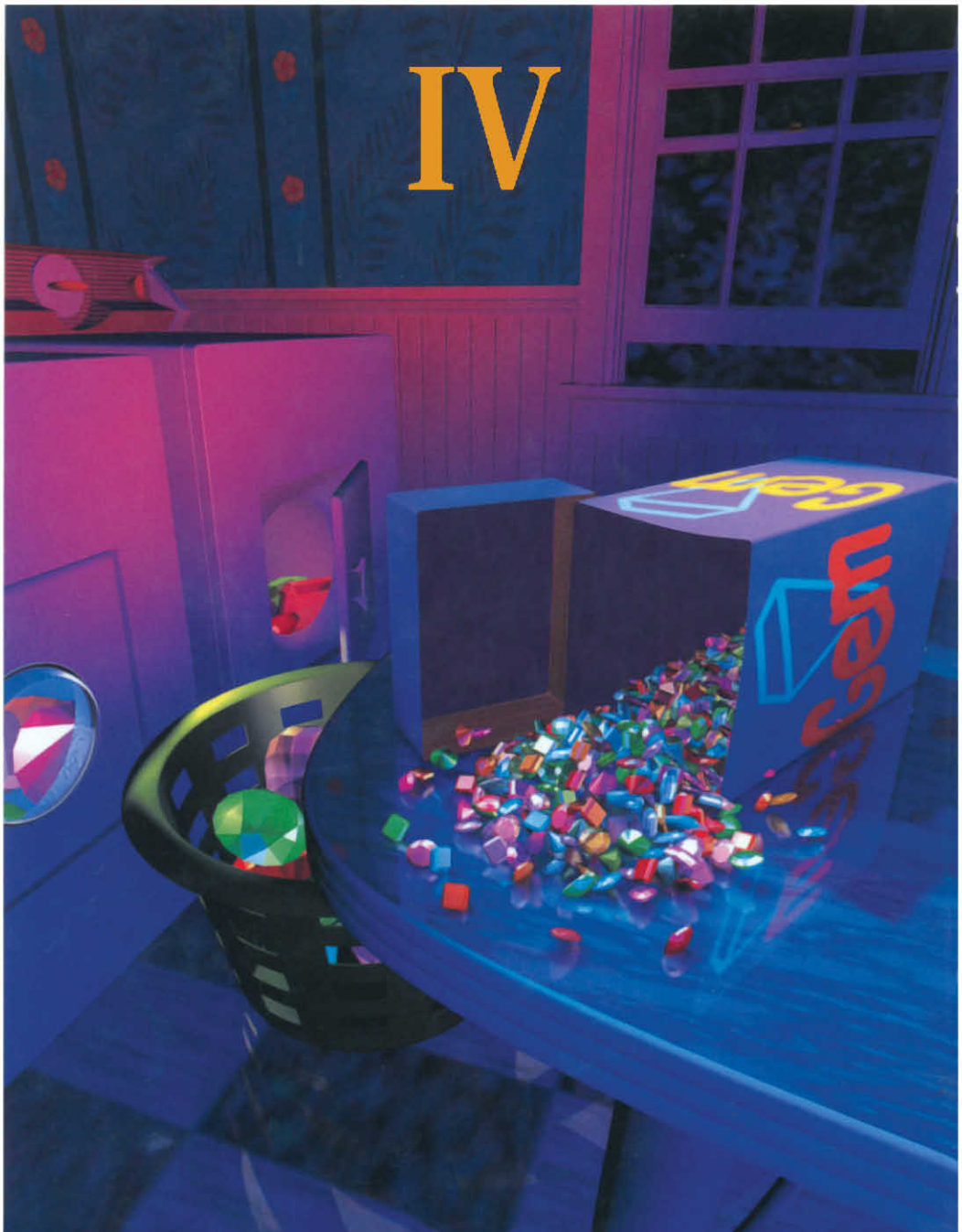
GRAPHICS GEMS

EDITED BY PAUL S. HECKBERT



AP PROFESSIONAL

MAC DISK ENCLOSED

GRAPHICS GEMS IV

This is a volume in

The Graphics Gems Series

A Collection of Practical Techniques for the Computer Graphics Programmer

Series Editor

Andrew Glassner
Xerox Palo Alto Research Center
Palo Alto, California

GRAPHICS GEMS IV

Edited by Paul S. Heckbert

Computer Science Department
Carnegie Mellon University
Pittsburgh, Pennsylvania

AP PROFESSIONAL

Boston San Diego New York
London Sydney Tokyo Toronto

This book is printed on acid-free paper ♾

AP PROFESSIONAL
955 Massachusetts Avenue, Cambridge, MA 02139

An imprint of ACADEMIC PRESS, INC.
A Division of HARCOURT BRACE & COMPANY

United Kingdom Edition published by
ACADEMIC PRESS LIMITED
24–28 Oval Road, London NW1 7DX

Library of Congress Cataloging-in-Publication Data
Graphics gems IV / edited by Paul S. Heckbert.
p. cm. -- (The Graphics gems series)
Includes bibliographical references and index.
ISBN 0-12-336156-7 (with Macintosh disk). —ISBN 0-12-336155-9
(with IBM disk)
1. Computer graphics. I. Heckbert, Paul S., 1958– .
II. Title: Graphics gems 4. III. Title: Graphics gems four.
IV. Series.
T385.G6974 1994
006.6'6--dc20 93-46995
CIP

Printed in the United States of America

94 95 96 97 MV 9 8 7 6 5 4 3 2 1

Contents

This page intentionally left blank

Author Index

Format: *author, institution, chapter number:* p. *start page.*
Author's full address is listed on the first page of each chapter.

Steve Hill, University of Kent, Canterbury, UK, X.1: p. 521.
R. Victor Klassen, Xerox Webster Research Center, Webster, NY, USA, IV.4: p. 261.
Manfred Kopp, Technical University of Vienna, Vienna, Austria, VII.1: p. 413.
Dani Lischinski, Cornell University, Ithaca, NY, USA, I.5: p. 47, IV.5: p. 278.
Joe Marks, Digital Equipment Corporation, Cambridge, MA, USA, IX.1: p. 497.
Henry Massalin, Microunity Corporation, Sunnyvale, CA, USA, VIII.3: p. 447.
Robert D. Miller, E. Lansing, MI, USA II.4: p. 132.
Yoshikazu Ohashi, Cognex, Needham, MA, USA, II.2: p. 120.
Alan W. Paeth, Okanagan University College, Kelowna, British Columbia, Canada, VIII.6: p. 486.
John W. Peterson, Taligent, Inc., Cupertino, CA, USA, IV.6: p. 286.
Rich Rabbitz, Martin Marietta, Moorestown, NJ, USA, I.8: p. 83.
John Schlag, Industrial Light and Magic, San Rafael, CA, USA, VIII.1: p. 433.
Christophe Schlick, Laboratoire Bordelais de Recherche en Informatique, Talence, France, VI.1: p. 385, VI.3: p. 401, VII.3: p. 422.
Peter Schorn, ETH, Zürich, Switzerland, I.2: p. 7.
Ching-Kuang Shene, Northern Michigan University, Marquette, MI, USA, IV.7: p. 321, V.1: p. 353.
Stuart Shieber, Harvard University, Cambridge, MA, USA, IX.1: p. 497.
Peter Shirley, Indiana University, Bloomington, IN, USA, V.4: p. 370.
Ken Shoemake, University of Pennsylvania, Philadelphia, PA, USA, III.1: p. 175, III.4: p. 207, III.5: p. 222, III.6: p. 230.
László Szirmay-Kalos, Technical University of Budapest, Budapest, Hungary, IX.2: p. 505.
Tim Van Hook, Silicon Graphics, Mountain View, CA, USA, II.3: p. 125.
Warren N. Waggenspack, Jr., Louisiana State University, Baton Rouge, LA, USA, V.2: p. 356.
Changyaw Wang, Indiana University, Bloomington, IN, USA, V.4: p. 370.
Greg Ward, Lawrence Berkeley Laboratory, Berkeley, CA, USA, VII.2: p. 415.
Kevin Weiler, Autodesk Inc., Sausalito, CA, USA, I.3: p. 16.
George Wolberg, City College of New York/CUNY, New York, NY, USA, VIII.3: p. 447.
Andrew Woo, Alias Research, Inc., Toronto, Ontario, Canada, VI.2: p. 388.
Kevin Wu, SunSoft, Mountain View, CA, USA, III.3: p. 199.
Guoliang Xu, Purdue University, West Lafayette, IN, USA, IV.3: p. 256.
Karel Zuiderveld, Utrecht University, Utrecht, The Netherlands, VIII.5: p. 474.

Foreword

Andrew S. Glassner

We make images to communicate. The ultimate measure of the quality of our images is how well they communicate information and ideas from the creator's mind to the perceiver's mind. The efficiency of this communication, and the quality of our image, depends on both what we want to say and to whom we intend to say it.

I believe that computer-generated images are used today in two distinct ways, characterized by whether the intended receiver of the work is a person or machine. Images in these two categories have quite different reasons for creation, and need to satisfy different criteria in order to be successful.

Consider first an image made for a machine. For example, an architect planning a garden next to a house may wish to know how much light the garden will typically receive per day during the summer months. To determine this illumination, the architect might build a 3D model of the house and garden, and then use computer graphics to simulate the illumination on the ground at different times of day in a variety of seasons. The images generated by the rendering program would be a by-product, and perhaps never even looked at; they were only generated in order to compute illumination. The only criterion for judgment for such images is an appropriate measure of *accuracy.*

Nobody will pass judgment on the *aesthetics* of these pictures, since no person with an aesthetic sense will ever see them. Accuracy does not require beauty. For example, a simulation may not produce images that are individually correct, but instead average to the correct answer. The light emitted by the sun may be modeled as small, discrete chunks, causing irregular blobs of illumination on the garden. When these blobs are averaged together over many hours and days, the estimates approach the correct value for the received sunlight. No one of these pictures is accurate individually, and probably none of them would be very attractive.

When we make images for people, we have a different set of demands. We almost always require that our images be *attractive* in some way. In this context, attractive does not necessarily mean beautiful, but it means that there must be an aesthetic component influenced by composition, color, weight, and so on. Even when we intend to act as analytic and dispassionate observers, humans have an innate sense of beauty that cannot be denied. This is the source of all ornament in art, music, and literature: we always desire something beyond the purely functional. Even the most utilitarian objects, such as hammers and pencils, are *designed* to provide grace and beauty to our eyes and offer comfort to our hands. When we weave together beauty and utility, we create elegance. People are more interested in beautiful things than neutral things, because they stimulate our senses and our feelings.

So even the most utilitarian image intended to communicate something to another person must be designed with that person in mind: the picture must be composed so that it is balanced in terms of form and space, the colors must harmonize, the shapes must not jar. It is by occasionally violating these principles that we can make one part of the image stand out with respect to the background; ignoring them produces images that have no focus and no balance, and thus do not capture and hold our interest. Their ability to communicate is reduced. Every successful creator of business charts, wallpaper designs, and scientific visualizations knows these rules and works with them.

So images intended for people must be attractive. Only then can we further address the idea of accuracy. What does it mean for an image intended for a person to be "accurate"?

Sometimes "accuracy" is interpreted to mean that the energy of the visible light calculated to form the image exactly matches the energy that would be measured if the modeled scene (including light sources) really existed, and were photographed; this idea is described in computer graphics by the term *photorealism.* This would certainly be desirable, under some circumstances, if the image were intended for a machine's analysis, but the human perceptual apparatus responds differently than a flatbed scanner. People are not very good at determining absolute levels of light, and we are easily fooled into thinking that the brightest and least chromatic part of an image is "white."

Again we return to the question of what we're trying to communicate. If the point of an image is that a garden is well-lit and that there is uniform illumination over its entire surface, then we do not care about the radiometric accuracy of the image as much as the fact that it conveys that information; the whole picture could be too bright or too dark by some constant factor and this message will still be carried without distortion. In the garden image, we expect a certain variation due to the variety of soil, rocks, plants, and other geometry in the scene. Very few people could spot the error in a good but imprecise approximation of such seemingly random fluctuation. In this type of situation, if you can't see the error, you don't care about it. So not only can the illumination be off by a constant factor, it can vary from the "true" value quite a bit from point to point and we won't notice, or if we do notice, we won't mind.

If we want to convey the sense of a scene viewed at night, then we need to take into account the entire observer of a night scene. The human visual system *adapts* to different light levels, which changes how it perceives different ranges of light. If we look at a room lit by a single 25-watt light bulb, and then look at it again when we use a 1000-watt bulb, the overall illumination has changed by a constant factor, but our perception of the room changes in a non-linear way. The room lit by the 25-watt bulb appears dark and shadowy, while the room lit by the 1000-watt bulb is stark and bright. If we display both on a CRT using the same intensity range, even though the underlying radiance values were computed with precision, both images will appear the same. Is this either *accurate* or *photorealistic*?

Sometimes some parts of an image intended for a person must be accurate, depending

on what that image is intended to communicate. If the picture shows a new object intended for possible manufacture, the precise shape may be important, or the way it reflects light may be critical. In these applications we are treating the person as a machine; we are inviting the person to analyze one or more characteristics of the image as a predictor of a real object or scene. When we are making an image of a smooth and glossy object prior to manufacture in order to evaluate its appearance, the shading must match that of the final object as accurately as possible. If we are only rendering the shape in order to make sure it will fit into some packing material, the shading only needs to give us information about the shape of the object; this shading may be arbitrarily inaccurate as long as we still get the right perception of shape. A silver candlestick might be rendered as though it were made of concrete, for example, if including the highlights and caustics would interfere with judging its shape. In this case our definition of "accuracy" involves our ability to judge the structure of shapes from their images, and does not include the optical properties of the shape.

My point is that images made for machines should be judged by very different criteria than images made for people. This can help us evaluate the applicability of different types of images with different objective accuracies. Consider the picture generated for an architect's client, with the purpose of getting an early opinion from the client regarding whether there are enough trees in the yard. The accuracy of this image doesn't matter as long as it looks good and is roughly correct in terms of geometry and shading. Too much precision in every part of the image may lead to too much distraction; because of its perceived realism and implied finality, the client may start thinking about whether a small shed in the image is placed just right, when it hasn't even been decided that there will be a shed at all. Precision implies a statement; vagueness implies a suggestion.

Consider the situation where someone is evaluating a new design for a crystal drinking glass; the precision of the geometry and the rendering will matter a great deal, since the reflections and sparkling colors are very important in this situation. But still, the numerical accuracy of the energy simulation need not be right, as long as the relative accuracy of the image is correct. Then there's the image made as a simulation for analysis by a machine. In this case the image must be accurate with respect to whatever criteria will be measured and whatever choice of measurement is used.

Images are for communication, and the success of an image should be measured only by how well it communicates. Sometimes too little objective accuracy can distort the message; sometimes too much accuracy can detract from the message. The reason for making a picture is to communicate something that must be said; the image should support that message and not dominate it. The medium must be chosen to fit the message.

To make effective images we need effective tools, and that is what this book is intended to provide. Every profession has its rules of thumb and tricks of the trade; in computer graphics, these bits of wisdom are described in words, equations, and programs. The

Graphics Gems series is like a general store; it's fun to drop in every once in a while and browse, uncovering unusual items with which you were unfamiliar, and seeing new applications for old ideas. When you're faced with a sticky problem, you may remember seeing just the right tool on display. Happily, our stock is in limitless supply, and as near as your bookshelf or library.

Preface

This book is a cookbook for computer graphics programmers, a kind of "Numerical Recipes" for graphics. It contains practical techniques that can help you do 2D and 3D modeling, animation, rendering, and image processing. The 52 articles, written by 54 authors worldwide, have been selected for their usefulness, novelty, and simplicity. Each article, or "Gem," presents a technique in words and formulas, and also, for most of the articles, in C or C++ code as well. The code is available in electronic form on the IBM or Macintosh floppy disk in the back pocket of the book, and is available on the Internet via FTP (see address below). The floppy disk also contains all of the code from the previous volumes: *Graphics Gems I, II,* and *III.* You are free to use and modify this code in any way you like.

A few of the Gems in this book deserve special mention because they provide implementations of particularly useful, but non-trivial algorithms. Gems IV.6 and IV.8 give very general, modular code to polygonize parametric and implicit surfaces, respectively. With these two and a polygon renderer, you could probably display 95% of all computer graphics models! Gem I.5 finds 2D Voronoi diagrams or Delaunay triangulations. These data structures are very widely used for mesh generation and other geometric operations. In the area of interaction, Gem III.1 provides code for control of orientation in 3D. This could be used in interactive 3D modelers. Finally, Gem I.8 gives code to find collisions of polyhedra, an important task in physically based modeling and animation.

This book, like the previous three volumes in the *Graphics Gems* series, lies somewhere between the media of textbook, journal, and computer bulletin board. Textbooks explain algorithms very well, but if you are doing computer graphics programming, then they may not provide what you need: an implementation. Similarly, technical journals seldom present implementations, and they are often much more theoretical than a programmer cares for. The third alternative, computer bulletin boards such as the USENET news group *comp.graphics.algorithms*, occasionally contains good code, but because they are unmoderated and unedited, they are so flooded with queries that it is tedious to find useful information. The *Graphics Gems* series is an attempt at a middle ground, where programmers worldwide can contribute graphics techniques that they have found useful, and the best of these get published. Most of the articles are written by the inventors of the techniques, so you will learn their motivations and see their programming techniques firsthand. Also, the implementations have been selected for their portability; they are not limited to UNIX, IBM, or Macintosh systems. Most of them will compile and run, perhaps with minor modifications, on any computer with a C or C++ compiler.

Assembling this book has been a collaborative process involving many people. In the Spring of 1993, a call for contributions was distributed worldwide via electronic mail and word of mouth. Submissions arrived in the Summer of 1993. These were read by me and many were also read by one or more of my outside reviewers: Eric Haines, Andrew Glassner, Chandrajit Bajaj, Tom Duff, Ron Goldman, Tom Sederberg, David Baraff, Jules Bloomenthal, Ken Shoemake, Mike Kass, Don Mitchell, and Greg Ward. Of the 155 articles submitted, 52 were accepted for publication. These were revised and, in most cases, formatted into LaTeX by the authors. Coordinating the project at Academic Press in Cambridge, Massachusetts, were Jenifer Niles and Brian Miller. Book composition was done by Rena Wells at Rosenlaui Publishing Services in Houston, Texas, and the cover image was made by Eben Ostby of Pixar, in Richmond, California. I am very thankful to all of these people and to the others who worked on this book for helping to make it a reality. Great thanks also to the *Graphics Gems* series editor, Andrew Glassner, for inviting me to be editor for this volume, and to my wife, Bridget Johnson-Heckbert, for her patience.

There are a few differences between this book and the previous volumes of the series. Organizationally, the code and bibliographies are not collected at the back of the book, but appear with the text of the corresponding article. These changes make each Gem more self-contained. The book also differs in emphasis. Relative to the previous volumes, I have probably stressed novelty more, and simplicity less, preferring an implementation of a complex computer graphics algorithm over formulas from analytic geometry, for example.

In addition to the *Graphics Gems* series, there are several other good sources for practical computer graphics techniques. One of these is the column "Jim Blinn's Corner" that appears in the journal *IEEE Computer Graphics and Applications.* Another is the book *A Programmer's Geometry,* by Adrian Bowyer and John Woodwark (Butterworth's, London, 1983), which is full of analytic geometry formulas. A mix of analytic geometry and basic computer graphics formulas is contained in the book *Computer Graphics Handbook: Geometry and Mathematics* by Michael E. Mortensen (Industrial Press, New York, 1990). Another excellent source is, of course, graphics textbooks.

Code in this book is available on the Internet by anonymous FTP from `princeton.edu` (`128.112.128.1`) in the directory `pub/GraphicsGems/GemsIV`. The code for other *Graphics Gems* books is also available nearby. Bug reports should be submitted as described in the `README` file there.

Paul Heckbert, March 1994

About the Cover

The cover: "Washday Miracle" by Eben Ostby. Copyright © 1994 Pixar.

When series editor Andrew Glassner called me to ask if I could help with a cover image for *Graphics Gems IV*, there were four requirements: the image needed to tell a story; it needed to have gems in it; it should be a computer-generated image; and it should look good. To these parameters, I added one of my own: it should tell a story that is different from the previous covers. Those stories were usually mystical or magical; accordingly, I decided to take the mundane as my inspiration.

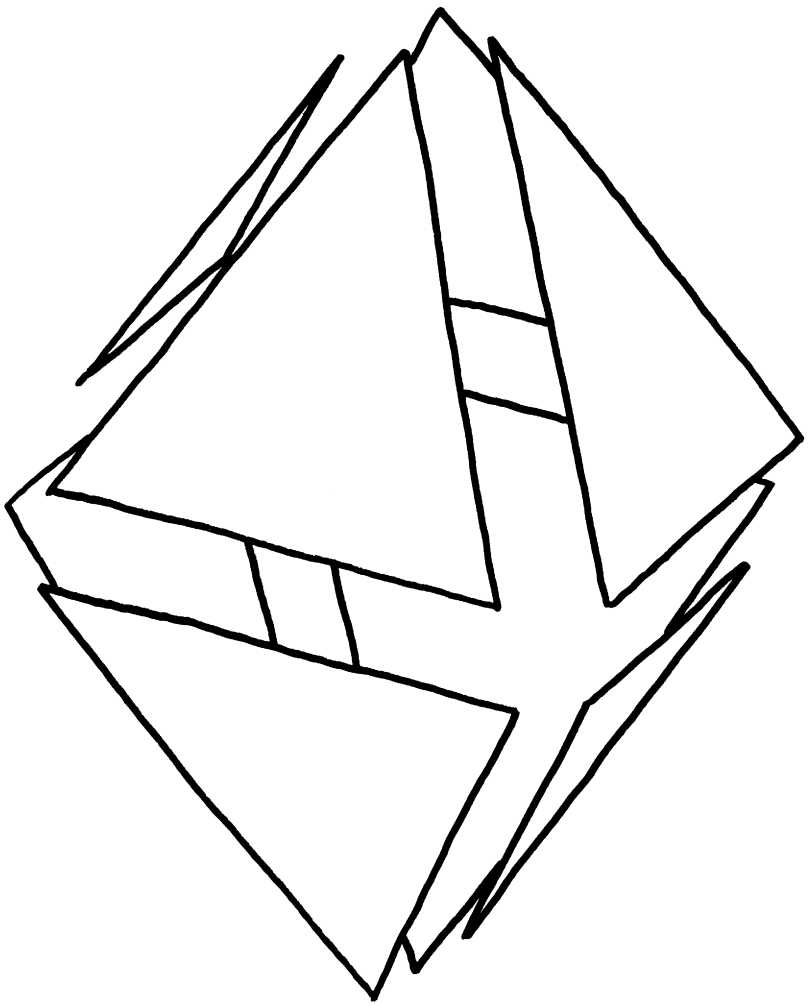
The image was created using a variety of tools, including Alias Studio; Menv, our own internal animation system; and Photorealistic RenderMan. The appliances, table, and basket were built in Alias. The gems were placed by a stochastic "gem-placer" running under Menv. The house set was built in Menv. Surface descriptions were written in the RenderMan shading language and include both procedural and painted textures.

For the number-conscious, this image was rendered at a resolution of 2048 by 2695 and contains the following:

16 lights
643 gems
30,529 lines or 2,389,896 bytes of model information
4 cycles: regular, delicate, Perma-Press, and Air Fluff

Galyn Susman did the lighting design. Andrew Glassner reviewed and critiqued, and made the image far better as a result. Matt Martin made prepress proofs. Pixar (in corpora Karen Robert Jackson and Ralph Guggenheim) permitted me time to do this.

Eben Ostby
Pixar



I

Polygons and Polyhedra

This part of the book contains five Gems on polygons and three on polyhedra. Polygons and polyhedra are the most basic and popular geometric building blocks in computer graphics.

I.1. Centroid of a Polygon, *by Gerard Bashein and Paul R. Detmer.*
Gives formulas and code to find the centroid (center of mass) of a polygon. This is useful when simulating Newtonian dynamics. Page 3.

I.2. Testing the Convexity of a Polygon, *by Peter Schorn and Frederick Fisher.*
Gives an algorithm and code to determine if a polygon is convex, non-convex (concave but not convex), or non-simple (self-intersecting). For many polygon operations, faster algorithms can be used if the polygon is known to be convex. This is true when scan converting a polygon and when determining if a point is inside a polygon, for instance. Page 7.

I.3. An Incremental Angle Point in Polygon Test, *by Kevin Weiler.*

I.4. Point in Polygon Strategies, *by Eric Haines.*
Provide algorithms for testing if a point is inside a polygon, a task known as point inclusion testing in computational geometry. Point-in-polygon testing is a basic task when ray tracing polygonal models, so these methods are useful for 3D as well as 2D graphics. Weiler presents a single algorithm for testing if a point lies in a concave polygon, while Haines surveys a number of algorithms for point inclusion testing in both convex and concave polygons, with empirical speed tests and practical optimizations. Pages 16 and 24.

I.5. Incremental Delaunay Triangulation, *by Dani Lischinski.*
Gives some code to solve a very important problem: finding Delaunay triangulations and Voronoi diagrams in 2D. These two geometric constructions are useful for triangular mesh generation and for nearest neighbor finding, respectively. Triangular mesh generation comes up when doing interpolation of surfaces from scattered data points, and in finite element simulations of all kinds, such as radiosity. Voronoi diagrams are used in many computational geometry algorithms. Page 47.

The final three Gems of this part of the book concern polyhedra: polygonal models that are intrinsically three-dimensional.

I.6. Building Vertex Normals from an Unstructured Polygon List, *by Andrew Glassner.*
Solves a fairly common rendering problem: if one is given a set of polygons in raw form, with no topological (adjacency) information, and asked to do smooth shading (Gouraud or Phong shading) of them, one must infer topology and compute vertex normals. Page 60.

I.7. Detecting Intersection of a Rectangular Solid and a Convex Polyhedron, *by Ned Greene.*
Presents an optimized technique to test for intersection between a convex polyhedron and a box. This is useful when comparing bounding boxes against a viewing frustum in a rendering program, for instance. Page 74.

I.8. Fast Collision Detection of Moving Convex Polyhedra, *by Rich Rabbitz.*
A turn-key piece of software that solves a difficult but basic problem in physically based animation and interactive modeling. Page 83.

I.1

Centroid of a Polygon

Gerard Bashein[1]
Department of Anesthesiology and
Center for Bioengineering, RN-10
University of Washington
Seattle, WA 98195
gb@locke.hs.washington.edu

Paul R. Detmer[1]
Department of Surgery and
Center for Bioengineering, RF-25
University of Washington
Seattle, WA 98195
pdetmer@u.washington.edu

This Gem gives a rapid and accurate method to calculate the area and the coordinates of the center of mass of a simple polygon.

Determination of the center of mass of a polygonal object may be required in the simulation of planar mechanical systems and in some types of graphical data analysis. When the density of an object is uniform, the center of mass is called the centroid. The naive way of calculating the centroid, taking the mean of the x and y coordinates of the vertices, gives incorrect results except in a few simple situations, because it actually finds the center of mass of a massless polygon with equal point masses at its vertices. As an example of how the naive method would fail, consider a simple polygon composed of many small line segments (and closely spaced vertices) along one side and only a few vertices along the other sides. The means of the vertex coordinates would then be skewed toward the side having many vertices.

Basic mechanics texts show that the coordinates $(\overline{x}, \overline{y})$ of the centroid of a closed planar region R are given by

$$\overline{x} = \frac{\iint_R x \, dx \, dy}{A} = \frac{\mu_x}{A} \tag{1}$$

$$\overline{y} = \frac{\iint_R y \, dx \, dy}{A} = \frac{\mu_y}{A} \tag{2}$$

where A is the area of R, and μ_x and μ_y are the first moments of R along the x- and y-coordinates, respectively.

In the case where R is a polygon given by the points (x_i, y_i), $i = 0, \ldots, n$, with $x_0 = x_n$ and $y_0 = y_n$, (Roberts 1965) and later (Rokne 1991), (Goldman 1991), and others have shown a rapid method for calculating its area based upon Green's theorem in a plane.

[1]Supported by grants HL42270 and HL41464 from the National Institutes of Health, Bethesda, MD.

IBM ISBN 0-12-336155-9
Macintosh ISBN 0-12-336156-7

$$A = \frac{1}{2}\sum_{i=0}^{n-1} a_i, \quad \text{where } a_i = x_i y_{i+1} - x_{i+1} y_i$$

Janicki *et al.* have also shown that the first moments μ_x and μ_y of a polygon can also be found by Green's theorem (Janicki *et al.* 1981), which states that given continuous functions $M(x, y)$ and $N(x, y)$ having continuous partial derivatives over a region R, which is enclosed by a contour C,

$$\int_C (M\,dx + N\,dy) = \iint_R \left(\frac{\partial N}{\partial x} - \frac{\partial M}{\partial y}\right) dx\,dy \tag{3}$$

To evaluate the numerator of (1), let $M = 0$ and $N = \frac{1}{2}x^2$. Then the right side of (3) equals μ_x, and the first moment can be calculated as

$$\mu_x = \frac{1}{2}\int_C x^2 dy$$

Then, representing the line segments between each vertex parametrically and summing the integrals over each line segment yields

$$\mu_x = \frac{1}{6}\sum_{i=0}^{n-1} (x_{i+1} + x_i) \cdot a_i$$

Similarly, to evaluate the numerator of (2), let $M = -\frac{1}{2}y^2$ and $N = 0$, and evaluate the left side of (3). The result becomes

$$\mu_y = \frac{1}{6}\sum_{i=0}^{n-1} (y_{i+1} + y_i) \cdot a_i$$

The form of the equations given above is particularly suited for numerical computation, because it takes advantage of a common factor in the area and moments, and because it eliminates one subtraction (and the consequent loss of accuracy) from each term of the summation for the moments. The loss of numerical accuracy due to the remaining subtraction can be reduced if, before calculating the centroid, the coordinate system is translated to place its origin somewhere close to the polygon.

The techniques used above can be generalized to find volumes, centroids, and moments of inertia of polyhedra (Lien and Kajiya 1984).

The following C code will calculate the x- and y-coordinates of the centroid and the area of any simple (non–self-intersecting) convex or concave polygon. The algebraic signs of both the area (output by the function) and first moments (internal variables only) will be positive when the vertices are ordered in a counterclockwise direction in the x–y plane, and negative otherwise. The coordinates of the centroid will have the

correct signs in either case. The method of computation is algebraically equivalent to breaking the polygon into component triangles, finding their signed areas and centroids, and combining the results. Non-simple polygons will have the contributions of their overlapping regions to the area and moments summed algebraically according to the direction (clockwise or counterclockwise) of each traversal of each region.

◊ C Code ◊

```
/*********************************************************************
polyCentroid: Calculates the centroid (xCentroid, yCentroid) and area
of a polygon, given its vertices (x[0], y[0]) ... (x[n-1], y[n-1]). It
is assumed that the contour is closed, i.e., that the vertex following
(x[n-1], y[n-1]) is (x[0], y[0]).  The algebraic sign of the area is
positive for counterclockwise ordering of vertices in x-y plane;
otherwise negative.

Returned values:  0 for normal execution;  1 if the polygon is
degenerate (number of vertices < 3);  and 2 if area = 0 (and the
centroid is undefined).
*********************************************************************/
int polyCentroid(double x[], double y[], int n,
                 double *xCentroid, double *yCentroid, double *area)
     {
     register int i, j;
     double ai, atmp = 0, xtmp = 0, ytmp = 0;
     if (n < 3) return 1;
     for (i = n-1, j = 0; j < n; i = j, j++)
          {
          ai = x[i] * y[j] - x[j] * y[i];
          atmp += ai;
          xtmp += (x[j] + x[i]) * ai;
          ytmp += (y[j] + y[i]) * ai;
          }
     *area = atmp / 2;
     if (atmp != 0)
          {
          *xCentroid =  xtmp / (3 * atmp);
          *yCentroid =  ytmp / (3 * atmp);
          return 0;
          }
     return 2;
     }
/***** end polyCentroid *****/
```

◊ Bibliography ◊

(Goldman 1991) Ronald N. Goldman. Area of planar polygons and volume of polyhedra. In James Arvo, ed., *Graphics Gems II*, pages 170–171. Academic Press, Boston, MA, 1991.

(Janicki *et al.* 1981) Joseph S. Janicki *et al.* Three-dimensional myocardial and ventricular shape: A surface representation. *Am. J. Physiol.*, 241:H1–H11, 1981.

(Lien and Kajiya 1984) S. Lien and J. T. Kajiya. A symbolic method for calculating the integral properties of arbitrary nonconvex polyhedra. *IEEE Computer Graphics & Applications*, 4(10):35–41, 1984.

(Roberts 1965) L. G. Roberts. Machine perception of three-dimensional solids. In J. P. Tippet *et al.*, eds., *Optical and Electro-Optical Information Processing.* MIT Press, Cambridge, MA, 1965.

(Rokne 1991) Jon Rokne. The area of a simple polygon. In James Arvo, ed., *Graphics Gems II*, pages 5–6. Academic Press, Boston, MA, 1991.

◊ I.2

Testing the Convexity of a Polygon

Peter Schorn
Institut für Theoretische Informatik
ETH, CH-8092 Zürich, Switzerland
schorn@inf.ethz.ch

Frederick Fisher
2630 Walsh Avenue
Kubota Pacific Computer, Inc.
Santa Clara, CA
fred@kpc.com

◊ Abstract ◊

This article presents an algorithm that determines whether a polygon given by the sequence of its vertices is convex. The algorithm is implemented in C, runs in time proportional to the number of vertices, needs constant storage space, and handles all degenerate cases, including non-simple (self-intersecting) polygons.

Results of a polygon convexity test are useful to select between various algorithms that perform a given operation on a polygon. For example, polygon classification could be used to choose between point-in-polygon algorithms in a ray tracer, to choose an output rasterization routine, or to select an algorithm for line-polygon clipping or polygon-polygon clipping. Generally, an algorithm that can assume a specific polygon shape can be optimized to run much faster than a general routine.

Another application would be to use this classification scheme as part of a filter program that processes input data, such as from a tablet. Results of the filter could eliminate complex polygons so that following routines may assume convex polygons.

◊ Issues in Solving the Problem ◊

The problem whose solution this article describes started out as a posting on the USENET bulletin board 'comp.graphics' which asked for a program that could decide whether a polygon is convex. Answering this question turned into a contest, managed by Kenneth Sloan, which aimed at the construction of a correct and efficient program. The most important issues discussed were:

- Correctness, especially in degenerate cases. Many people quickly succeeded in writing a program which could handle *almost* all cases. The challenge was a program which works in all, even degenerate, cases. Some degenerate examples are depicted in Figure 1.

IBM ISBN 0-12-336155-9
Macintosh ISBN 0-12-336156-7

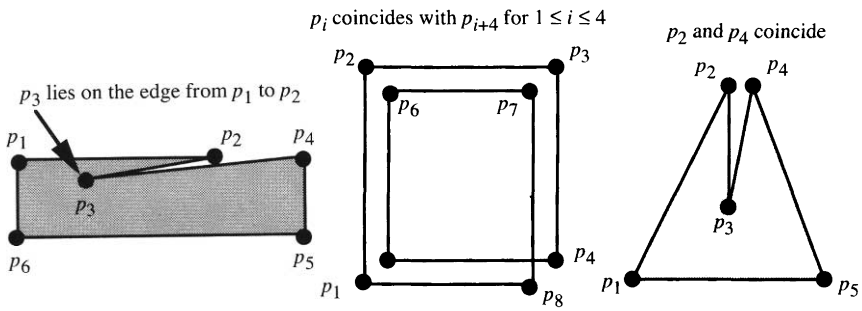


Figure 1. Some degenerate cases.

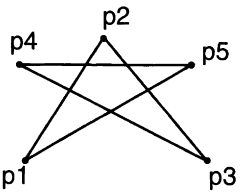


Figure 2. Non-convex polygon with a right turn at each vertex.

Although the first two examples might be considered convex (their interior is indeed convex), a majority of the participants in the discussion agreed that these cases should be considered not convex. Further complications are "all points collinear" and "repeated points."

- What is a convex polygon? This question is very much related to correctness and a suitable definition of a convex polygon was a hotly debated topic. When one thinks about the problem for the first time, a common mistake is to require a right turn at each vertex and nothing else. This leads to the counterexample in Figure 2.
- Efficiency. The program should run in time proportional to the number of vertices. Furthermore, only constant space for the program was allowed. This required a solution to read the polygon vertices from an input stream without saving them.
- Imprecise arithmetic. The meaning of "three points are collinear" becomes unclear when the coordinates of the points are only approximately correct or when floating-point arithmetic is used to test for collinearity or right turns. This article assumes exact arithmetic in order to avoid complications.

◇ What Is a Convex Polygon? ◇

Answering this question is an essential step toward the construction of a robust program. There are at least four approaches:

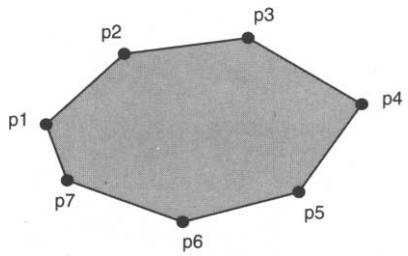


Figure 3. An undisputed convex polygon.

- The cavalier attitude: I know what a convex polygon is when I see one. For example the polygon in Figure 3 is clearly convex.
- The "what works for me" approach: A polygon P is convex if my triangulation routine (renderer, etc.) which expects convex polygons as input can handle P.
- The "algorithm as definition" approach: A polygon is convex if my convexity testing program declares it as such.
- A more abstract, mathematical approach starting with the definition of a convex set: A set S of points is convex $\Leftrightarrow$

$$(p \in S) \wedge (q \in S) \Rightarrow \forall \lambda : 0 \leq \lambda \leq 1 : \lambda \cdot p + (1 - \lambda) \cdot q \in S$$

  This roughly means that a set of points S is convex iff for any line drawn between two points in the set S, then all points on the line segment are also in the set.

In the following we propose a different, formal approach, which has the following advantages:

- It captures the intuition about a convex polygon.
- It gives a reasonable answer in degenerate cases.
- It distinguishes between clockwise- and counterclockwise orientations.
- It leads to a correct and efficient algorithm.

Classification: Given a sequence $P = p_1, p_2, \ldots, p_n$ of points in the plane such that

1. n is an integer and $(n > 0)$.
2. Consecutive vertices are different. $p_i \neq p_{i+1}$ for $1 \leq i \leq n$ (we assume $p_{n+1} = p_1$).
3. We restrict consideration to sequences where p_1 is lexicographically the smallest, i.e., $p_1 < p_i$ for $2 \leq i \leq n$ where $p < q \Leftrightarrow (p_x < q_x) \vee ((p_x = q_x) \wedge (p_y < q_y))$.
4. All convex polygons are monotone polygons, that is the x-coordinate of the points increases monotonically and then decreases monotonically. p_j is the "rightmost vertex."
   $\exists j : 1 < j \leq n : p_i < p_{i+1}$ for $1 \leq i < j$ and $p_{i+1} < p_i$ for $j \leq i \leq n$

Then if $p_i = [X_i, Y_i]$, and

$$d(i) = (X_{i-1} - X_i) \cdot (Y_i - Y_{i+1}) - (Y_{i-1} - Y_i) \cdot (X_i - X_{i+1})$$

P denotes a left- (counterclockwise) convex polygon $\Leftrightarrow$

$$(\forall i : 1 \leq i \leq n : d(i) \leq 0) \wedge (\exists i : 1 \leq i \leq n : d(i) < 0)$$

P denotes a right- (clockwise) convex polygon $\Leftrightarrow$

$$(\forall i : 1 \leq i \leq n : d(i) \geq 0) \wedge (\exists i : 1 \leq i \leq n : d(i) > 0)$$

P denotes a degenerate-convex polygon $\Leftrightarrow$

$$\forall i : 1 \leq i \leq n : d(i) = 0$$

P denotes a non-convex polygon $\Leftrightarrow$

$$(\exists i : 1 \leq i \leq n : d(i) < 0) \wedge (\exists i : 1 \leq i \leq n : d(i) > 0)$$

This classification of vertex-sequences agrees with our intuition for convex polygons (see Figure 3). For clockwise convex polygons there is a right turn at each vertex, and for counterclockwise convex polygons there is a left turn at each vertex. If the points satisfy condition 4 but lie on a line, the polygon is classified as degenerate-convex.

For purposes of simplifying the classification, conditions 2, 3, and 4 constrain the possible polygons. However, the classification can be extended to sequences not satisfying conditions 2, 3, or 4. Any sequence can easily meet conditions 2 and 3 if we remove consecutive duplicate points and perform a cyclic shift, moving the lexicographically smallest point to the beginning of the sequence. If condition 4 cannot be met, the sequence denotes a non-convex polygon.

◊ Implementation in C ◊

The following C program shows how the classification scheme can be turned into a correct and efficient implementation. The program accepts lines which contain two numbers, denoting the x- and y-coordinates of a point (see the function `GetPoint`). Duplicate points are removed on the fly (see the function `GetDifferentPoint`).

Since we do not want to store more than a constant number of points, we cannot perform a cyclic shift of the input vertices in order to assure condition 3. Instead, the program counts how often the lexicographic order of the input vertices changes. If this number exceeds two, the input polygon is definitely not convex.

In addition to the four cases distinguished in the classification scheme, the program introduces a fifth case (`NotConvexDegenerate`) for polygons whose vertices all lie on a line but do not satisfy condition 4.

♢ Program to Classify a Polygon's Shape ♢

```
#include <stdio.h>

typedef enum { NotConvex, NotConvexDegenerate,
               ConvexDegenerate, ConvexCCW, ConvexCW } PolygonClass;

typedef struct { double x, y; } Point2d;

int WhichSide(p, q, r)          /* Given a directed line pq, determine  */
Point2d       p, q, r;          /* whether qr turns CW or CCW.          */
{
    double result;
    result = (p.x - q.x) * (q.y - r.y) - (p.y - q.y) * (q.x - r.x);
    if (result < 0) return -1;  /* q lies to the left  (qr turns CW).   */
    if (result > 0) return  1;  /* q lies to the right (qr turns CCW).  */
    return 0;                   /* q lies on the line from p to r.      */
}

int Compare(p, q)               /* Lexicographic comparison of p and q  */
Point2d     p, q;
{
    if (p.x < q.x) return -1;   /* p is less than q.                    */
    if (p.x > q.x) return  1;   /* p is greater than q.                 */
    if (p.y < q.y) return -1;   /* p is less than q.                    */
    if (p.y > q.y) return  1;   /* p is greater than q.                 */
    return 0;                   /* p is equal to q.                     */
}

int GetPoint(f, p)              /* Read p's x- and y-coordinates from f */
FILE    *f;                     /* and return true, iff successful.     */
Point2d *p;
{
    return !feof(f) && (2 == fscanf(f, "%lf%lf", &(p->x), &(p->y)));
}

int GetDifferentPoint(f, previous, next)
FILE    *f;                     /* Read next point into 'next' until it */
Point2d previous, *next;        /* is different from 'previous' and     */
{                               /* return true iff successful.          */
    int eof;
    while((eof = GetPoint(f, next)) && (Compare(previous, *next) == 0));
    return eof;
}

/* CheckTriple tests three consecutive points for change of direction
 * and for orientation.
 */
#define CheckTriple                                                     \
        if ( (thisDir = Compare(second, third)) == -curDir )            \
            ++dirChanges;                                               \
        curDir = thisDir;                                               \
        if ( thisSign = WhichSide(first, second, third) ) {             \
```

```
            if ( angleSign == -thisSign )                                   \
                return NotConvex;                                           \
            angleSign = thisSign;                                           \
        }                                                                   \
        first = second; second = third;

/* Classify the polygon vertices on file 'f' according to: 'NotConvex'  */
/* 'NotConvexDegenerate', 'ConvexDegenerate', 'ConvexCCW', 'ConvexCW'.  */
PolygonClass ClassifyPolygon(f)
FILE                        *f;
{
    int          curDir, thisDir, thisSign, angleSign = 0, dirChanges = 0;
    PolygonClass result;
    Point2d      first, second, third, saveFirst, saveSecond;

    if ( !GetPoint(f, &first) || !GetDifferentPoint(f, first, &second) )
        return ConvexDegenerate;
    saveFirst = first;  saveSecond = second;
    curDir = Compare(first, second);
    while( GetDifferentPoint(f, second, &third) ) {
        CheckTriple;
    }
    /* Must check that end of list continues back to start properly. */
    if ( Compare(second, saveFirst) ) {
        third = saveFirst; CheckTriple;
    }
    third = saveSecond;  CheckTriple;

    if ( dirChanges > 2 ) return angleSign ? NotConvex : NotConvexDegenerate;
    if ( angleSign  > 0 ) return ConvexCCW;
    if ( angleSign  < 0 ) return ConvexCW;
    return ConvexDegenerate;
}

int main()
{
    switch ( ClassifyPolygon(stdin) ) {
        case NotConvex:           fprintf( stderr,"Not Convex\n");
                                  exit(-1); break;
        case NotConvexDegenerate: fprintf( stderr,"Not Convex Degenerate\n");
                                  exit(-1); break;
        case ConvexDegenerate:    fprintf( stderr,"Convex Degenerate\n");
                                  exit( 0); break;
        case ConvexCCW:           fprintf( stderr,"Convex Counter-Clockwise\n");
                                  exit( 0); break;
        case ConvexCW:            fprintf( stderr,"Convex Clockwise\n");
                                  exit( 0); break;
    }
}
```

♢ Optimizations ♢

The previous code was chosen for its conciseness and readability. Other versions of the code were written which accept a vertex count and pointer to an array of vertices. Given this interface, it is possible to obtain good performance measurements by timing a large number of calls to the polygon classification routine.

Variations of the code presented have resulted in a two to four times performance increase, depending on the polygon shape. Optimizations for a particular machine or programming language will undoubtedly produce different results. Some considerations are:

- Convert each of the routines to macro definitions.
- Instead of keeping track of the first, second, and third points, keep track of the previous delta (second − first), and a current delta (third − second). This will speed up parts of the algorithm: The macro `Compare` needs only compare two numbers with zero, instead of four numbers with each other; the routine for getting a different point calculates the delta as it determines if the new point is different; the cross product calculation uses the deltas directly instead of subtracting vertices each time; the comparison for the `WhichSide` routine may be moved up to the `CheckTriple` routine to save a comparison at the expense of a little more code; and preparing to examine the next point requires three moves instead of four.
- Checking for less than three vertices is possible, but generally slows down the other cases.
- Every time the variable `dirChanges` is incremented, it would be possible to check if the number is now greater than two. This will slow down the convex cases, but makes it possible to exit early for polygons which violate classification condition 4. If it is important to distinguish between `NotConvex` and `NotConvexDegenerate`, this optimization may not be used.

♢ Reasonably Optimized Routine to Classify a Polygon's Shape ♢

```
/*

. . . code omitted which reads polygon, stores in an array, and calls
        classifyPolygon2()
*/

typedef double  Number;          /* float or double */

#define ConvexCompare(delta)                                          \
    ( (delta[0] > 0) ? -1 :      /* x coord diff, second pt > first pt */\
```

```
      (delta[0] < 0) ?  1 :     /* x coord diff, second pt < first pt */\
      (delta[1] > 0) ? -1 :     /* x coord same, second pt > first pt */\
      (delta[1] < 0) ?  1 :     /* x coord same, second pt > first pt */\
      0 )                       /* second pt equals first point */

#define ConvexGetPointDelta(delta, pprev, pcur )                          \
    /* Given a previous point 'pprev', read a new point into 'pcur' */   \
    /* and return delta in 'delta'.                                  */   \
    pcur = pVert[iread++];                                                \
    delta[0] = pcur[0] - pprev[0];                                        \
    delta[1] = pcur[1] - pprev[1];                                        \

#define ConvexCross(p, q) p[0] * q[1] - p[1] * q[0];

#define ConvexCheckTriple                                                 \
    if ( (thisDir = ConvexCompare(dcur)) == -curDir ) {                   \
          ++dirChanges;                                                   \
          /* The following line will optimize for polygons that are  */ \
          /* not convex because of classification condition 4,       */ \
          /* otherwise, this will only slow down the classification. */ \
          /* if ( dirChanges > 2 ) return NotConvex;                 */ \
    }                                                                     \
    curDir = thisDir;                                                     \
    cross = ConvexCross(dprev, dcur);                                     \
    if ( cross > 0 ) { if ( angleSign == -1 ) return NotConvex;           \
                       angleSign = 1;                                     \
                     }                                                    \
    else if (cross < 0) { if (angleSign == 1) return NotConvex;           \
                          angleSign = -1;                                 \
                        }                                                 \
    pSecond = pThird;           /* Remember ptr to current point. */      \
    dprev[0] = dcur[0];         /* Remember current delta.        */      \
    dprev[1] = dcur[1];                                                   \

classifyPolygon2( nvert, pVert )
int     nvert;
Number  pVert[][2];
/* Determine polygon type. return one of:
 *      NotConvex, NotConvexDegenerate,
 *      ConvexCCW, ConvexCW, ConvexDegenerate
 */
{
    int      curDir, thisDir, dirChanges = 0,
             angleSign = 0, iread, endOfData;
    Number   *pSecond, *pThird, *pSaveSecond, dprev[2], dcur[2], cross;

    /* if ( nvert <= 0 ) return error;       if you care */

    /* Get different point, return if less than 3 diff points. */
    if ( nvert < 3 ) return ConvexDegenerate;
    iread = 1;
    while ( 1 ) {
        ConvexGetPointDelta( dprev, pVert[0], pSecond );
```

```
        if ( dprev[0] || dprev[1] ) break;
        /* Check if out of points. Check here to avoid slowing down cases
         * without repeated points.
         */
        if ( iread >= nvert ) return ConvexDegenerate;
    }

    pSaveSecond = pSecond;

    curDir = ConvexCompare(dprev);      /* Find initial direction */

    while ( iread < nvert ) {
        /* Get different point, break if no more points */
        ConvexGetPointDelta(dcur, pSecond, pThird );
        if ( dcur[0] == 0.0  &&  dcur[1] == 0.0 ) continue;

        ConvexCheckTriple;              /* Check current three points */
    }

    /* Must check for direction changes from last vertex back to first */
    pThird = pVert[0];                  /* Prepare for 'ConvexCheckTriple' */
    dcur[0] = pThird[0] - pSecond[0];
    dcur[1] = pThird[1] - pSecond[1];
    if ( ConvexCompare(dcur) ) {
        ConvexCheckTriple;
    }

    /* and check for direction changes back to second vertex */
    dcur[0] = pSaveSecond[0] - pSecond[0];
    dcur[1] = pSaveSecond[1] - pSecond[1];
    ConvexCheckTriple;                  /* Don't care about 'pThird' now */

    /* Decide on polygon type given accumulated status */
    if ( dirChanges > 2 )
        return angleSign ? NotConvex : NotConvexDegenerate;

    if ( angleSign > 0 ) return ConvexCCW;
    if ( angleSign < 0 ) return ConvexCW;
    return ConvexDegenerate;
}
```

◇ Acknowledgments ◇

We are grateful to the participants of the electronic mail discussion: Gavin Bell, Wayne Boucher, Laurence James Edwards, Eric A. Haines, Paul Heckbert, Steve Hollasch, Torben Ægidius Mogensen, Joseph O'Rourke, Kenneth Sloan, Tom Wright, and Benjamin Zhu.

◊ I.3

An Incremental Angle Point in Polygon Test

Kevin Weiler
Autodesk Inc.
2320 Marinship Way
Sausalito, CA 94965
kjw@autodesk.com

This algorithm can determine whether a given test point is inside of, or outside of, a given polygon boundary composed of straight line segments. The algorithm is not sensitive to whether the polygon is concave or convex or whether the polygon's vertices are presented in a clockwise or counterclockwise order. Extensions allow the algorithm to handle polygons with holes and non-simple polygons. Only four bits of precision are required for all of the incremental angle calculations.

◊ Introduction ◊

There are two commonly used algorithms for determining whether a given test point is inside or outside of a polygon.

The first, the semi-infinite line technique, extends a semi-infinite line from the test point outward, and counts the number of intersections of the edges of the polygon boundary with the semi-infinite line. An odd number of intersections indicates the point is inside the polygon, while an even number (including zero) indicates the point is outside the polygon.

The second, the incremental angle technique, uses the angle of the vertices of the polygon relative to the point being tested, where there is a total angle of 360 degrees all the way around the point. For each vertex of the polygon, the difference angle (the incremental angle) between the angle of that vertex of the polygon and the angle of the next vertex of the polygon, as viewed from the test point, is added to a running sum. If the final sum of the incremental angles is plus or minus 360 degrees, the polygon surrounds the test point and the point is inside of the polygon. If the sum is 0 degrees, the point is outside of the polygon.

What is less commonly known about the incremental angle technique is that only four bits of precision are required for all of the incremental angle calculations, greatly simplifying the necessary calculations. The angle value itself requires only two bits of precision, lending itself to a quadrant technique where the quadrants are numbered

IBM ISBN 0-12-336155-9
Macintosh ISBN 0-12-336156-7

from 0 to 3. The incremental or delta angle requires an additional sign bit to indicate clockwise or counterclockwise direction, for a total of three bits to represent the incremental angle itself. The accumulated angle requires four bits total: three to represent the magnitude, ranging from 0 to 4, plus a sign bit.

The following algorithm describes a four-bit precision incremental angle point in polygon test. Extensions for polygons with holes and for degenerate polygons are also described.

The algorithm described was inspired by the incremental angle surrounder test sometimes used for the Warnock hidden surface removal algorithm. That surrounder algorithm determines if a polygon surrounds rectangular screen areas by partitioning the space around the rectangular window using an eight neighbor partitioning technique (Newman and Sproull 1973, pp. 520–521, 526–527), (Rogers 1985, pp. 249–251). If one shrinks the central rectangular window of that partitioning scheme down to a point (shrinking the rectangular partitions directly above and below and to the left and right of the window down to lines), the partitioning becomes a quadrant style division of the space around the point. This reduces the precision of angle calculations needed and simplifies the algorithm to the point in polygon test presented here.

Further discussion and comparisons of point in polygon techniques can be found in Eric Haines' article in this volume (Haines 1994).

◇ Preliminaries ◇

For sake of completeness, before describing the algorithm, simple type definitions used in the following code as well as a typical definition for a polygon representation are given below.

```
        /* type for quadrant id's, incremental angles, accumulated angle values */
typedef short quadrant_type;

        /* type for result value from point in polygon test */
typedef enum pt_poly_relation {INSIDE, OUTSIDE} pt_poly_relation;

        /* polygon vertex definition */
typedef struct vertex_struct {
  double x,y;                    /* coordinate values */
  struct vertex_struct  *next;   /* circular singly linked list from poly */
  } vertex, *vertex_ptr;

        /* polygon definition */
typedef struct polygon_struct {
  vertex_ptr last;               /* pointer to end of circular vertex list */
  } polygon, *polygon_ptr;

        /* polygon vertex access */
#define polygon_get_vertex(poly, vertex) \
    ((vertex == NULL) ? poly->last->next : vertex->next)
```

The quadrant and return result types are self-explanatory.

Polygon vertices are regarded as structures that allow direct access of the X and Y coordinate values in C via `vertex->x` and `vertex->y` structure member dereferencing.

Polygons are treated here as objects that have a single access routine: `polygon_get_vertex(poly, vertex)`, where `poly` specifies a pointer to the polygon. If `vertex` is `NULL`, the function will return a pointer to an arbitrary vertex of the polygon. Otherwise, if `vertex` is a pointer to a given vertex of the polygon, the function will return a pointer to the next vertex in the ordered circular list of vertices of the polygon. Given the list representation of the polygons as described here, polygon vertices are regarded as unique even if their coordinate values are not.

◇ The Algorithm ◇

The basic idea of the algorithm, as previously stated, is to accumulate the sum of the incremental angles between the vertices of the polygon as viewed from the test point, and then see if the angles add up to the logical equivalent of a full 360 degrees, meaning the point is surrounded by the polygon.

The algorithm is presented here in four small pieces. First, a macro to determine the quadrant angle of a polygon vertex is presented. Second, a macro to determine x-intercepts of polygon edges is presented. Third, a macro to adjust the angle delta is presented. Fourth, the main point in polygon test routine is presented.

First, the angle can be calculated using only two bits of precision with a simple quadrant technique to determine the two-bit value of the angle, where x and y are the coordinates of the test point (Figure 1).

```
    /* determine the quadrant of a polygon point relative to the test point */
#define quadrant(vertex, x, y) \
  ( (vertex->x > x) ? ((vertex->y > y) ? 0 : 3) : ( (vertex->y > y) ? 1 : 2) )
```

This classifies the space around the test point into four quadrants. Since the test used to determine the quadrant uses greater-than operations, the quadrant boundaries, shown as solid lines in the diagram, lie just above and to the right of the axes centered on the coordinates of the test point, as shown with dotted lines in the figure.

In some situations it is important to determine whether the polygon edge passes to the right of or to the left of the test point. This can be determined from the x-intercept value of the polygon edge where it intersects the infinite horizontal line passing through the y value of the test point. The x-intercept can be calculated with:

```
    /* determine x-intercept of a polygon edge
       with a horizontal line at the y value of the test point */
#define x_intercept(pt1, pt2,  yy) \
  (pt2->x - ( (pt2->y - yy) * ((pt1->x - pt2->x) / (pt1->y - pt2->y)) ) )
```

It should be noted that this x-intercept code is not a general implementation as it ignores division by zero, which occurs when the y coordinate difference is zero. The

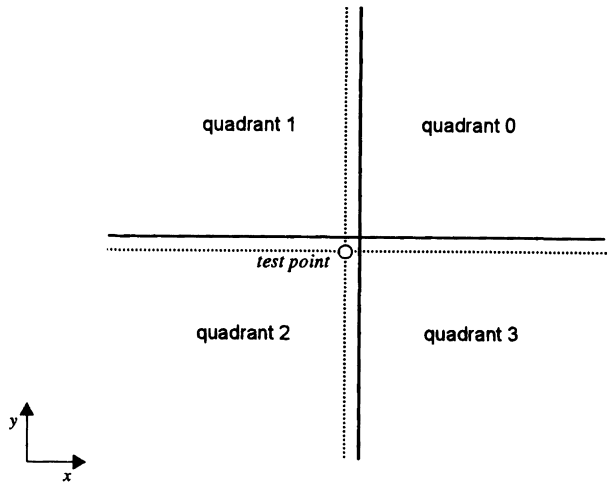


Figure 1. Quadrants.

implementation is adequate for our purposes here, however, as it will never be called under this condition.

The incremental angle itself is calculated simply by subtracting the quadrant value (angle) of one polygon vertex from the quadrant value of the next vertex. There are a few problems with this approach that must be fixed. First, because of the quadrant numbering scheme, incremental angles that cross between quadrant 0 and quadrant 3 have values of 3 instead of the proper value of 1 and the signs are also reversed. This can be fixed with a simple substitution of values. Second, an incremental angle that passes from a given quadrant to the diagonal quadrant will have its sign reversed if it passes to the right of the test point. This must be tested for by checking the x-intercept of any delta which has a value of plus or minus 2. If it passes to the right of the test point, its sign is reversed and thus must be adjusted. These adjustments are illustrated in Figure 2 and the code below. Only one of the two sets of diagonals is shown in the diagram.

```
#define adjust_delta(delta, vertex, next_vertex, xx, yy)              \
  switch (delta) {                                                    \
            /* make quadrant deltas wrap around */                    \
    case  3:    delta = -1; break;                                    \
    case -3:    delta =  1; break;                                    \
            /* check if went around point cw or ccw */               \
    case  2: case -2: if (x_intercept(vertex, next_vertex, yy) > xx)  \
                  delta =  - (delta);                                 \
              break;                                                  \
    }
```

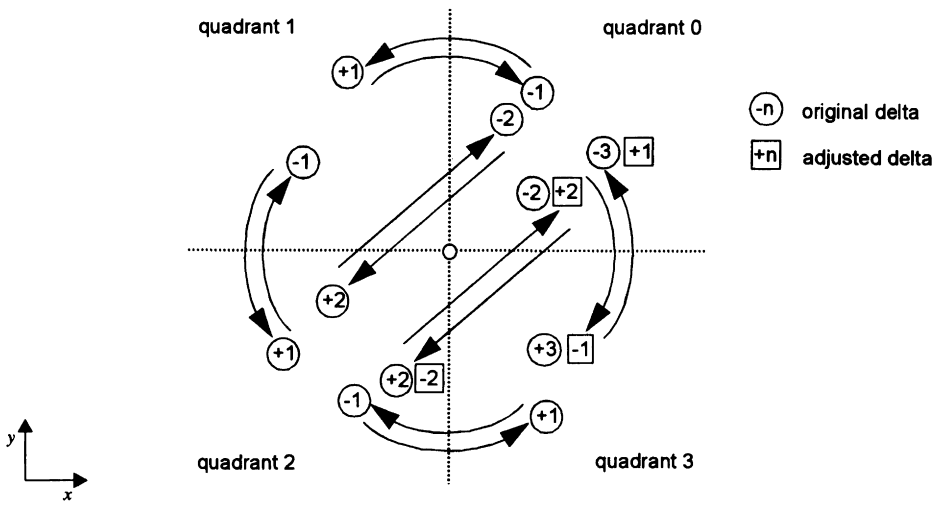


Figure 2. Adjusting delta.

A side effect of the quadrant numbering scheme is that, when adjusted, the sign of the delta value indicates whether the angle moves in a clockwise or counterclockwise direction, depending on the orientation of the coordinate axis being used. The sign of the final accumulated angle therefore also indicates the orientation of the polygon.

With these support macros and definitions out of the way, the point in polygon algorithm itself becomes simple.

In its initialization section, the algorithm prepares for traversal of the polygon points by finding the first vertex and calculating its angle from the test point, and zeroing the running sum angle. Then the algorithm loops on all of the other points (vertices) in the polygon. During the loop, the next vertex is found, its angle calculated, and the delta angle is calculated. The delta is then adjusted as necessary and added to the running sum. The loop then prepares for the next iteration. When all the points of the polygon have been seen and the loop terminated, the value of the running sum is checked. If it is equal to plus or minus 4, the angle covers a full 360 degrees and the point is inside of the polygon boundary. Otherwise the angle value is 0 and the test point is outside of the boundary of the polygon.

If the test point is actually on the polygon boundary itself, the result returned by the algorithm could be inside or outside depending on whether the actual interior of the polygon was to the right or left of the test point.

It is interesting to compare this incremental angle approach with the semi-infinite line approach. When examined closely, operation by operation, the incremental angle algorithm presented here is very similar to the semi-infinite line technique. In general,

the incremental angle method takes a constant amount of time per vertex regardless of axis crossings of the polygon edges (the exception is when the vertices of the polygon edge are in diagonal quadrants, which takes the same amount of time for both approaches). The semi-infinite line technique performs more operations when its preferred axis is crossed, and fewer operations when the other axis is crossed. To put it a different way, the semi-infinite line technique has both deeper and shallower code branch alternatives than the incremental angle technique presented depending on whether its preferred axis is crossed or not. Because of this variable behavior, worst case scenarios can be constructed to make either algorithm perform better than the other.

Performance comparisons done by Haines (Haines 1994) give statistics that show the incremental angle technique presented here to be slower than the semi-infinite line technique. Some of this performance difference will be reduced if the C compiler performs case statement optimizations which utilize indexed jump tables.

```
        /* determine if a test point is inside of or outside of a polygon */
        /* polygon is "poly", test point is at "x","y" */
pt_poly_relation
point_in_poly(polygon_ptr poly, double x, double y)
{
  vertex_ptr  vertex, first_vertex, next_vertex;
  quadrant_type quad, next_quad, delta, angle;

            /* initialize */
  vertex = NULL;    /* because polygon_get_vertex is a macro */
  vertex = first_vertex = polygon_get_vertex(poly,vertex);
  quad = quadrant(vertex, x, y);
  angle = 0;
            /* loop on all vertices of polygon */
  do {
    next_vertex = polygon_get_vertex(poly,vertex);
                /* calculate quadrant and delta from last quadrant */
    next_quad = quadrant(next_vertex, x, y);
    delta = next_quad - quad;
    adjust_delta(delta,vertex,next_vertex,x,y);
                /* add delta to total angle sum */
    angle = angle + delta;
                /* increment for next step */
    quad = next_quad;
    vertex = next_vertex;
    } while (vertex != first_vertex);

            /* complete 360 degrees (angle of + 4 or -4 ) means inside */
  if ((angle == +4) || (angle == -4)) return INSIDE; else return OUTSIDE;
}
```

◊ Extension for Polygons with Holes ◊

In order to determine whether a test point is inside of or outside of a polygon which has holes, the point in polygon test needs to be applied separately to each of the polygon's boundaries. It is preferable to start with the outermost boundary of the polygon, since the polygon's area is in most applications likely to be smaller than the total area in which the test point might lie. If the test point is outside of this polygon boundary, then it is outside of the entire polygon. If it is inside, then each hole boundary needs to be checked. If the test point is inside any of the hole boundaries, then the test point is outside of the entire polygon and checking can stop immediately. If the test point is outside of every hole boundary (as well as being inside the outermost boundary), then the point is inside of the polygon. Note that because the point in polygon test presented is insensitive to whether the polygon boundaries are clockwise or counterclockwise, both the outermost polygon boundary and the hole boundaries may be of any orientation. For polygons with holes, however, the algorithm must be told which boundary is the outermost boundary (some polygon representations encode this information in the orientation of the boundaries).

◊ Extensions for Non-Simple Polygons ◊

Non-simple polygons (polygons which self-intersect, with boundaries which touch, cross, or overlap themselves) are handled by the algorithm with minor modifications to the final test of the accumulated angle. The final angle value test:

```
if ((angle == +4) || (angle == -4)) return INSIDE; else return OUTSIDE;
```

must be modified to handle non-simple polygons properly in all cases. Two different rules are commonly used to determine the interior of non-simple polygons (there are also others, but they are less common because their implementations are more difficult). Both rules allow the non-simple polygon to completely surround the point an arbitrary number of times.

With the first rule, the odd winding number rule, if the number of surroundings is odd, then the point is inside. An even number indicates the point is outside the polygon. The code for this is:

```
if (angle & 4) return INSIDE; else return OUTSIDE; /* odd number windings rule */
```

where an odd number of surroundings means that the 4-bit in the angle value will be set since a valid angle value, unless it is 0, will be a multiple of 4.

The second rule, the non-zero winding number rule, accepts any number of surroundings to mean the point is in the interior of the polygon. With this rule, the final angle value test becomes:

```
if (angle != 0) return INSIDE; else return OUTSIDE; /* non-zero winding rule */
```

Of course, the accumulated angle value can no longer be contained within a four-bit number under these conditions, but this characteristic is probably little more than a curiosity anyway, except for its original effect of reducing angle calculations to simple quadrant testing.

♢ Bibliography ♢

(Haines 1994) Eric Haines. Point in Polygon Strategies. In Paul Heckbert, editor, *Graphics Gems IV*, 24–46. Academic Press, Boston, 1994.

(Newman and Sproull 1973) William Newman and Robert Sproull. *Principles of Interactive Computer Graphics*, 1st edition. McGraw-Hill, New York, 1973.

(Rogers 1985) David Rogers. *Procedural Elements for Computer Graphics.* McGraw-Hill, New York, 1985.

◊ I.4

Point in Polygon Strategies

Eric Haines
3D/Eye Inc.
1050 Craft Road
Ithaca, NY 14850
erich@eye.com

Testing whether a point is inside a polygon is a basic operation in computer graphics. This Gem presents a variety of efficient algorithms. No single algorithm is the best in all categories, so the capabilities of the better algorithms are compared and contrasted. The variables examined are the different types of polygons, the amount of memory used, and the preprocessing costs. Code is included in this article for several of the best algorithms; the Gems IV distribution includes code for all the algorithms discussed.

◊ Introduction ◊

The motivation behind this Gem is to provide practical algorithms that are simple to implement and are fast for typical polygons. In applied computer graphics we usually want to check a point against a large number of triangles and quadrilaterals and occasionally test complex polygons. When dealing with floating-point operations on these polygons we do not care if a test point exactly on an edge is classified as being inside or outside, since these cases are normally extremely rare.

In contrast, the field of computational geometry has a strong focus on the order of complexity of an algorithm for all polygons, including pathological cases that are rarely encountered in real applications. The order of complexity for an algorithm in computational geometry may be low, but there is usually a large constant of proportionality or the algorithm itself is difficult to implement. Either of these conditions makes the algorithm unfit for use. Nonetheless, some insights from computational geometry can be applied to the testing of various sorts of polygons and can also shed light on connections among seemingly different algorithms.

Readers that are only interested in the results should skip to the "Conclusions" section.

IBM ISBN 0-12-336155-9
Macintosh ISBN 0-12-336156-7

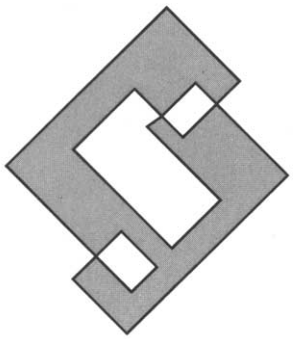


Figure 1. Jordan curve.

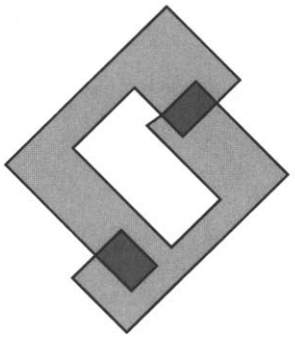


Figure 2. Winding number.

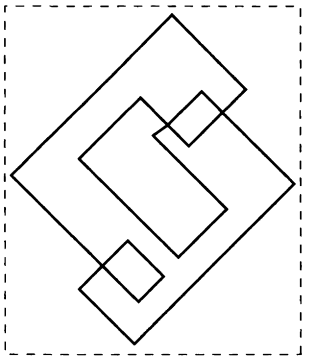


Figure 3. Bounding box.

◊ Definitions ◊

In this Gem a polygon is defined by an ordered set of vertices which form edges making a closed loop. The first and last vertices are connected by an edge, i.e., they are not the same. More complex objects, such as polygons with holes for font lettering, can be built from these polygons by applying the point in polygon test to each loop and concatenating the results.

There are two main types of polygons we will consider in this Gem: general and convex. If a number of points are to be tested against a polygon, it may be worthwhile determining whether the polygon is convex at the start so you are able to use a faster test. General polygons have no restrictions on the placement of vertices. Convex polygon determination is discussed in another Gem in this volume (Schorn and Fisher 1994). If you do not read this other Gem, at least note that a polygon with no concave angles is not necessarily convex; a good counterexample is a star formed by five vertices.

One definition of whether a point is inside a region is the *Jordan Curve Theorem*, also known as the parity or even-odd test. Essentially, it says that a point is inside a polygon if, for any ray from this point, there is an odd number of crossings of the ray with the polygon's edges. This definition means that some areas enclosed by a polygon are not considered inside (Figure 1).

If the entire area enclosed by the polygon is to be considered inside, then the *winding number* is used for testing. This value is the number of times the polygon goes around the point. In Figure 2 the darkly shaded areas have a winding number of two. Think of the polygon as a loop of string pulled tight around a pencil point; the number of loops around the point is the winding number. If a point is outside, the polygon does not wind around it and so the winding number is zero. Winding numbers also have a sign, which corresponds to the direction the edges wrap around the point. The winding

number test can be converted to the parity test; an odd winding number is equivalent to the parity test's inside condition.

In ray tracing and other applications the original polygon is three-dimensional. To simplify computation it is worthwhile to project the polygon and test point into two dimensions. One way to do this is simply to ignore one coordinate. The best coordinate to drop is the one that yields the largest area for the 2D polygon formed. This is easily done by taking the absolute value of each coordinate of the polygon plane's normal and finding the largest; the corresponding coordinates are ignored (Glassner 1989). Precomputing some or all of this information once for a polygon uses more memory but increases the speed of the intersection test itself.

Point in polygon algorithms often benefit from having a bounding box around polygons with many edges. The point is first tested against this box before the full polygon test is performed; if the box is missed, so is the polygon (Figure 3). Most statistics generated in this Gem assume this bounding box test was already passed successfully.

In ray tracing, (Worley and Haines 1993) points out that the polygon's 3D bounding box can be treated like a 2D bounding box by throwing away one coordinate, as done above for polygons. By analysis of the operations involved, it can be shown to be more profitable in general to first intersect the polygon's plane and then test whether the point is inside the 2D bounding box, rather than first testing the 3D bounding box and then the plane. Other bounding box variants can be found in (Woo 1992).

◊ General Algorithms ◊

This section discusses the fastest algorithms for testing points against general polygons. Three classes of algorithms are compared: those which use the vertex list as their only data structure, those which do preprocessing and create an alternate form of the polygon, and those which create additional efficiency structures. The advantages of a vertex list algorithm is that no additional information or preprocessing is needed. However, the other two types of algorithms offer faster testing times in many cases.

Crossings Test

The fastest algorithm without any preprocessing is the crossings test. The earliest presentation of this algorithm is (Shimrat 1962), though it has a bug in it, corrected by (Hacker 1962). A ray is shot from the test point along an axis (+X is commonly used), and the number of crossings is computed for the even-odd test (Figure 4). One way to think about this algorithm is to consider the test point to be at the origin and to check the edges against this point. If the Y coordinates of a polygon edge differ in sign, then the edge can cross the test ray. In this case, if both X coordinates are positive, the edge

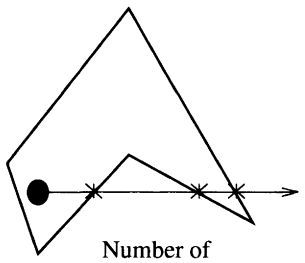


Figure 4. Crossings test.

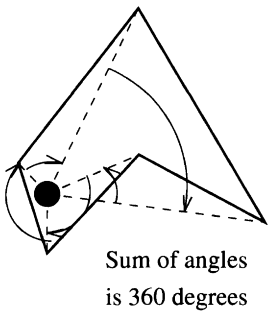


Figure 5. Angle summation.

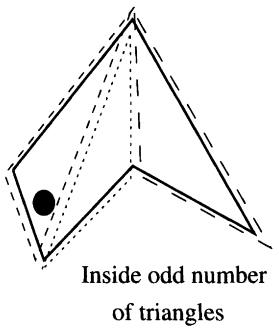


Figure 6. Triangle fan.

and ray must intersect and a crossing is recorded. Else, if the X signs differ, then the X intersection of the edge and the ray is computed and if positive a crossing is recorded.

What happens when the test ray intersects one or more vertices of the polygon? This problem can be ignored by considering the test ray to be a half-plane divider, with one of the half-planes including the ray's points (Preparata and Shamos 1985, Glassner 1989). In other words, whenever the ray would intersect a vertex, the vertex is always classified as being infinitesimally above the ray. In this way, no vertices are considered intersected and the code is both simpler and speedier.

MacMartin pointed out that for polygons with a large number of edges there are generally runs of edges that have Y coordinates with the same sign (Haines 1992). For example, a polygon representing Brazil might have a thousand edges, but only a few of these will straddle a given latitude line and there are long runs of contiguous edges on one side of this line. So a faster strategy is to loop through just the Y coordinates as fast as possible; when they differ then retrieve and check the X coordinates.

Either the even-odd or winding number test can be used to classify the point. The even-odd test is done by simply counting the number of crossings. The winding number test is computed by keeping track of whether the crossed edge passes from the Y- to the Y+ half-plane (add 1) or vice versa (subtract 1). The final value is then the number of counterclockwise windings about the point.

The slowest algorithm for testing points is by far the pure angle summation method. It's simple to describe: sum the signed angles formed at the point by each edge's endpoints (Figure 5). The winding number can then be computed by finding the nearest multiple of 360 degrees. The problem with this pure scheme is that it involves a large number of costly math function calls.

However, the idea of angle summation can be used to formulate a fast algorithm for testing points; see Weiler's Gem in this volume (Weiler 1994). There is a strong

connection between Weiler's algorithm and the crossings test. Weiler avoids expensive trigonometry computations by adding or subtracting one or more increments of 90 degrees as the loop vertices move from quadrant to quadrant (with the test point at the origin). The crossings test is similar in that it can be thought of as counting movements of 360 degrees when an edge crosses the test ray. The crossings test tends to be faster because it does not have to categorize and record all quadrant-to-quadrant movements but only those which cross the test ray. Weiler's formulation is significant for the way it adds to the understanding of underlying principles.

Triangle Fan Tests

In *Graphics Gems*, (Badouel 1990) presents a method of testing points against convex polygons. The polygon is treated as a fan of triangles emanating from one vertex and the point is tested against each triangle by computing its barycentric coordinates. As (Berlin 1985) points out, this test can also be used for non-convex polygons by keeping a count of the number of triangles that overlap the point; if odd, the point is inside the polygon (Figure 6). Unlike the convex test, where an intersection means that the test is done, all the triangles must be tested against the point for the non-convex test. Also, for the non-convex test there may be multiple barycentric coordinates for a given point, since triangles can overlap.

The barycentric test is faster than the crossings test for triangles but becomes quite slow for polygons with more edges. However, (Spackman 1993) notes that pre-normalizing the barycentric equations and storing a set of precomputed values gives better performance. This version of the algorithm is twice as fast as the crossings test for triangles and is in general faster for polygons with few edges. The barycentric coordinates (which are useful for interpolation and texture mapping) are also computed.

A faster triangle fan tester, proposed by (Green 1993), is to store a set of half-plane equations for each triangle and test each in turn. If the point is outside any of the three edges, it is outside the triangle. The half-plane test is an old idea, but storing the half-planes instead of deriving them on the fly from the vertices gives this scheme its speed at the cost of some additional storage space. For triangles this scheme is the fastest of all of the algorithms discussed so far. It is also very simple to code and so lends itself to assembly language translation. Theoretically the Spackman test should usually have a smaller average number of operations per test, but in practice the optimized code for the half-plane test is faster.

Both the half-plane and Spackman triangle testers can be sped up further by sorting the order of the edge tests. Worley and Haines (Spackman 1993) note that the half-plane triangle test is more efficient if the longer edges are tested first. Larger edges tend to cut off more exterior area of the polygon's bounding box and so can result in earlier

exit from testing a given triangle. Sorting in this way makes the test up to 1.7 times faster, rising quickly with the number of edges in the polygon. However, polygons with a large number of edges tend to bog down the sorted edge triangle algorithm, with the crossings test being faster above around 10 edges.

A problem occurs in general triangle fan algorithms when the code assumes that a point that lies on a triangle edge is always inside that triangle. For example, a quadrilateral is treated as two triangles. If a point is exactly on the edge between the two triangles it will be classified as being inside both triangles and so will be classified as being outside the polygon (this problem does not happen with the convex test).

The code presented for these algorithms does not fully address this problem. In reality, a random point tested against a polygon has an infinitesimal chance of landing exactly on any edge. For rendering purposes this problem can be ignored, with the result being one misshaded pixel once in a great while. A more robust solution (which will slow down the test) is to note whether an edge is to include the points exactly on it or not. Also, an option which has not been explored is to test shared interior edges only once against the point and share the results between the adjacent triangles.

Grid Method

An even faster, and more memory intensive, method of testing for points inside a polygon is lookup grids. The idea is to impose a grid inside the bounding box containing the polygon. Each grid cell is categorized as being fully inside, fully outside, or indeterminate. The indeterminate cells also have a list of edges that overlap the cell, and also one corner (or more) is determined to be inside or outside.

To test a point against this structure is extremely quick in most cases. For a reasonable polygon many of the cells are either inside or outside, so testing consists of a simple look-up. If the cell contains edges, then a line segment is formed from the test point to the cell corner and is tested against all edges in the list (Antonio 1992). Since the state of the corner is known, the state of the test point can be found from the number of intersections (Figure 7). Salesin and Stolfi suggest an algorithm similar to this as part of their ray tracing acceleration technique (Salesin and Stolfi 1989).

Care must be taken when a polygon edge exactly (or even nearly exactly) crosses a grid corner, as this corner is then unclassifiable. Rather than coping with the topological and numerical precision problems involved, one simple solution is to just start generating the grid from scratch again, giving slightly different dimensions to the bounding box. Also, when testing the line segment against the edges in a list, exact intersections of an edge endpoint must be counted only once.

One additional enhancement partially solves this problem. Each grid cell has four sides. If no polygon edges cross a side, then that side will be fully inside or outside the polygon. A horizontal or vertical test line segment can then be generated from the test

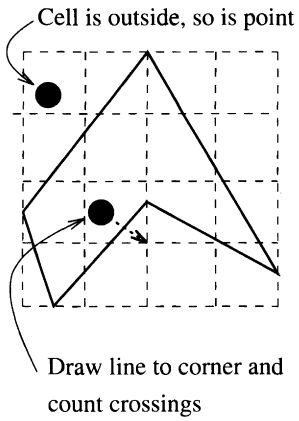


Figure 7. Grid crossings test.

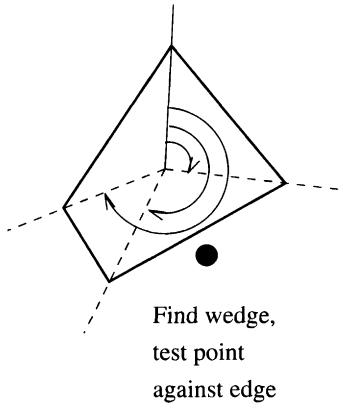


Figure 8. Inclusion test.

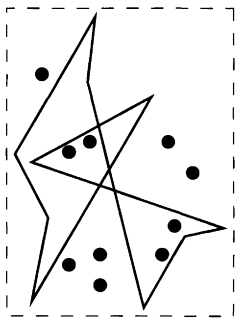


Figure 9. Random polygon.

point to this cell side and the faster crossings test can be used against the edges in the cell. In addition, this crossings test deals with endpoint intersection more robustly.

Note: the grid test code is in the Gems IV code distribution, but has been left out of the book because of its length.

Pixel Based Testing

One interesting case that is related to gridding is that of pixel-limited picking. When a dataset is displayed on the screen and a large amount of picking is to be done on a still image, a specialized test is worthwhile. Hanrahan and Haeberli note that the image can be generated once into a separate buffer, filling in each polygon's area with an identifying index (Hanrahan and Haeberli 1990). When a pixel is picked on this fixed image, it is looked up in this buffer and the polygon selected is known immediately.

◊ Convex Polygons ◊

Convex polygons can be intersected faster due to their geometric properties. For example, the crossings test can quit as soon as two Y-sign difference edges are found, since this is the maximum that a convex polygon can have. Also, note that more polygons can use this faster crossings test by checking only the change in the Y direction (and not X and Y as for the full convexity test); see (Schorn and Fisher 1994). For example, a block letter "E" has at most two Y intersections for any test point's horizontal line (and so is called *monotone* in Y), so it can use the faster crossings test.

The triangle fan tests can exit as soon as any triangle is found to contain the point. These algorithms can be enhanced by both sorting the edges of each triangle by length and also sorting the testing order of triangles by their areas. Relatively larger triangles are more likely to enclose a point and so end testing earlier. Note that this faster test can be applied to any polygon that is decomposed into non-overlapping triangles; convex polygons always have this property when tessellated into a triangle fan.

The exterior algorithm prestores the half-plane for each polygon edge and tests the point against this set. If the point is outside any edge, then the point must be outside the entire convex polygon. This algorithm uses less additional storage than the triangle fan and is very simple to code. The order of edges tested affects the speed of this algorithm; testing edges in the order of which cuts off the most area of the bounding box earliest on is the best ordering. Finding this optimal ordering is non-trivial, but doing the edges in order is often the worst strategy, since each neighboring edge usually cuts off little more area than the previous. Randomizing the order of the edges makes this algorithm up to 10% faster overall for regular polygons.

The exterior algorithm looks for an early exit due to the point being outside the polygon, while the triangle fan convex test looks for one due to the point being inside. For example, for 100 edge polygons, if all points tested are inside the polygon the triangle fan is 1.7 times faster; if all test points are outside the exterior test is more than 16 times faster (but only 4 times faster if the edges are not randomized). So when the polygon/bounding box area ratio is low the exterior algorithm is usually best; in fact, performance is near constant time as this ratio decreases, since after only a few edges most points are categorized as outside the polygon.

A hybrid of the exterior algorithm and the triangle fan is to test triangles and exit early when the point is outside the polygon. A point is outside the polygon if it is outside any exterior triangle edge. This strategy combines the early exit features of both algorithms and so it is less dependent on bounding box fit. Our code uses sorting by triangle area instead of randomizing the exterior edge order, so it favors a higher polygon/bounding box area ratio.

A method with O(log n) performance is the inclusion algorithm (Preparata and Shamos 1985). The polygon is preprocessed by adding a central point to it and is then divided into wedges. The angles from an anchor edge to each wedge's edges are computed and saved, along with half-plane equations for the polygon edges. When a point is tested, the angle from the anchor edge is computed and a binary search is used to determine the wedge it is in, and then the corresponding polygon edge is tested against it (Figure 8). Note that this test can be used on any polygon that can be tessellated into a non-overlapping star of triangles. This algorithm is slower for polygons with few edges because the startup cost is high, but the binary search makes for a much faster test when there are many edges. However, if the bounding box is much larger than the polygon the exterior edge test is faster.

◊ Statistics ◊

The timings given in Tables 1–3 were produced on an HP 720 RISC workstation; timings had similar performance ratios on an IBM PC 386 with no FPU. The general non-convex algorithms were tested using two sorts of polygons: those generated with random points and regular (i.e., equal length sides and vertex angles) polygons with a random rotation applied. Random polygons tend to be somewhat unlikely (no one ever uses 1000-edge random polygons for anything except testing), while regular polygons are more orderly than a "typical" polygon; normal behavior tends to be somewhere in between. Test points were generated inside the bounding box for the polygon. Figure 9 shows a typical 10-sided random polygon and some test points. Convex algorithms were tested with only regular polygons, and so have a certain bias to them.

Test points were generated inside the box bounding the polygon; looser fitting boxes yield different results. Timings are in microseconds per polygon. They are given to two significant figures, since their accuracy is roughly $\pm 10\%$. However, the best way to get useful timings is to run the code on the target machine; there is a testbed program provided in the Gems IV code distribution which can be used to try new algorithms and generate timings under various test conditions. Also, of course, hacking the code for a particular machine and compiler can make a significant difference.

◊ Discussion ◊

The crossings test is generally useful, but we can do better. Testing triangles using either sorted triangle fan algorithm is more than twice as fast, though for polygons with many edges the crossings test is still faster.

Given enough resolution (and enough memory!), gridding gives near constant time performance for most normal polygons, though it performs a bit slower when entirely random polygons are tested. Interestingly, even for polygons with just a few edges the gridding algorithm outperforms most of the other tests.

Testing times can be noticeably decreased by using an algorithm optimized for convex testing when possible. For example, the convex sorted half-plane test is 1.4 times faster for 10-sided polygons than its general case counterpart. For convex polygons with many edges the inclusion test is extremely efficient because of its O(log n) behavior.

Other algorithms remain to be discovered and tested; for example, a practical general polygon algorithm with better than O(n) performance and low storage costs would fill a useful niche.

Table 1. General Algorithms, Random Polygons

Algorithm	Number of edges per polygon						
	3	4	10	20	50	100	1000
Crossings	2.8	3.1	5.7	10	25	48	470
Half Plane w/edge sort	1.1	1.7	5.7	12	32	65	650
Half Plane, no sort	1.2	2.0	6.3	14	36	72	740
Spackman w/edge sort	1.3	2.1	6.0	13	32	66	670
Spackman, no sort	1.4	2.2	6.4	14	35	70	720
Barycentric	2.4	4.0	13	29	76	150	1600
Weiler angle	3.7	4.3	8.7	16	39	77	760
Trigonometric angle	42	51	110	210	520	1030	10300
Grid (100x100)	1.8	1.9	1.9	1.9	2.2	2.5	9.2
Grid (20x20)	2.0	2.0	2.2	2.5	3.6	5.5	38

Table 2. General Algorithms, Regular Polygons

Algorithm	Number of edges per polygon						
	3	4	10	20	50	100	1000
Crossings	2.6	2.7	4.3	7.2	16	32	300
Half Plane w/edge sort	1.3	1.8	4.6	9.2	23	45	460
Half Plane, no sort	1.3	2.1	6.7	14	37	74	760
Spackman w/edge sort	1.5	2.1	5.4	10	26	51	510
Spackman, no sort	1.5	2.3	5.8	11	28	55	550
Barycentric	2.5	4.2	13	26	68	140	1400
Weiler angle	3.5	4.0	7.9	15	35	70	690
Trigonometric angle	39	51	120	230	560	1200	11100
Grid (100x100)	1.8	1.8	1.8	1.8	1.8	1.8	1.9
Grid (20x20)	2.0	2.0	2.0	2.0	2.0	2.1	2.8

Table 3. Convex Algorithms, Regular Polygons

Algorithm	Number of edges per polygon						
	3	4	10	20	50	100	1000
Inclusion	5.5	5.7	6.3	6.6	7.1	7.6	9.9
Hybrid Sorted Half Plane	1.3	1.6	3.3	6.1	14	28	280
Sorted Half Plane	1.2	1.6	3.4	6.2	15	29	280
Unsorted Half Plane	1.2	1.9	5.7	12	30	61	620
Random Exterior Edges	1.3	1.7	3.8	7.1	17	33	320
Ordered Exterior Edges	1.3	1.7	3.8	7.3	18	35	350
Convex Crossings	2.5	2.5	3.6	5.6	12	22	220

◊ Conclusions ◊

- If no preprocessing nor extra storage is available, use the **Crossings** test.
- If a little preprocessing and extra storage is available:
  - For general polygons
    - with few sides, use the **Half-Plane** or **Spackman** test.
    - with many sides, use the **Crossings** test.
  - For convex polygons
    - with few sides, use the **Hybrid Half-Plane** test.
    - with many sides, use the **Inclusion** test.
    - But if the bounding box/polygon area ratio is high, use the **Exterior Edges** test.
- If preprocessing and extra storage is available in abundance, use the **Grid Test** (except for perhaps triangles).

Of course, some of these conclusions may vary with machine architecture and compiler optimization.

◊ C Code ◊

ptinpoly.h

```
/* ptinpoly.h - point in polygon inside/outside algorithms header file.
 */

/* Define CONVEX to compile for testing only convex polygons (when possible,
 * this is faster). */
/* #define CONVEX */

/* Define HYBRID to compile triangle fan test for CONVEX with exterior edges
 * meaning an early exit (faster - recommended).
 */
/* #define HYBRID */

/* Define DISPLAY to display test triangle and test points on screen. */
/* #define DISPLAY */

/* Define RANDOM to randomize order of edges for exterior test (faster -
 * recommended). */
/* #define RANDOM */

/* Define SORT to sort triangle edges and areas for half-plane and Spackman
```

```
 * tests (faster - recommended). */
/* #define SORT */

/* Define WINDING if a non-zero winding number should be used as the criterion
 * for being inside the polygon.  Only used by the general crossings test and
 * Weiler test.  The winding number computed for each is the number of
 * counterclockwise loops the polygon makes around the point.
 */
/* #define WINDING */

/* ============================ System Related ============================== */

/* Define your own random number generator; change as needed. */
/* SRAN initializes random number generator, if needed. */
#define SRAN()          srand48(1)
/* RAN01 returns a double from [0..1) */
#define RAN01()         drand48()
double  drand48() ;

/* =========== Half-Plane stuff ============================================= */

typedef struct {
    double      vx, vy, c ;     /* edge equation  vx*X + vy*Y + c = 0 */
#ifdef CONVEX
#ifdef HYBRID
    int         ext_flag ;      /* TRUE == exterior edge of polygon */
#endif
#endif
} PlaneSet, *pPlaneSet ;


#ifdef  CONVEX
#ifdef  SORT
/* Size sorting structure for half-planes */
typedef struct {
    double      size ;
    pPlaneSet   pps ;
} SizePlanePair, *pSizePlanePair ;
#endif
#endif


#ifdef  CONVEX
pPlaneSet       ExteriorSetup() ;
void    ExteriorCleanup() ;
#ifdef  SORT
int     CompareSizePlanePairs() ;
#endif
#endif
pPlaneSet       PlaneSetup() ;
void    PlaneCleanup() ;
```

ptinpoly.c

```
/* ptinpoly.c - point in polygon inside/outside code.

   by Eric Haines, 3D/Eye Inc, erich@eye.com

   This code contains the following algorithms:
        crossings - count the crossing made by a ray from the test point
        half-plane testing - test triangle fan using half-space planes
        exterior test - for convex polygons, check exterior of polygon
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "ptinpoly.h"

#define X       0
#define Y       1
#define TRUE    1

#ifndef HUGE
#define HUGE    1.79769313486232e+308
#endif

#define MALLOC_CHECK( a )       if ( !(a) ) {                               \
                                    fprintf( stderr, "out of memory\n" ) ; \
                                    exit(1) ;                               \
                                }

/* ======= Crossings algorithm ============================================ */

/* Shoot a test ray along +X axis.  The strategy, from MacMartin, is to
 * compare vertex Y values to the testing point's Y and quickly discard
 * edges which are entirely to one side of the test ray.
 *
 * Input 2D polygon _pgon_ with _numverts_ number of vertices and test point
 * _point_, returns 1 if inside, 0 if outside.  WINDING and CONVEX can be
 * defined for this test.
 */
int CrossingsTest( pgon, numverts, point )
double  pgon[][2] ;
int     numverts ;
double  point[2] ;
{
#ifdef  WINDING
register int    crossings ;
#endif
register int    j, yflag0, yflag1, inside_flag, xflag0 ;
register double ty, tx, *vtx0, *vtx1 ;
#ifdef  CONVEX
```

```
register int    line_flag ;
#endif

    tx = point[X] ;
    ty = point[Y] ;

    vtx0 = pgon[numverts-1] ;
    /* get test bit for above/below X axis */
    yflag0 = ( vtx0[Y] >= ty ) ;
    vtx1 = pgon[0] ;

#ifdef  WINDING
    crossings = 0 ;
#else
    inside_flag = 0 ;
#endif
#ifdef  CONVEX
    line_flag = 0 ;
#endif
    for ( j = numverts+1 ; --j ; ) {

        yflag1 = ( vtx1[Y] >= ty ) ;
        /* check if endpoints straddle (are on opposite sides) of X axis
         * (i.e., the Y's differ); if so, +X ray could intersect this edge.
         */
        if ( yflag0 != yflag1 ) {
            xflag0 = ( vtx0[X] >= tx ) ;
            /* check if endpoints are on same side of the Y axis (i.e., X's
             * are the same); if so, it's easy to test if edge hits or misses.
             */
            if ( xflag0 == ( vtx1[X] >= tx ) ) {

                /* if edge's X values both right of the point, must hit */
#ifdef  WINDING
                if ( xflag0 ) crossings += ( yflag0 ? -1 : 1 ) ;
#else
                if ( xflag0 ) inside_flag = !inside_flag ;
#endif
            } else {
                /* compute intersection of pgon segment with +X ray, note
                 * if >= point's X; if so, the ray hits it.
                 */
                if ( (vtx1[X] - (vtx1[Y]-ty)*
                     ( vtx0[X]-vtx1[X])/(vtx0[Y]-vtx1[Y])) >= tx ) {
#ifdef  WINDING
                    crossings += ( yflag0 ? -1 : 1 ) ;
#else
                    inside_flag = !inside_flag ;
#endif
                }
            }
#ifdef  CONVEX
            /* if this is second edge hit, then done testing */
```

```
            if ( line_flag ) goto Exit ;

            /* Note that one edge has been hit by the ray's line. */
            line_flag = TRUE ;
#endif
        }

        /* Move to next pair of vertices, retaining info as possible. */
        yflag0 = yflag1 ;
        vtx0 = vtx1 ;
        vtx1 += 2 ;
    }
#ifdef  CONVEX
    Exit: ;
#endif
#ifdef  WINDING
    /* Test if crossings is not zero. */
    inside_flag = (crossings != 0) ;
#endif

    return( inside_flag ) ;
}


/* ======= Triangle half-plane algorithm ================================== */

/* Split the polygon into a fan of triangles and for each triangle test if
 * the point is inside of the three half-planes formed by the triangle's edges.
 *
 * Call setup with 2D polygon _pgon_ with _numverts_ number of vertices,
 * which returns a pointer to a plane set array.
 * Call testing procedure with a pointer to this array, _numverts_, and
 * test point _point_, returns 1 if inside, 0 if outside.
 * Call cleanup with pointer to plane set array to free space.
 *
 * SORT and CONVEX can be defined for this test.
 */

/* Split polygons along set of x axes - call preprocess once. */
pPlaneSet       PlaneSetup( pgon, numverts )
double  pgon[][2] ;
int     numverts ;
{
int     i, p1, p2 ;
double  tx, ty, vx0, vy0 ;
pPlaneSet       pps, pps_return ;
#ifdef  SORT
double  len[3], len_temp ;
int     j ;
PlaneSet        ps_temp ;
#ifdef  CONVEX
pPlaneSet       pps_new ;
pSizePlanePair p_size_pair ;
```

```
#endif
#endif

    pps = pps_return =
            (pPlaneSet)malloc( 3 * (numverts-2) * sizeof( PlaneSet )) ;
    MALLOC_CHECK( pps ) ;
#ifdef  CONVEX
#ifdef  SORT
    p_size_pair =
        (pSizePlanePair)malloc( (numverts-2) * sizeof( SizePlanePair ) ) ;
    MALLOC_CHECK( p_size_pair ) ;
#endif
#endif

    vx0 = pgon[0][X] ;
    vy0 = pgon[0][Y] ;

    for ( p1 = 1, p2 = 2 ; p2 < numverts ; p1++, p2++ ) {
        pps->vx = vy0 - pgon[p1][Y] ;
        pps->vy = pgon[p1][X] - vx0 ;
        pps->c = pps->vx * vx0 + pps->vy * vy0 ;
#ifdef  SORT
        len[0] = pps->vx * pps->vx + pps->vy * pps->vy ;
#ifdef  CONVEX
#ifdef  HYBRID
        pps->ext_flag = ( p1 == 1 ) ;
#endif
        /* Sort triangles by areas, so compute (twice) the area here. */
        p_size_pair[p1-1].pps = pps ;
        p_size_pair[p1-1].size =
                        ( pgon[0][X] * pgon[p1][Y] ) +
                        ( pgon[p1][X] * pgon[p2][Y] ) +
                        ( pgon[p2][X] * pgon[0][Y] ) -
                        ( pgon[p1][X] * pgon[0][Y] ) -
                        ( pgon[p2][X] * pgon[p1][Y] ) -
                        ( pgon[0][X] * pgon[p2][Y] ) ;
#endif
#endif
        pps++ ;
        pps->vx = pgon[p1][Y] - pgon[p2][Y] ;
        pps->vy = pgon[p2][X] - pgon[p1][X] ;
        pps->c = pps->vx * pgon[p1][X] + pps->vy * pgon[p1][Y] ;
#ifdef  SORT
        len[1] = pps->vx * pps->vx + pps->vy * pps->vy ;
#endif
#ifdef  CONVEX
#ifdef  HYBRID
        pps->ext_flag = TRUE ;
#endif
#endif
        pps++ ;
        pps->vx = pgon[p2][Y] - vy0 ;
        pps->vy = vx0 - pgon[p2][X] ;
```

```
        pps->c = pps->vx * pgon[p2][X] + pps->vy * pgon[p2][Y] ;
#ifdef  SORT
        len[2] = pps->vx * pps->vx + pps->vy * pps->vy ;
#endif
#ifdef  CONVEX
#ifdef  HYBRID
        pps->ext_flag = ( p2 == numverts-1 ) ;
#endif
#endif

        /* Find an average point that must be inside of the triangle. */
        tx = ( vx0 + pgon[p1][X] + pgon[p2][X] ) / 3.0 ;
        ty = ( vy0 + pgon[p1][Y] + pgon[p2][Y] ) / 3.0 ;

        /* Check sense and reverse if test point is not thought to be inside
         * first triangle.
         */
        if ( pps->vx * tx + pps->vy * ty >= pps->c ) {
            /* back up to start of plane set */
            pps -= 2 ;
            /* Point is thought to be outside, so reverse sense of edge
             * normals so that it is correctly considered inside.
             */
            for ( i = 0 ; i < 3 ; i++ ) {
                pps->vx = -pps->vx ;
                pps->vy = -pps->vy ;
                pps->c  = -pps->c ;
                pps++ ;
            }
        } else {
            pps++ ;
        }

#ifdef  SORT
        /* Sort the planes based on the edge lengths. */
        pps -= 3 ;
        for ( i = 0 ; i < 2 ; i++ ) {
            for ( j = i+1 ; j < 3 ; j++ ) {
                if ( len[i] < len[j] ) {
                    ps_temp = pps[i] ;
                    pps[i] = pps[j] ;
                    pps[j] = ps_temp ;
                    len_temp = len[i] ;
                    len[i] = len[j] ;
                    len[j] = len_temp ;
                }
            }
        }
        pps += 3 ;
#endif
    }

#ifdef  CONVEX
```

```
#ifdef  SORT
    /* Sort the triangles based on their areas. */
    qsort( p_size_pair, numverts-2,
            sizeof( SizePlanePair ), CompareSizePlanePairs ) ;

    /* Make the plane sets match the sorted order. */
    for ( i = 0, pps = pps_return
        ; i < numverts-2
        ; i++ ) {

        pps_new = p_size_pair[i].pps ;
        for ( j = 0 ; j < 3 ; j++, pps++, pps_new++ ) {
            ps_temp = *pps ;
            *pps = *pps_new ;
            *pps_new = ps_temp ;
        }
    }
    free( p_size_pair ) ;
#endif
#endif

    return( pps_return ) ;
}

#ifdef  CONVEX
#ifdef  SORT
int CompareSizePlanePairs( p_sp0, p_sp1 )
pSizePlanePair  p_sp0, p_sp1 ;
{
    if ( p_sp0->size == p_sp1->size ) {
        return( 0 ) ;
    } else {
        return( p_sp0->size > p_sp1->size ? -1 : 1 ) ;
    }
}
#endif
#endif


/* Check point for inside of three "planes" formed by triangle edges. */
int PlaneTest( p_plane_set, numverts, point )
pPlaneSet       p_plane_set ;
int     numverts ;
double  point[2] ;
{
register pPlaneSet      ps ;
register int    p2 ;
#ifndef CONVEX
register int    inside_flag ;
#endif
register double tx, ty ;

    tx = point[X] ;
```

```
    ty = point[Y] ;

#ifndef CONVEX
    inside_flag = 0 ;
#endif

    for ( ps = p_plane_set, p2 = numverts-1 ; --p2 ; ) {

        if ( ps->vx * tx + ps->vy * ty < ps->c ) {
            ps++ ;
            if ( ps->vx * tx + ps->vy * ty < ps->c ) {
                ps++ ;
                /* Note: we make the third edge have a slightly different
                 * equality condition, since this third edge is in fact
                 * the next triangle's first edge.  Not fool-proof, but
                 * it doesn't hurt (better would be to keep track of the
                 * triangle's area sign so we would know which kind of
                 * triangle this is).  Note that edge sorting nullifies
                 * this special inequality, too.
                 */
                if ( ps->vx * tx + ps->vy * ty <= ps->c ) {
                    /* point is inside polygon */
#ifdef CONVEX
                    return( 1 ) ;
#else
                    inside_flag = !inside_flag ;
#endif
                }
#ifdef  CONVEX
#ifdef  HYBRID
                /* check if outside exterior edge */
                else if ( ps->ext_flag ) return( 0 ) ;
#endif
#endif
                ps++ ;
            } else {
#ifdef  CONVEX
#ifdef  HYBRID
                /* check if outside exterior edge */
                if ( ps->ext_flag ) return( 0 ) ;
#endif
#endif
                /* get past last two plane tests */
                ps += 2 ;
            }
        } else {
#ifdef  CONVEX
#ifdef  HYBRID
            /* check if outside exterior edge */
            if ( ps->ext_flag ) return( 0 ) ;
#endif
#endif
            /* get past all three plane tests */
```