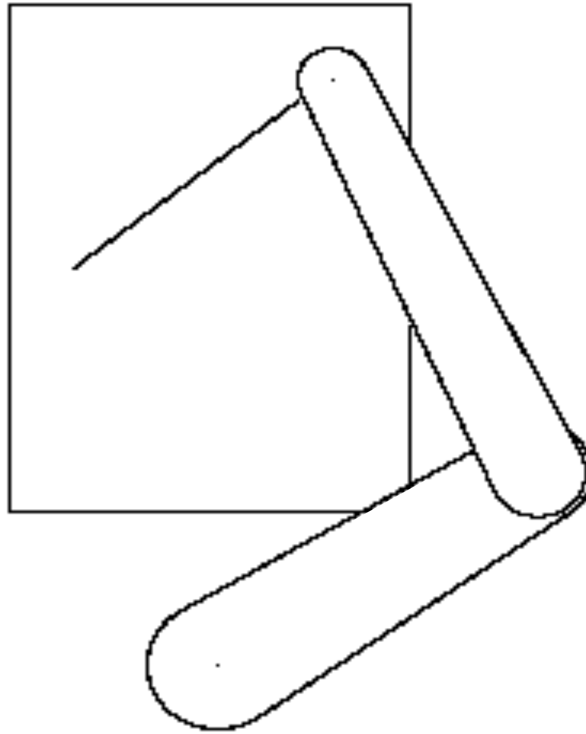


# Flexing the Power of Algorithmic Geometry

An introduction to modern software geometry theory and problem solving

Pierre Bierre

**Flexing the Power of Algorithmic Geometry**
By Pierre Bierre

Copyright 2009 Spatial Thoughtware. All rights reserved.
Printed in the United States of America.
Published by Spatial Thoughtware, 980 Riesling Dr.,
Pleasanton CA 94566 USA

Educational method Patent Pending per U.S. Patent Application 61/062,660.

The method in Part II Chapter 14 **Rotational Inference: Inferring a Rotator from its Effect** is claimed in U.S. Patent Application 60/519,411.

Java™ and all Java trademarks and logos are trademarks of Oracle, Inc.

Eclipse™ is a trademark of the Eclipse Foundation, Inc.

While every precaution has been taken in the preparation of this book, the author and publisher assume no responsibility for errors and omissions, or for damages resulting from use of the information contained herein.

**ISBN: 978-0-9827526-0-9** **08/2010**

**Why Algorithmic Geometry?**

The geometry I learned as a high school student was the result of 2.5 millennia of learning and problem solving carried out with writing systems… paper and pencil. Since the 1980s, computers and software have made geometry come alive in a way that would seem magical to the ancients. The impact can be seen all around us in computer graphics, robotics, computer vision, CAD, GPS…pursuits still emerging from infancy.

The effect of computing on geometry has been epochal, essentially redefining a new *partnership* for solving problems. The human is still on top as the creative thinker and actor, but the new assistant brings to the table seemingly inexhaustible stamina for crunching numbers, throwing colored dots on a screen, or issuing motor-move commands. This unique partnership of grey matter and silicon makes possible the complex machine behavior put forth in today's embedded-geometry products. Paper and pencil remain essential tools on the creative end of this partnership.

The question needs to be asked: How relevant are traditional, pre-computational geometry concepts when doing geometry in software? Two generations of experience yield an expectably nuanced answer. Points and distances are still paramount. We rely heavily on Pythagoras but use almost no trigonometry. For rotation, we pass over angles in favor of computationally-elegant *direction vectors*. By the end of the chapter on 2D extended lines, you will have gained an appreciation for why slope (dy/dx) is poorly suited for representing line tilt, and how *orientation* does the job. These modern representational concepts work the same way in 2D and 3D, and offer ease-of-use in thinking and writing algorithms.

What is totally new and powerful in software geometry is the ability to *automate* problem solving. That is, once we forge a solution to a problem (e.g., the intersection of two lines in 2D), we permanently create a *servant* who can be called on to obediently carry out this work whenever needed, in whatever context. We further enjoy the luxury of being allowed to forget *how* we solved line intersection, while using it to solve more complicated problems, such as the intersection of two 3D planes. These dutiful servants are *algorithms*.

Being able to write algorithms and then take them for granted, we end up constructing layer upon layer of increasingly more powerful problem-solving agents. If we do things right, every geometry problem we solve is incarnated as an automated problem-solver, plowable back into a continuing spiral of upwardly-compatible cyberfunctionality.

This ongoing synergy of human problem solving and algorithmic mechanization, with immediate feedback on the screen, gives tremendous power to those practicing algorithmic geometry. In this regard, traditional paper and pencil geometry has undergone its most sweeping transformation, one highly empowering to people of our time. So, have at it!

# Part I 2D Geometry


| | | Page |
|---|---|---|
| Chapter 1. | Representing 2D Points as Vectors | 7 |
| Chapter 2. | Object Motion | 15 |
| Chapter 3. | Circles | 19 |
| Chapter 4. | Representing Direction with Vectors | 21 |
| Chapter 5. | Representing Extended 2D Lines | 29 |
| Chapter 6. | Coordinate Rotation | 43 |
| Chapter 7. | Adding and Subtracting Angles in DirVec2 Space | 49 |
| Chapter 8. | Intersection of 2 Lines | 52 |
| Chapter 9. | Intersection of Line and Circle | 60 |
| Chapter 10. | Intersection of 2 Circles | 64 |
| Chapter 11. | Triangulation Methods | 70 |
| Chapter 12. | (optional) Line Segments and Circular Arcs | 75 |


## Exploring 3D Challenges:


| | |
|---|---|
| Socket Wrench Pick and Place Robot | 80 |
| Shoulder-Elbow Robot Arm | 83 |
| Analog Clockface Parallel Graphics | 86 |
| Location by Triangulation | 89 |

# Part II 3D Geometry


| | | Page |
|---|---|---|
| Chapter 1. | Axes and Points | 96 |
| Chapter 2. | Direction Vectors | 99 |
| Chapter 3. | Planes | 103 |
| Chapter 4. | Rotators | 105 |
| Chapter 5. | Lines | 109 |
| Chapter 6. | Intersection of 2 Planes ➔ Line | 114 |
| Chapter 7. | Intersection of Line and Plane ➔ Point | 119 |
| Chapter 8. | Intersection of 3 Planes ➔ Point | 122 |
| Chapter 9. | Spheres and Circles | 123 |
| Chapter 10. | Intersection of 2 Spheres ➔ Circle | 124 |
| Chapter 11. | Intersection of Circle and Sphere ➔ 2 Points | 128 |
| Chapter 12. | GPS Triangulation: Intersection of 3 Spheres ➔ 2 Points | 130 |
| Chapter 13. | Directional Triangulation | 134 |
| Chapter 14. | Rotational Inference: Inferring a Rotator from its Effect | 136 |


## Exploring 3D Challenges:


| | |
|---|---|
| 3D Wireframe Graphics | 140 |
| Automated Gas Pump Attendant | 143 |
| CAD Pipe Outline Graphics | 146 |
| Optical Reflection and Refraction | 147 |
| Computer Vision | 151 |
| Molecular Brownian Rotation | 154 |
| Deep Space Navigation | 155 |

**Page**

**Summary Perspective on Algorithmic Geometry** **157**

**Appendix A. Getting Started with Eclipse Java and AlgoGeom2D** **161**

**Appendix B. Answers to Paper and Pencil Problems** **162**

**About the Author** **171**

## Acknowledgments

The inspiration to modernize 9-12 math education arose from reading the Hart-Rudman Commission's recommendations circa 2000. To paraphrase, math and science education have more to offer future national security than anything the military can provide.

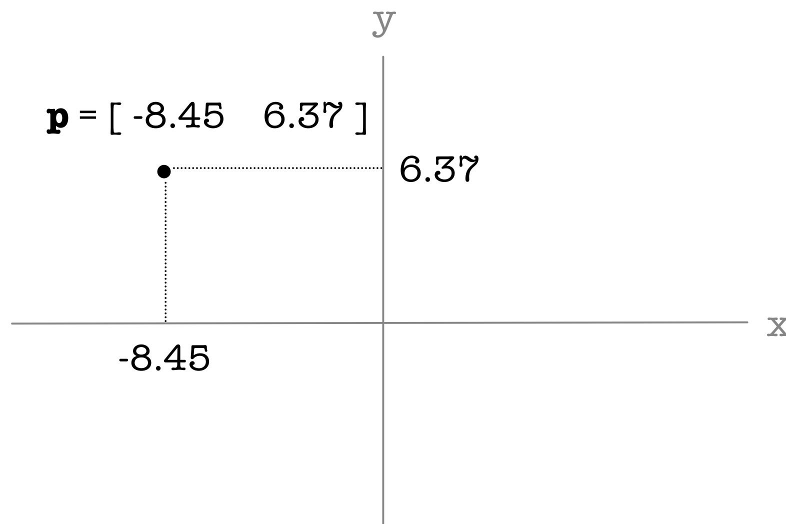
I have been blessed with many great teachers along the way. Thomas Judd and Bill Rankin brought the fruits of Sputnik-era science curriculum reforms to Greece Arcadia HS just on time for my cohort. PSSC Physics helped me decide to go on to Stony Brook for Theoretical Physics. In college, David Emmerich and Herb Bliden encouraged my interest in psychophysics. At CS grad school in UC Boulder, I received first class mentoring from Paul Ziegler, and learned numerical computing from one of its best, Bobby Schnabel.

Working at Stanford Neuropsychology Lab in the '80s, Karl Pribram pushed me to to meld ideas in math, computing and brain science while studying mammalian vision systems.

AlgoGeom.org owes much to Brian Lukoff, who designed a teaching portal and on-line assessment suite, and shaped the Java language conventions used. Beth Injasoulian pioneered the way for professional development as the first credentialed math teacher to learn AlgoGeom, and Gregory Duran and Stan Hitomi paved the way for our first public school delivery in 2010 at Dougherty Valley High in San Ramon CA. Stan worked with Dick Farnsworth and Jim Bono at Lawrence Livermore Labs to secure initial funding.

## Chapter 1. Representing 2D Points as Vectors

Descartes' system of coordinate geometry serves as the foundation and entranceway for algorithmic geometry. The point **p** is known by its [ x y ] coordinate location, where x and y are real numbers.




The first step toward automated data handling we make is to treat [ x y ] as one object, i.e. we glue the two numeric values together into a single object called a *vector*. Using vectors roughly halves the mental work of solving problems involving points. Our lowest-level algorithms will explicitly deal with x and y, but after these algorithms are in place, we begin to process points and locations using operations that work directly on points and locations…we will process *vectors*. Vector operations lead to powerful problem-solving methods.

So commonplace will it become to think in terms of vectors, the exception will be to have to handle single, real number variables, which in the context of vector math are called *scalars*. The x and y values inside the vector are scalars, as are distances between points.

Computer Graphics: Working in pure Cartesian coordinates

Interactive graphics is an ideal workspace for learning 2D algorithmic geometry. The pixels on the screen are individually addressable, and therefore provide a Cartesian playground where [ x y ] coordinates take on a geometric meaning. The computer's mouse input position is sensed using the same pixel coordinates.

Java display windows use *integer-based* pixel coordinates that are *upside-down*. We ignore this *faux pas*, and superimpose a *right-side-up*, *real number* Cartesian coordinate system onto the pixel space, where: