Statistics and Computing

Series Editors:
J. Chambers
D. Hand
W. Härdle

Leland Wilkinson

The Grammar of Graphics

Second Edition

With 410 Illustrations, 319 in Full Color

With contributions by Graham Wills, Dan Rope, Andrew Norton, and Roger Dubbs

Springer

Leland Wilkinson
SPSS Inc.
233 S. Wacker Drive
Chicago, IL 60606-6307
USA
leland@spss.com

Library of Congress Control Number:

ISBN-10: 0-387-24544-8 Printed on acid-free paper.
ISBN-13: 978-0387-24544-7

Printed in Canada.

9 8 7 6 5 4 3 2 1

springeronline.com

To John Hartigan and Amie Wilkinson

Who can hide in secret places so that I cannot see them? Do I not fill heaven and earth?
Jeremiah 23.24

Cleave a piece of wood, I am there; lift up the stone and you will find me there.
Gospel of Thomas 77

God hides in the smallest pieces.
Caspar Barlaeus

God hides in the details.
Aby Warburg

God is in the details.
Ludwig Mies van der Rohe

The devil is in the details.
George Shultz

Bad programmers ignore details. Bad designers get lost in details.
Nate Kirby

Preface

Preface to First Edition

Before writing the graphics for SYSTAT in the 1980's, I began by teaching a seminar in statistical graphics and collecting as many different quantitative graphics as I could find. I was determined to produce a package that could draw every statistical graphic I had ever seen. The structure of the program was a collection of procedures named after the basic graph types they produced. The graphics code was roughly one and a half megabytes in size.

In the early 1990's, I redesigned the SYSTAT graphics package using object-based technology. I intended to produce a more comprehensive and dynamic package. I accomplished this by embedding graphical elements in a tree structure. Rendering graphics was done by walking the tree and editing worked by adding and deleting nodes. The code size fell to under a megabyte.

In the late 1990's, I collaborated with Dan Rope at the Bureau of Labor Statistics and Dan Carr at George Mason University to produce a graphics production library called GPL, this time in Java. Our goal was to develop graphics components. This book was nourished by that project. So far, the GPL code size is under half a megabyte.

I have not yet achieved that elusive zero-byte graphics program, but I do believe that bulk, in programming or in writing, can sometimes be an inverse measure of clarity of thought. Users dislike "bloatware" not only because it is a pig that wastes their computers' resources but also because they know it usually reflects design-by-committee and sloppy thinking.

Notwithstanding my aversion to bulk, this book is longer than I had anticipated. My original intent was to outline a new paradigm for quantitative graphics using examples from publications and from SYSTAT. As the GPL project proceeded and we were able to test concepts in a working program, I began to realize that the details of the system were as important as the outlines. I also found that it was easier to write about the generalities of graphics than about the particulars. As every programmer knows, it is easier to wave one's hands than to put them to the keyboard. And as every programmer knows in the middle of the night, the computer "wonderfully focuses the mind."

The consequence is a book that is not, as some like to say, "an easy read." I do not apologize for this. Statistical graphics is not an easy field. With rare exceptions, theorists have not taken graphics seriously or examined the field deeply. And I am convinced that those who have, like Jacques Bertin, are not often read carefully. It has taken me ten years of programming graphics to understand and appreciate the details in Bertin.

I am not referring to the abstruseness of the mathematics in scientific and technical charts when I say this is not an easy field. It is easier to graph New-

ton's law of gravitation than to draw a pie chart. And I do not mean that no one has explored aspects of graphics in depth or covered the whole field with illumination. I mean simply that few have viewed quantitative graphics as an area that has peculiar rules and deep grammatical structure. As a result, we have come to expect we can understand graphics by looking at pictures and speaking in generalities. Against that expectation, I designed this book to be read more than once. On second reading, you will discover the significance of the details and that will help you understand the necessity of the framework.

Who should read this book? The simple answer is, of course, anyone who is interested in business or scientific graphics. At the most elementary level are readers who are looking for a graphical catalog or thesaurus. There are not many types of graphics that do not appear somewhere in this book. At the next level are those who want to follow the arguments without the mathematics. One can skip all the mathematics and still learn what the fundamental components of quantitative graphics are and how they interact. At the next level are those who have completed enough college mathematics to follow the notation. I have tried to build the argument, except for the statistical methods in Chapter 7, from elementary definitions. I chose a level comparable to an introductory computer science or discrete math text, and a notation that documents the algorithms in set terminology computer science students will recognize.

I intend to reach several groups. First are college and graduate students in computer science and statistics. This is the only book in print that lays out in detail how to write computer programs for business or scientific graphics. For all the attention computer graphics courses devote to theory, modeling, animation, and realism, the vast majority of commercial applications involve quantitative graphics. As a software developer, I believe the largest business market for graphics will continue to be analysis and reporting, despite the enthusiastic predictions (driven by conventional wisdom) for data mining, visualization, animation, and virtual reality. The reason, I think, is simple. People in business and science have more trouble communicating than discovering.

The second target group for this book comprises mathematicians, statisticians, and computer scientists who are not experts in quantitative graphics. I hope to be able to convey to them the richness of this field and to encourage them to explore it beyond what I have been able to do. Among his many accomplishments in the fields of graphics and statistics, William Cleveland is largely responsible for stimulating psychologists (including me) to take a closer look at graphical perception and cognition. I hope this book will stimulate experts in other fields to examine the language of graphics.

The third target group consists of statistics and computer science specialists in graphics. These are the colleagues most likely to recognize that this book is more the assembly of a large puzzle than the weaving of a whole cloth. I cannot assume every expert will understand this book, however, for reasons similar to why expertise in procedural programming can hinder one from learning object-oriented design. Those who skim through or jump into the middle of this book are most likely to misunderstand. There are many terms in

this book — *graph*, *graphic*, *variable*, *frame*, *point*, *line*, *shape*, *scale* — that have unfortunately come to be overloaded in ordinary and technical discourse. I have taken care to define these terms when they first appear and then to refine the definitions in context later in the book. Preconceptions about these terms, however technical, can prevent one from following my argument. And those who have heard me talk about graphics algebra in meetings or colloquia need to keep in mind that algebra is only one of fifteen chapters in this book. Before drawing any conclusions, one should read the book from start to finish and attend to the details.

The popular saying "God is in the details," whose lineage I have traced on the frontispiece, has an ancient heritage. It is usually attributed in English to the architect Ludwig Mies van der Rohe, who probably was quoting the art historian Aby Warburg. Elizabeth Sears, a Warburg scholar, told me that Warburg's saying is "much quoted, its sources variously sought." She cited Kany (1985), who devoted an entire article to the topic. William Heckscher (1958) found a possible source in the 17th century humanist Caspar Barlaeus (see correspondents' notes in Safire, 1997). The idea has much older roots in Western culture, however. It is a corollary of an immanent creator — the opposite of an absconding God. Church fathers and rabbis discussed God's omnipresence along these lines in the first millennium, and I have cited a verse from Jeremiah that gives evidence of biblical roots. In our time, we have altered its meaning by focusing on our attending to details rather than on God being in them. I do not know if George Shultz is the first to have given the saying an ironic twist. He used the expression "the devil is in the details" when referring to the intricacies of the SALT talks in a speech to the Council on Foreign Relations. In retrospect, however, Shultz informed me that he may have been quoting some earlier wit. My favorite recent redaction is by a programmer at SPSS. Nate Kirby's observation that bad programmers ignore details and bad designers get lost in them captures for me the difficulty in creating a complex system.

This book was composed in Times® Roman and Italic with Adobe FrameMaker®. The quantitative graphics were produced with SYSTAT®. Rick Wessler drew Figure 9.57 with Caligari TrueSpace®. Figure 20.1 (also on the cover) was created originally in GPL. The remaining non-statistical diagrams and tables were drawn with tools in FrameMaker and Adobe Photoshop®.

I have many to thank. Dan Rope is one of the few individuals I have met who is both master designer and master coder. He gave me the rare opportunity to test almost every idea in this book in detail and contributed many of his own. The few untested ideas will probably be discovered some years from now when others program them, but they will not be Dan's fault. Dan Carr, my other GPL collaborator, taught me with examples. One graphic from Dan (he has done many) can teach me more about good design than some books.

The group once at Bell Labs and now at its descendent institutions has continued to be a unique source of inspiration. Bill Cleveland has energized and advised me for almost two decades; he wins the citation derby in this book. John Chambers, John Tukey, Paul Tukey, Rick Becker, Deborah

Swayne, Andreas Buja, Allan Wilks, Daryl Pregibon, James Landwehr, Lorraine Denby, and Mark Hansen have listened encouragingly and repeatedly to half-baked ideas.

At Northwestern, Bruce Spencer, Shelby Haberman, Martin Tanner, Keith Burns, and Amie Wilkinson have provided critical advice and support. At the University of Chicago, Stephen Stigler, William Kruskal, Ron Thisted, Benson Farb, and Peter McCullaugh have done the same. My Yale cohorts Jerry Dallal and Bill Eddy have helped me over more years than the work in this book spans.

At SYSTAT, Dan Michaels and Steve Willie introduced me to object-oriented design. Tom Leuthner worked extensively with me on maps. Mark Bjerknes, Greg Mullins, Mike Pechnyo, and Sasha Khalileev helped alert me to graphical issues raised by Mac, DOS, Windows, UNIX, and other environments. Laszlo Engelman shared numerical and graphical tricks that continue to amaze me. Pat Fleury and MaryAnn Hill made sure I continued to think about the graphical needs of real users.

At SPSS, Jack Noonan, Ed Hamburg, Mark Battaglia, and Nancy Dobrozdravic provided me the environment needed to complete this project. Josh Barr, Matt Levine, Andrew Walaszek, Jay Jayaprasad, Matt Rubin, Rajesh Selukar, John Burkey, Scott Adler, Janice Krinsky, Sheri Gilley, ViAnn Beadle, Rick Marcantonio, Joel York, Keith Kroeger, Jing Shyr, Ming-Long Lam, Jim Cortese, Scott Sipiora, Dave Hess, Tex Hull, Kim Peck, John Fry, Bonnie Shapiro, Richard Oliver, and Hiroaki Minato contributed numerous suggestions.

Elsewhere, I have received support and advice from several colleagues. The most significant have been Robert Abelson (my dissertation advisor at Yale), Susanna Epp, and Helmut Epp. Jack Cohen and Louis Guttman, no longer here to guide me, are always in my thoughts.

Some have graciously taken time to read all or part of this manuscript. I thank Amie Wilkinson, Paul Velleman, Howard Wainer, Wendell Garner, Michael Kubovy, Stephen Kosslyn, Wayne Oldford, David Scott, Cynthia Brewer, Alan MacEachren, Dianne Cook, Jürgen Symanzik, Jim Russell, and Stephen Stigler. And I especially thank a series editor at Springer-Verlag who made the effort to tell me what was sense and what was nonsense so I could revise intelligently.

At Springer-Verlag, I thank John Kimmel. John is a gentleman. A gentleman (or gentlewoman) is someone who never needs to sign a contract. I finally signed one for this book, but it was many years after I knew, and I know John knew, that I would not publish this book with anyone but John. I also thank Mary Ann Cottone for gentle persistence in coordinating the production of an extremely complex four-color manuscript.

My wife Ruth VanDemark has patiently lived with this book through many nights and weekends. And, while thanking family, I must confess a special pleasure in joining my son-in-law Benson Farb as a Springer author. Benson tutored me in the rudiments of geometry.

Lastly, the two to whom this book is dedicated — my mentor and my daughter. John Hartigan made me feel at home during my visits to the Yale statistics department in the early 1970's and encouraged my steps. Amie Wilkinson urged me to keep walking and taught me new steps. John and Amie share a remarkable intensity and an ability, rare among mathematicians, to explain to people like me what they do. This book is only a shadow of what they gave.

Chicago, Illinois Leland Wilkinson

Preface to Second Edition

The first edition of this book was a monograph. I intended to present a new, object-oriented way of thinking about statistical graphics. The second edition is a multigraph. I intend to supplement this presentation with a survey of ideas that are useful for understanding the meaning of statistical graphics. I have extended the grammar metaphor by calling the first part *Syntax*, and the second *Semantics*. Consequently, the second part of this book includes more of the work of others. Although it is not a survey, it covers the main themes that I consider essential to understanding how charts and graphs work.

Part 2 takes mathematical, psychological, and applied points of view. Quantitative graphics are meaningless unless they have mathematical foundations. They are solipsistic unless they follow experimentally tested psychological principles. And they are irrelevant unless they reveal patterns in real data so we can understand our world.

The first edition used "I" throughout. The second uses "we" because the visualization team I have been privileged to work with at SPSS has shaped my ideas as much as I have shaped theirs. Dan Rope, as I mentioned in the first edition, has been a master at developing these ideas; he has added management to his other talents without giving up coding. Graham Wills, in a fortunate circumstance, joined us shortly after the first edition of this book appeared; he brought unique energy (if you've met Graham you know what I mean) and extensive knowledge of graphics and statistics to our thinking. Andy Norton has been an old friend; he brought us a deep understanding of object-oriented design, Java, and database technology. And Roger Dubbs joined SPSS from a computer game company; it should be obvious what he brought us, in addition to a keen design sense and extensive knowledge of programming methodology. I should also mention Taylor Stockwell, Valeri McGuire, Will LaForest, and Fred Esch, who worked for Illumitek and eventually for SPSS; their enthusiasm and dedication remind me of the early days at SYSTAT in the 1980's.

Graham Wills contributed to the *Automation* chapter and developed the proofs in the *Algebra* chapter. He and Roger Dubbs wrote ViZml. Dan Rope contributed to the *Control* chapter and wrote the bulk of nViZn. Andy Norton

contributed to the *Time* chapter and wrote Dancer and the GPL interpreter. Roger Dubbs contributed to the *How to Make a Pie* chapter and, with Graham, wrote ViZml. I also thank Andy for an especially close reading of the entire book; all the mistakes in this book are Andy's fault, not mine.

As we mention in the text, this book is about more than visualization. It is about communicating information. This perspective unites the era of computing with the eras of mapping, charts, and graphics from previous centuries. We have tried to reference the original sources for ideas whenever possible and we have tried to avoid referencing rediscoveries and reinterpretations that appear in different fields unless they contribute substantial new information. If our references look quaint sometimes, it is because they are original. As much as possible, we have tried to avoid reprising Stigler's law of eponomy ("No scientific discovery is named after its original discoverer").

Finally, I want to thank Jack Noonan for supporting an environment that has demonstrated that discovery and innovation are possible in a publicly–owned company.

Leland Wilkinson
Chicago, 2005

Contents

1

Introduction

Grammar gives language rules. The word stems from the Greek noun for letter or mark (*γράμμα*). And that derives from the Greek verb for writing (*γράφω*), which is the source of our English word **graph**. Grammar means, more generally, rules for art and science, as in the richly illustrated *The Grammar of Ornament* (Jones, 1856), and Karl Pearson's *The Grammar of Science* (Pearson, 1892).

Grammar has a technical meaning in linguistics. In the transformational theory of Chomsky (1956), a grammar is a formal system of rules for generating lawful statements in a language. Chomsky helped explain many surface characteristics of specific natural languages through a deep, universal structure. Chomsky's context-free grammar is the progenitor of modern computer language parsers.

Grammar makes language expressive. A language consisting of words and no grammar (*statement* = *word*) expresses only as many ideas as there are words. By specifying how words are combined in statements, a grammar expands a language's scope.

This book is about grammatical rules for creating perceivable graphs, or what we call **graphics**. The grammar of graphics takes us beyond a limited set of charts (words) to an almost unlimited world of graphical forms (statements). The rules of graphics grammar are sometimes mathematical and sometimes aesthetic. Mathematics provides symbolic tools for representing abstractions. Aesthetics, in the original Greek sense, offers principles for relating sensory attributes (color, shape, sound, etc.) to abstractions. In modern usage, aesthetics can also mean taste. This book is not about good taste, practice, or graphic design, however. There are many fine guides to creating good graphics (*e.g.*, Cleveland, 1985, 1995; Tufte, 1983, 1990, 1997; Kosslyn, 1994). This book focuses instead on rules for constructing graphs mathematically and then representing them as graphics aesthetically.

The title of this book also recalls Bertin's *Semiology of Graphics* (1967), the first and most influential structural theory of statistical graphics. Bertin's work has pervaded our thinking. Semiology deals with signs. Although Bertin put his signs on paper, his work applies as well to virtual space.

Some of the rules and graphics in this book may seem self-evident, especially to those who have never written a computer program. Programming a computer exposes contradictions in commonsense thinking, however. And programming a computer to draw graphs teaches most surely the ancient lesson that God is in the details.

1.1 Graphics Versus Charts

We often call graphics **charts** (from χάρτης or Latin *charta*, a leaf of paper or papyrus). There are pie charts, bar charts, line charts, and so on. This book shuns chart typologies. For one thing, charts are usually instances of much more general objects. Once we understand that a pie is a divided bar in polar coordinates, we can construct other polar graphics that are less well known. We will also come to realize why a histogram is not a bar chart and why many other graphics that look similar nevertheless have different grammars.

There is also a practical reason for shunning chart typology. If we endeavor to develop a charting instead of a graphing program, we will accomplish two things. First, we inevitably will offer fewer charts than people want. Second, our package will have no deep structure. Our computer program will be unnecessarily complex, because we will fail to reuse objects or routines that function similarly in different charts. And we will have no way to add new charts to our system without generating complex new code. Elegant design requires us to think about a theory of graphics, not charts.

A chart metaphor is especially popular in user interfaces. The typical interface for a charting program is a catalog of little icons of charts. This is easy to construct from information gathered in focus groups, surveys, competitive analysis, and user testing. Much more difficult is to understand what users intend to do with their data when making a graphic. Instead of taking this risk, most charting packages channel user requests into a rigid array of chart types. To atone for this lack of flexibility, they offer a kit of post-creation editing tools to return the image to what the user originally envisioned. They give the user an impression of having explored data rather than the experience.

If a chart view is restrictive, how do we impose structure on a graphic view? The concept of a graphic is so general that we need organizing principles to create instances of graphics. We may not want to put a pie chart in a catalog, but we need to give users some simple way to produce one. For that, we need methodologies based on object-oriented design.

1.2 Object-Oriented Design

Many of the insights in this book were stimulated by a recent development in applied computer science called **object-oriented analysis and design** (Meyer, 1988; Rumbaugh *et al*., 1991; Booch, 1994). Object-oriented design (OOD) involves a plethora of techniques and principles that often baffle those trained in classical procedural programming languages and software design. Its methodology resembles a search for the objects that throw shadows on the wall of Plato's cave. Good objects are hard to find.

1.2.1 What is OOD?

OOD uses a variety of strategies for making software flexible and reusable:

- **Objects** are basic components of systems. They represent relatively autonomous agents that go about their business doing things useful for each other and for the general community of objects that comprise the system. The names of some of the most widely used objects in contemporary OOD systems express this utilitarian perspective: **Factory, Decorator, Facade, Proxy, Iterator, Observer, Visitor** (Gamma *et al.*, 1995). These objects do things that are aptly described by their names. A factory builds things. A decorator applies patterns to things. An observer looks for messages. A visitor roams and brings gifts.
- Objects communicate with each other through simple **messages**. These messages are distributed throughout the system. Because they may float freely throughout system, instead of being confined to the rigid protocols of classical programs, they resemble the communications within a living community.
- Objects are relatively stupid. They do a few things well, as do lobsters.
- Intelligence resides in the system, not in objects, because activities in concert have a life of their own that cannot be explained by separate, uncoordinated activities. For an OOD, as for life itself, the whole is more than the sum of its parts.
- Because objects are relatively stupid, they are also relatively simple and useful for a variety of purposes, even in new systems. Objects are often **reusable**, although this aspect of OOD has been oversold by some proponents.
- Because objects respond only to a few messages, and talk to other objects via simple messages of their own, what they do is **encapsulated**. Other objects have no idea how they work. And they don't care. They only need to know what to do with messages.
- Components of object-oriented systems are relatively **modular**. If parts of the system are discarded or malfunction, the rest of the system usually can continue to function.

- Objects can **inherit** attributes and behavior from other objects. This saves time and space in a well-organized system, because we can derive instances of things from more general classes of things.
- Objects are often **polymorphous**. That is, different objects can be induced to respond to the same message in different ways. Their responses may even be unanticipated by their designer, but in an elegant system their responses will not usually be harmful. Polymorphism also implies that objects don't care what type of data they process. They are flexible enough to return reasonable responses to all sorts of data. This includes the simplest response, which is not to respond. This kind of robustness is quite different from classical procedural systems which crash or cause other routines to crash when fed illegal or unanticipated data. Well-designed polymorphous objects are not perverse.
- OOD induces designers to **abstract**. Nate Kirby, an object-oriented programmer and designer at SPSS, has noted that bad programmers *ignore* details and bad designers *get lost in* details. To a designer, whenever a category or class of object seems fitting, it elicits thoughts of a more general category of which it is an instance. Because of this generalizing tendency, object-oriented systems are intrinsically **hierarchical**.

1.2.2 What is not OOD?

1.2.2.1 OOD is not a Language

OOD is not a programming language. Some languages, like Simula, Smalltalk, and Java, make it easy to implement objects and difficult to implement procedures. Others, like *C++*, enable development of both objects and procedures. And others, like *C*, Pascal, Ada, BASIC, Algol, Lisp, APL, and FORTRAN, make it difficult (but not impossible) to develop objects. Using a language that facilitates object specifications is no guarantee that a system will be object-oriented, however. Indeed, some commercial *C++* graphics and numerical libraries are translations of older FORTRAN and *C* procedures. These older routines are disguised in **wrappers** whose names are designed to make them appear to be objects. By the definitions in this book, a Java library with classes that are called PieChartModel, PieChartViewer, and PieChartController, is no more object-oriented than a FORTRAN program with three subroutines of the same names.

1.2.2.2 OOD is not a GUI

OOD has been associated with the development of modern graphical user interfaces (GUIs) because it is easiest to instantiate the behavior of an icon or graphic control through well-defined objects. OODs can be implemented in scripting or command-based systems, however, and GUIs with behavior indis-

tinguishable from object-driven ones can be programmed through direct calls to an operating system tool kit. It is extremely difficult (though not impossible) to infer the design of a system through its behavior.

1.2.2.3 OOD is not an Interactive System

While a modern desktop system tends to allow a user to interact with its components in a flexible manner, this has nothing to do with OOD. For example, Data Desk, the most interactive commercial statistics package (Velleman, 1998), is not based on OOD. This is not necessarily a drawback. Indeed, it can be an advantage. Data Desk's design has served the package well because it was conceived with a close attention to the capabilities of the operating systems under which it resides. The extent to which an OOD system is interactive depends on how controller classes are implemented. Without user controls, OOD systems may be relatively autonomous.

1.2.3 Why OOD?

OOD has failed to realize some of the more extravagant claims of its proponents in the last decade. In our experience, OOD has not brought increased reliability to the development process. Reliability of a system depends more on clean design and intensive testing early in the development process than on a particular design method. Nor has OOD given us increased portability of programs. Operating systems have evolved more rapidly in the last few years than ever before. Manufacturers' promises that their object **frameworks** (the objects programmers use for routine tasks) would remain immutable, or at least upward-compatible, have not been kept. Nor has OOD given us more rapid and responsive software. It is hard to beat assembly language or *C* programs at execution time. While there are exceptions, it is generally true that the most attractive elements of OOD — encapsulation and polymorphism — usually penalize performance. Nor has OOD given us more rapid development schedules. Indeed, OOD can retard development because objects are often more difficult to conceive properly, and modifying pre-existing objects is more difficult than changing procedures. Despite the marketing hype for OOD, it is hard to beat the development cycles realized in some of the APL and Lisp systems of two decades ago.

Still, an OOD paradigm is the best way to think about graphics. APL is ideally suited to developing small matrix algebra functions because it is a matrix functional language. It is unbeatable for prototyping numerical methods. Lisp is ideal for manipulating lists of words and symbols because it is a list processing language. OOD, by contrast, is a natural framework for thinking about graphics because graphics *are* objects (Hurley and Oldford, 1991). We can see and touch these objects. Having a language that naturally implements these objects is an added benefit. If none of this work appeared on a computer, however, we would still find the effort worthwhile. Defining objects helps organize thoughts.

1.3 An Object-Oriented Graphics System

A **graph** is a set of points. A mathematical graph cannot be seen. It is an abstraction. A **graphic**, however, is a physical representation of a graph. This representation is accomplished by realizing graphs with **aesthetic** attributes such as *size* or *color*.

An object-oriented graphics system requires explicit definitions for these realizations and rules for relating them to data and for organizing their behavior in a computational environment. If we are lucky, this graphics system should have generality, yet will rest on a few simple objects and rules. This book is an attempt to reveal the richness of such a system.

From the OOD perspective, graphics are collections of objects. If the messages between these objects follow a simple grammar, then they will behave consistently and flexibly. To introduce this idea, we will focus on three stages of graphic creation:

1) Specification
2) Assembly
3) Display

After introducing these stages, we will show how they work in an example.

1.3.1 Specification

Specification involves the translation of user actions into a formal language. The user may not be aware of this language, but it is required for an automated system to understand the graphic request. Another way of defining specification is to say that it is the deep grammar of a graphic. A graphic, unlike a picture, has a highly organized and constrained set of rules. A picture, of course, has its own rules, especially real pictures such as photographs and videos (Biederman, 1981). Nevertheless, an artist is privileged to bend the rules to make a point (Bosch, Dali, or Picasso, obviously, but also Rembrandt, Cezanne, or Close). And a special-effects technician may use tricks to make us think that a video or virtual scene is more real than what we observe in our daily lives. Not so with graphics. We have only a few rules and tools. We cannot change the location of a point or the color of an object (assuming these are data-representing attributes) without lying about our data and violating the purpose of the statistical graphic — to represent data accurately and appropriately. Consequently, the core of a graphics system must rest on specification.

Statistical graphic specifications are expressed in six statements:

1) DATA: a set of data operations that create variables from datasets,
2) TRANS: variable transformations (*e.g., rank*),
3) SCALE: scale transformations (*e.g.*, *log*),
4) COORD: a coordinate system (*e.g.*, *polar*),
5) ELEMENT: graphs (*e.g.*, *points*) and their aesthetic attributes (*e.g.*, *color*),
6) GUIDE: one or more guides (*axes*, *legends*, etc.).

In most of the figures in this book, we will add a syntactical specification of the graphic in order to make the definition explicit. An earlier version of this specification language (Wilkinson, 1996) incorporated all aspects in a single algebra. The notation was unwieldy and idiosyncratic, however, so we have separated them into components. These components link data to objects and specify a scene containing those objects.

1.3.2 Assembly

A scene and its description are different. In order to portray a scene, we must coordinate its geometry, layout, and aesthetics in order to render it accurately. A statistical graphics computer program must be able to assemble a graphical scene from a specification in the same manner as a drawing or modeling program puts together a realistic scene from specification components. This book is more about specification than scene assembly, but it is important to think about assembly while learning about specification so that we do not confuse surface features with deep structures. How we build a scene from a specification affects how the result behaves. A scene can be dynamic or static, linked to external data or isolated, modifiable or immutable, depending on how we assemble it.

1.3.3 Display

For us to perceive it, a graph must be rendered using aesthetic attributes and a display system (*e.g.*, paper, video, hologram). Contemporary operating systems provide access to rendering methods, preferably through well-defined, abstract interfaces. Production graphics require little in this area other than basic drawing capabilities. Dynamic graphics and scientific visualization, by contrast, require sophisticated designs to enable **brushing**, **drill-down**, **zooming**, **linking**, and other operations relating data to graphics. Becker and Cleveland (1987), Cleveland and McGill (1988), Cook and Weisberg (1994), and Swayne *et al.* (1998) introduce these issues. More recently, virtual reality displays and immersive environments have expanded the available aesthetics to touch and sound.

1.4 An Example

Figure 1.1 shows a graphic of 1990 death rates against birth rates per 100,000 population for 27 selected countries in a UN databank. The plot contains two graphic elements: a *point* (collection of points) whose labels show country names, and a *contour* of a kernel density estimate (Silverman, 1986) that represents the concentration of the countries. We have also included three *guides* that help us understand the graphics. The first is a general geometric object called a *form* that is, in this instance, a line delineating zero population growth. Countries to the left of this line tend to lose population, and countries to the right tend to gain. The other two are guides that delineate axes for the represented space. Other examples of guides are legends and titles.

The graphic is striking because it reveals clearly the patterns of explosive population growth. The density contours show two concentrations of countries. One, to the left, has relatively lower death rates and small-to-moderate birth rates. The second, in the upper right, has high death rates and extraordinarily high birth rates. The latter countries tend to be developing. We have kept the sample of countries small so that we can read the country labels. Adding other countries from the database does not change the overall picture.

```
ELEMENT: point(position(birth*death), size(0), label(country))
ELEMENT: contour(position(
           smooth.density.kernel.epanechnikov.joint(birth*death)),
           color.hue())
GUIDE: form.line(position((0,0),(30,30)), label("Zero Population Growth"))
GUIDE: axis(dim(1), label("Birth Rate"))
GUIDE: axis(dim(2), label("Death Rate"))
```

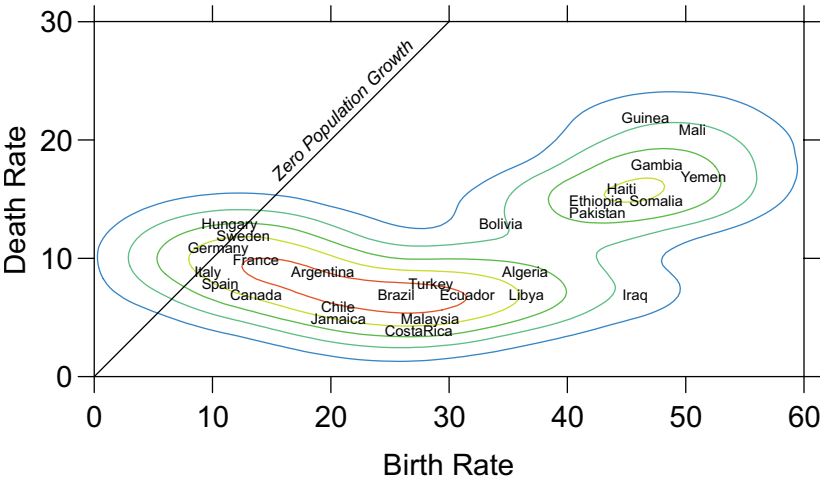

Figure 1.1 *Plot of death rates against birth rates for selected countries*

1.4.1 Specification

The specification above the figure makes use of only ELEMENT, and GUIDE components. The data are assumed to have been organized in a cases-by-variables matrix, there are no transformations, and the coordinates are rectangular, so we can assume default settings. The first two lines show the two graphic elements in the plot: a *point*, and a *contour*. Both graphic elements are positioned by the variables birth and death, which are scaled as percents. The frame in which they are embedded is determined by the algebraic expression birth*death. The *point* graphic actually does not show because its *size* attribute is set to zero. Normally, we would see symbols (perhaps dots) for each country. Instead, we have country labels for each point in the cloud set to the values of the variable country in the data. The *contour* graphic represents the density of countries in different regions of the frame. Where there are more countries near each other, the density contour is higher. These contours are computed by a kernel smoothing algorithm that we will discuss further in Chapter 7. The dot notation *smooth.density.kernel.epanechnikov.joint*() means that Epanechnikov kernel smoothing is a member of a hierarchy of density smoothing methods. Different contours are given a *color.hue* aesthetic based on the kernel density values at the level of each contour.

The guides consist of the line, the axes, and their corresponding scales and labels. The *form* guide is displayed with a line from (0,0) to (30,30) in the metric anchored by both rate measures. This line is labeled with an associated text string ("Zero Population Growth"). In most of the figures, we will omit GUIDE specifications to keep the description simpler.

1.4.2 Assembly

Assembling a scene from a specification requires a variety of structures in order to index and link components with each other. One of the structures we can use is a network or a tree. Figure 1.2 shows a tree for the graphic in Figure 1.1. Each node in the tree, shown with a box, represents a type of object in Figure 1.1. Each branch in the tree, shown with an arrow, represents a type of relation between objects. The triangular-headed arrows represent “is a” relations. The diamond-headed arrows represent “has a” relations.

"Is a" relations provide a way to derive common functionality from one class. The result of such relations is **inheritance**. For example, an Axis is a Guide in the same sense that a piano is a keyboard instrument. Any aspect of a piano that has to do with being a keyboard instrument (having a sound produced by pressing one or more keys, for example) is inherited by other keyboard instruments. Any aspect of a piano that does not have to do with being a keyboard instrument (having hammers, for example) is not necessarily shared by other keyboard instruments (harpsichords pluck, pianos strike). If we derive common functionality from a general class, then a subclass can inherit skills from its parent without having to do any extra work. Tasks related

to having keys, for example, can be defined and implemented in one Keyboard class. Tasks relating to guiding, such as relating numeric values to text strings, can be implemented in one Guide class. In a similar manner, the Contour and Point classes are both Graphs. They inherit capabilities that enable them to represent themselves in a frame.

"Has a" relations provide a way to group related attributes and functions under a class. The result of such relations is **aggregation**. For example, an Axis has a Scale, a Rule, and a Label in the same sense that a piano has a keyboard, strings, and pedals. The aggregation of these features and functions is what helps us distinguish a piano from other objects. In a similar manner, a Chart has a Frame, one or more Guides, and one or more Graphs. If we implement aggregation well, our objects will be small and efficient and our computer code will be comprehensible and modular.

.

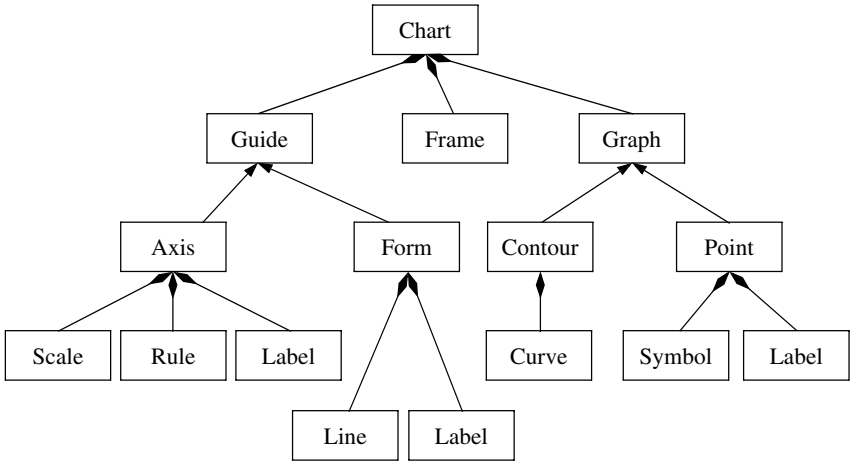

Figure 1.2 *Design tree for chart in Figure 1.1*

1.4.3 Display

The tree in Figure 1.2, together with a set of rendering tools (symbols, polylines, polygons) and layout designer, provides a structured environment in which each object in a graphic can draw itself. There is no single agent needed to figure out all the rules for drawing each object.

A grammar of graphics facilitates coordinated activity in a set of relatively autonomous components. This grammar enables us to develop a system in which adding a graphic to a frame (say, a *surface*) requires no adjustments or changes in definitions other than the simple message “add this graphic.” Similarly, we can remove graphics, transform scales, permute attributes, and make other alterations without redefining the basic structure.

1.4.4 Revision

Revision means, literally, to see again. For graphics, it implies that we want to change, query, and explore without having to go through all the work of specifying and creating a new graphic. By carefully separating the process of graph creation into hierarchical components, we enable a flexible environment that offers new views without recalculating every step in the system. And we can link controllers to any component or property in the system to provide direct manipulation of data, variables, frames, or rendering. If more than one graphic depends on the same sub-component, then they are linked as well.

Figure 1.3 shows an example. Even though the graphic looks different, the positional frame is the same as in Figure 1.1. We have omitted *point* and *form* and we have replaced *contour* with *polygon* to represent a kernel density. The *hue* of each polygon comes from the estimated density of the countries at that location. We have omitted the guides from this and subsequent specifications to save space. These will be discussed in more detail in Chapter 12.

ELEMENT: *polygon*(*position*(
smooth.density.kernel.epanechnikov.joint(birth*death)), *color.hue*())

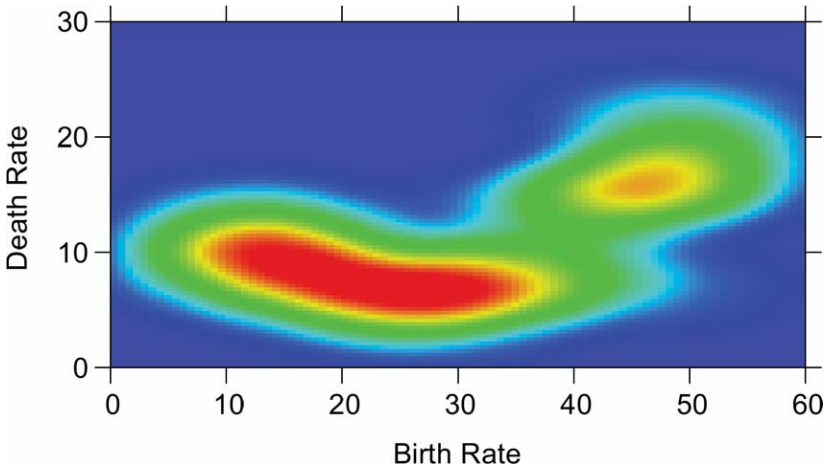

Figure 1.3 *Kernel density of death and birth rates*

Figure 1.4 adds a new variable to the specification of the point graphic in Figure 1.1. This variable, military, is the annual military expenditures per capita normalized as U.S. dollar equivalents. We are using this variable to determine the size of each symbol, so that size of plotting symbol represents military expenditure for each country.

ELEMENT: *contour*(*position*(
smooth.density.kernel.epanechnikov.joint(birth*death), *color.hue*())
ELEMENT: *point*(*position*(birth*death), *size*(military))

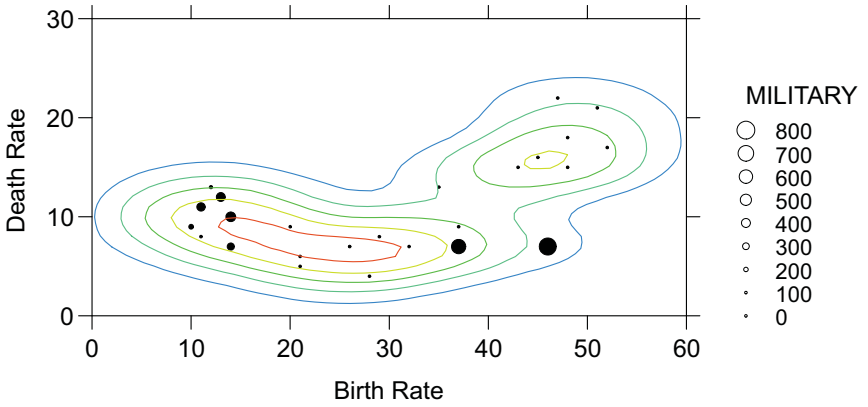

Figure 1.4 *Military expenditures vs. birth and death rates*

Figure 1.4 conveys a rather troubling message. The highest relative military expenditures are often in the most rapidly growing, politically unstable countries. Still, this statistic conceals the *absolute* level of military expenditures. The highest absolute levels of military spending are in advanced nations with larger populations.

Finally, Figure 1.5 shows a map of the difference between death and birth for the selected countries. The goal is to reveal the location of countries that are growing rapidly in population. We have used size of the plotting symbol to represent the magnitude of the difference (the few small negative differences have been set to zero).

There are two sets of positioning variables that define the frame. The first, lat and lon, represent the location of the countries measured. These are used to plot the circles showing death-birth differences. The second, latitude and longitude, are used to denote the locations on the map that anchor the boundaries of the polygons defining the continental borders. The polygons for the map are read from a shape file containing their vertices. The *map* data function handles the translation of polygon IDs and polygon vertices in the file to a splitter variable and longitude and latitude vertices for the *polygon* geometric function. The *point* and *polygon* graphics both use a *position* attribute to control which variables determine their position. The *position* dimensions are transformed with a *mercator* cartographic projection. The axes and grid lines respond to the projection, as well as the graphics in the frame. We will examine in Chapter 9 map projections that are better suited for representing the countries data.

```
DATA: longitude, latitude = map(source("World"))
TRANS: bd = max(birth-death, 0)
COORD: project.mercator()
ELEMENT: point(position(lon*lat), size(bd), color(color.red))
ELEMENT: polygon(position(longitude*latitude))
```

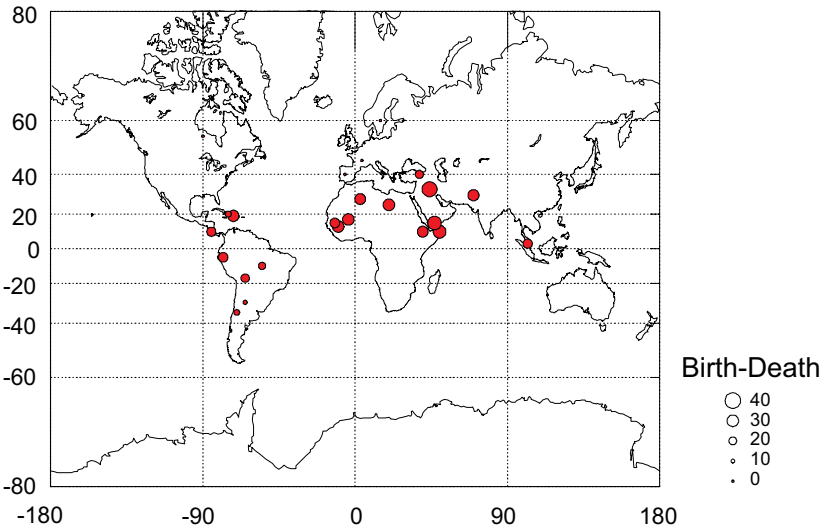

Figure 1.5 *Excess birth (vs. death) rates in selected countries*

1.5 What This Book Is Not

Because this book spans the fields of computer science, geography, statistics, and graphic design, it is likely to be misunderstood by specialists in different areas for different reasons. We cannot anticipate all of these reasons, but here are a likely few.

1.5.1 Not a Command Language

A cursory reading of this book might lead one to conclude that its purpose is to present a new graphics scripting language. Indeed, each figure is accompanied by a specification that resembles a command language. One impetus for this conclusion would be occasional similarities to existing quantitative graphics languages such as those in Mathematica®, SYSTAT®, S-Plus®, and SAS-Graph®. These packages can produce a large variety of statistical graphics because they evolved to fulfill the needs of statisticians for sophisticated and flexible technical graphics. They were not developed with a comprehensive theory of graphics in mind, however. Often, their constructs have similar reg-

ularities because of the constraints of the graphics world. We owe a debt to all of these systems for being able to produce unusual graphics with them and to discover the common implicit structures. To appreciate the real difference between this and command-based systems, however, see Chapter 17.

Another conclusion one might draw after a brief glance is that this system is designed to be a static specification language instead of a dynamic, exploratory system. On the contrary, by regularizing the rules for graph behavior in graphics frames, it provides a richer environment for dynamic and exploratory graphics. This is especially true for paneled graphics, which are either avoided altogether in most dynamic graphics systems or hard-wired to specific data structures. In fact, the primary focus of our interest is in designing a system that is flexible enough to change state without re-specification. A naive approach to implementing such a system would be to create commands from user gestures, feed those commands to an interpreter, and then display the results. This method, employed in some existing packages, would indeed constrain it to be a static system. There is nothing in the theory presented in this book, however, to suggest that this is the best or even most appropriate implementation method.

1.5.2 Not a Taxonomy

Taxonomies are useful to scientists when they lead to new theory or stimulate insights into a problem that previous theorizing might conceal. Classification for its own sake, however, is as unproductive in design as it is in science. In design, objects are only as useful as the system they support. And the test of a design is its ability to handle scenarios that include surprises, exceptions, and strategic reversals. This book includes a few classifications, but they are in the service of developing objects that are flexible and powerful in a coherent system. Other classifications of the same problem domain are possible, but many of them would not lead to a parsimonious and general system. Some classifications have been attempted based on cluster analyses of ordinary users' visual judgments of similarities among real statistical graphics (*e.g.*, Lohse *et al*., 1994). This approach may be useful for developing interfaces but contributes nothing to a deeper understanding of graphs. Customary usage and standards can blind us to the diversity of the graphics domain; a formal system can liberate us from conventional restrictions.

1.5.3 Not a Drafting Package

This system was not designed to produce any graphic imaginable. Indeed, the motivation is almost the opposite: to develop a closed system and *then* to examine whether it can produce both popular and esoteric graphics. We have tried to avoid adding functions, graphs, or operators that do not work independently across the system. There are doubtless many statistical graphics the

system in this book cannot completely specify. We can add many different graphs, transformations, types of axes, annotations, etc., but there are two limitations we will always face with a formal system.

The first limitation applies to any free-hand drawing. Clearly, we cannot expect to use a formal data-driven system to produce sketches on cocktail napkins. It will always be possible to find creative designs that are not formally linked to data. The province of drafting systems is computer-assisted design (CAD) and desktop publishing (DTP). Those areas have their own rules driven more by the physical appearance of real objects than by the theoretical constructs of functional and data analysis.

The second limitation derives from the syntactical structure of the system itself. The operators in this system are capable, as we shall see, of producing a surprisingly wide variety of graphics, perhaps more than any other formal system or computer graphing program. Nevertheless, one can imagine certain structures that may not be modeled by a language with the operators presented here. It is, after all, a closed system. This graphics system was designed with surveys of statistical graphics usage (*e.g.*, Fienberg, 1979) and existing commercial and scientific graphics software in mind. Nevertheless, one cannot over-estimate the inventiveness and ingenuity of real users when they display their ideas.

1.5.4 Not a Book of Virtues

This system is capable of producing some hideous graphics. There is nothing in its design to prevent its misuse. We will occasionally point out some of these instances (*e.g.*, Figure 9.25). That the system *can* produce such graphics is simply a consequence of its basis on the mathematical rules that determine the meaning of graphs, rather than on the *ad hoc* rules we sometimes use to produce graphics. These rules are not based on personal preferences but rather on the mathematics and perceptual dimensions underlying the graphics we draw in practice. These rules are just as capable of producing graphics for *USA Today* as for *Scientific American*.

This system cannot produce a meaningless graphic, however. This is a strong claim, vulnerable to a single counter-example. It is a claim based on the formal rules of the system, however, not on the evaluation of specific graphics it may produce. This is an essential difference between the approach in this book and in other texts on statistical graphics and visualization. We are much less interested in designing or evaluating specific graphics than in understanding the rules that produced them. Unless one specifies those rules explicitly, one cannot begin to claim that a particular graphic is meaningless or not.

We also cannot disagree strongly enough with statements about the dangers of putting powerful tools in the hands of novices. Computer algebra, statistics, and graphics systems provide plenty of rope for novices to hang themselves and may even help to inhibit the learning of essential skills needed by researchers. The obvious problems caused by this situation do not justify