CHAPTER 1

The Seven Stages of Visualizing Data

The greatest value of a picture is when it forces *us to notice what we never expected to see.*

—John Tukey

What do the paths that millions of visitors take through a web site look like? How do the 3.1 billion A, C, G, and T letters of the human genome compare to those of the chimp or the mouse? Out of a few hundred thousand files on your computer's hard disk, which ones are taking up the most space, and how often do you use them? By applying methods from the fields of computer science, statistics, data mining, graphic design, and visualization, we can begin to answer these questions in a meaningful way that also makes the answers accessible to others.

All of the previous questions involve a large quantity of data, which makes it extremely difficult to gain a "big picture" understanding of its meaning. The problem is further compounded by the data's continually changing nature, which can result from new information being added or older information continuously being refined. This deluge of data necessitates new software-based tools, and its complexity requires extra consideration. Whenever we analyze data, our goal is to highlight its features in order of their importance, reveal patterns, and simultaneously show features that exist across multiple dimensions.

This book shows you how to make use of data as a resource that you might otherwise never tap. You'll learn basic visualization principles, how to choose the right kind of display for your purposes, and how to provide interactive features that will bring users to your site over and over again. You'll also learn to program in Processing, a simple but powerful environment that lets you quickly carry out the techniques in this book. You'll find Processing a good basis for designing interfaces around large data sets, but even if you move to other visualization tools, the ways of thinking presented here will serve you as long as human beings continue to process information the same way they've always done.

Why Data Display Requires Planning

Each set of data has particular display needs, and the *purpose* for which you're using the data set has just as much of an effect on those needs as the data itself. There are dozens of quick tools for developing graphics in a cookie-cutter fashion in office programs, on the Web, and elsewhere, but complex data sets used for specialized applications require unique treatment. Throughout this book, we'll discuss how the characteristics of a data set help determine what kind of visualization you'll use.

Too Much Information

When you hear the term "information overload," you probably know exactly what it means because it's something you deal with daily. In Richard Saul Wurman's book *Information Anxiety* (Doubleday), he describes how the *New York Times* on an average Sunday contains more information than a Renaissance-era person had access to in his entire lifetime.

But this is an exciting time. For $300, you can purchase a commodity PC that has thousands of times more computing power than the first computers used to tabulate the U.S. Census. The capability of modern machines is astounding. Performing sophisticated data analysis no longer requires a research laboratory, just a cheap machine and some code. Complex data sets can be accessed, explored, and analyzed by the public in a way that simply was not possible in the past.

The past 10 years have also brought about significant changes in the graphic capabilities of average machines. Driven by the gaming industry, high-end 2D and 3D graphics hardware no longer requires dedicated machines from specific vendors, but can instead be purchased as a $100 add-on card and is standard equipment for any machine costing $700 or more. When not used for gaming, these cards can render extremely sophisticated models with thousands of shapes, and can do so quickly enough to provide smooth, interactive animation. And these prices will only decrease—within a few years' time, accelerated graphics will be standard equipment on the aforementioned commodity PC.

Data Collection

We're getting better and better at collecting data, but we lag in what we can do with it. Most of the examples in this book come from freely available data sources on the Internet. Lots of data is out there, but it's not being used to its greatest potential because it's not being visualized as well as it could be. (More about this can be found in Chapter 9, which covers places to find data and how to retrieve it.)

With all the data we've collected, we still don't have many satisfactory answers to the sort of questions that we started with. This is the greatest challenge of our information-rich era: how can these questions be answered quickly, if not instantaneously? We're getting so good at measuring and recording things, why haven't we kept up with the methods to understand and communicate this information?

Thinking About Data

We also do very little sophisticated thinking about information itself. When AOL released a data set containing the search queries of millions of users that had been "randomized" to protect the innocent, articles soon appeared about how people could be identified by—and embarrassed by—information regarding their search habits. Even though we can collect this kind of information, we often don't know quite what it means. Was this a major issue or did it simply embarrass a few AOL users? Similarly, when millions of records of personal data are lost or accessed illegally, what does that mean? With so few people addressing data, our understanding remains quite narrow, boiling down to things like, "My credit card number might be stolen" or "Do I care if anyone sees what I search?"

Data Never Stays the Same

We might be accustomed to thinking about data as fixed values to be analyzed, but data is a moving target. How do we build representations of data that adjust to new values every second, hour, or week? This is a necessity because most data comes from the real world, where there are no absolutes. The temperature changes, the train runs late, or a product launch causes the traffic pattern on a web site to change drastically.

What happens when things start moving? How do we interact with "live" data? How do we unravel data as it changes over time? We might use animation to play back the evolution of a data set, or interaction to control what time span we're looking at. How can we write code for these situations?

What Is the Question?

As machines have enormously increased the capacity with which we can create (through measurements and sampling) and store data, it becomes easier to disassociate the data from the original reason for collecting it. This leads to an all-too frequent situation: approaching visualization problems with the question, "How can we possibly understand so much data?"

As a contrast, think about subway maps, which are abstracted from the complex shape of the city and are focused on the rider's goal: to get from one place to the next. Limiting the detail of each shape, turn, and geographical formation reduces this complex data set to answering the rider's question: "How do I get from point A to point B?"

Harry Beck invented the format now commonly used for subway maps in the 1930s, when he redesigned the map of the London Underground. Inspired by the layout of circuit boards, the map simplified the complicated Tube system to a series of vertical, horizontal, and 45° diagonal lines. While attempting to preserve as much of the relative physical layout as possible, the map shows only the connections between stations, as that is the only information that riders use to decide their paths.

When beginning a visualization project, it's common to focus on all the data that has been collected so far. The amounts of information might be enormous—people like to brag about how many gigabytes of data they've collected and how difficult their visualization problem is. But great information visualization never starts from the standpoint of the data set; it starts with questions. Why was the data collected, what's interesting about it, and what stories can it tell?

The most important part of understanding data is identifying the question that you want to answer. Rather than thinking about the data that was collected, think about how it will be used and work backward to what was collected. You collect data because you want to know something about it. If you don't really know why you're collecting it, you're just hoarding it. It's easy to say things like, "I want to know what's in it," or "I want to know what it means." Sure, but what's meaningful?

The more specific you can make your question, the more specific and clear the visual result will be. When questions have a broad scope, as in "exploratory data analysis" tasks, the answers themselves will be broad and often geared toward those who are themselves versed in the data. John Tukey, who coined the term Exploratory Data Analysis, said "...pictures based on exploration of data should force their messages upon us."* Too many data problems are labeled "exploratory" because the data collected is overwhelming, even though the original purpose was to answer a specific question or achieve specific results.

One of the most important (and least technical) skills in understanding data is asking good questions. An appropriate question shares an interest you have in the data, tries to convey it to others, and is curiosity-oriented rather than math-oriented. Visualizing data is just like any other type of communication: success is defined by your audience's ability to pick up on, and be excited about, your insight.

Admittedly, you may have a rich set of data to which you want to provide flexible access by not defining your question too narrowly. Even then, your goal should be to highlight key findings. There is a tendency in the visualization field to borrow from the statistics field and separate problems into *exploratory* and *expository*, but for the purposes of this book, this distinction is not useful. The same methods and process are used for both.

In short, a proper visualization is a kind of narrative, providing a clear answer to a question without extraneous details. By focusing on the original intent of the question, you can eliminate such details because the question provides a benchmark for what is and is not necessary.

* Tukey, John Wilder. *Exploratory Data Analysis*. Reading, MA: Addison-Wesley, 1977.

A Combination of Many Disciplines

Given the complexity of data, using it to provide a meaningful solution requires insights from diverse fields: statistics, data mining, graphic design, and information visualization. However, each field has evolved in isolation from the others.

Thus, visual design—-the field of mapping data to a visual form—typically does not address how to handle thousands or tens of thousands of items of data. Data mining techniques have such capabilities, but they are disconnected from the means to interact with the data. Software-based information visualization adds building blocks for interacting with and representing various kinds of abstract data, but typically these methods undervalue the aesthetic principles of visual design rather than embrace their strength as a necessary aid to effective communication. Someone approaching a data representation problem (such as a scientist trying to visualize the results of a study involving a few thousand pieces of genetic data) often finds it difficult to choose a representation and wouldn't even know what tools to use or books to read to begin.

Process

We must reconcile these fields as parts of a single process. Graphic designers can learn the computer science necessary for visualization, and statisticians can communicate their data more effectively by understanding the visual design principles behind data representation. The methods themselves are not new, but their isolation within individual fields has prevented them from being used together. In this book, we use a process that bridges the individual disciplines, placing the focus and consideration on how data is understood rather than on the viewpoint and tools of each individual field.

The process of understanding data begins with a set of numbers and a question. The following steps form a path to the answer:

Acquire
: Obtain the data, whether from a file on a disk or a source over a network.

Parse
: Provide some structure for the data's meaning, and order it into categories.

Filter
: Remove all but the data of interest.

Mine
: Apply methods from statistics or data mining as a way to discern patterns or place the data in mathematical context.

Represent
: Choose a basic visual model, such as a bar graph, list, or tree.

Refine
: Improve the basic representation to make it clearer and more visually engaging.

Interact
: Add methods for manipulating the data or controlling what features are visible.

Of course, these steps can't be followed slavishly. You can expect that they'll be involved at one time or another in projects you develop, but sometimes it will be four of the seven, and at other times all of them.

Part of the problem with the individual approaches to dealing with data is that the separation of fields leads to different people each solving an isolated part of the problem. When this occurs, something is lost at each transition—like a "telephone game" in which each step of the process diminishes aspects of the initial question under consideration. The initial format of the data (determined by how it is acquired and parsed) will often drive how it is considered for filtering or mining. The statistical method used to glean useful information from the data might drive the initial presentation. In other words, the final representation reflects the results of the statistical method rather than a response to the initial question.

Similarly, a graphic designer brought in at the next stage will most often respond to specific problems with the representation provided by the previous steps, rather than focus on the initial question. The visualization step might add a compelling and interactive means to look at the data filtered from the earlier steps, but the display is inflexible because the earlier stages of the process are hidden. Furthermore, practitioners of each of the fields that commonly deal with data problems are often unclear about how to traverse the wider set of methods and arrive at an answer.

This book covers the whole path from data to understanding: the transformation of a jumble of raw numbers into something coherent and useful. The data under consideration might be numbers, lists, or relationships between multiple entities.

It should be kept in mind that the term *visualization* is often used to describe the art of conveying a physical relationship, such as the subway map mentioned near the start of this chapter. That's a different kind of analysis and skill from *information visualization*, where the data is primarily numeric or symbolic (e.g., A, C, G, and T—the letters of genetic code—and additional annotations about them). The primary focus of this book is information visualization: for instance, a series of numbers that describes temperatures in a weather forecast rather than the shape of the cloud cover contributing to them.

An Example

To illustrate the seven steps listed in the previous section, and how they contribute to effective information visualization, let's look at how the process can be applied to understanding a simple data set. In this case, we'll take the zip code numbering system that the U.S. Postal Service uses. The application is not particularly advanced, but it provides a skeleton for how the process works. (Chapter 6 contains a full implementation of the project.)

What Is the Question?

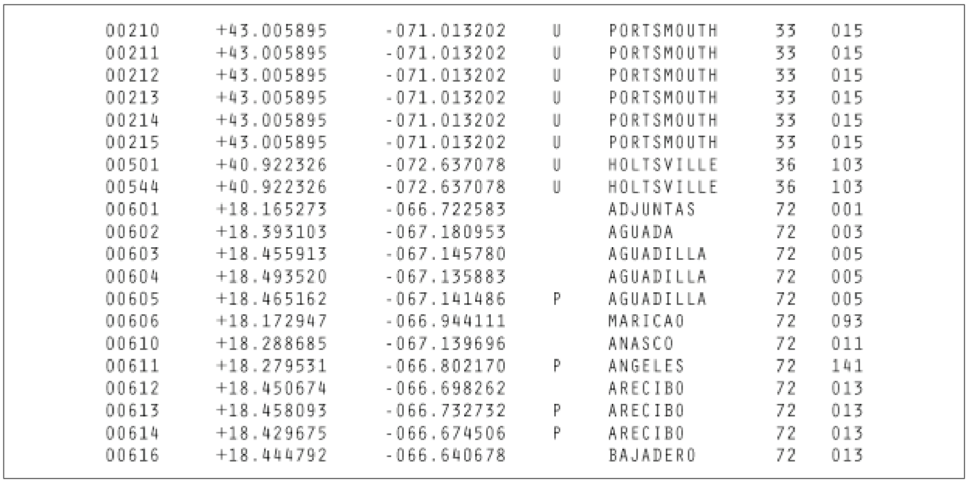
All data problems begin with a question and end with a narrative construct that provides a clear answer. The Zipdecode project (described further in Chapter 6) was developed out of a personal interest in the relationship of the zip code numbering system to geographic areas. Living in Boston, I knew that numbers starting with a zero denoted places on the East Coast. Having spent time in San Francisco, I knew the initial numbers for the West Coast were all nines. I grew up in Michigan, where all our codes were four-prefixed. But what sort of area does the second digit specify? Or the third?

The finished application was initially constructed in a few hours as a quick way to take what might be considered a boring data set (a long list of zip codes, towns, and their latitudes and longitudes) and create something engaging for a web audience that explained how the codes related to their geography.

Acquire

The acquisition step involves obtaining the data. Like many of the other steps, this can be either extremely complicated (i.e., trying to glean useful data from a large system) or very simple (reading a readily available text file).

A copy of the zip code listing can be found on the U.S. Census Bureau web site, as it is frequently used for geographic coding of statistical data. The listing is a freely available file with approximately 42,000 lines, one for each of the codes, a tiny portion of which is shown in Figure 1-1.



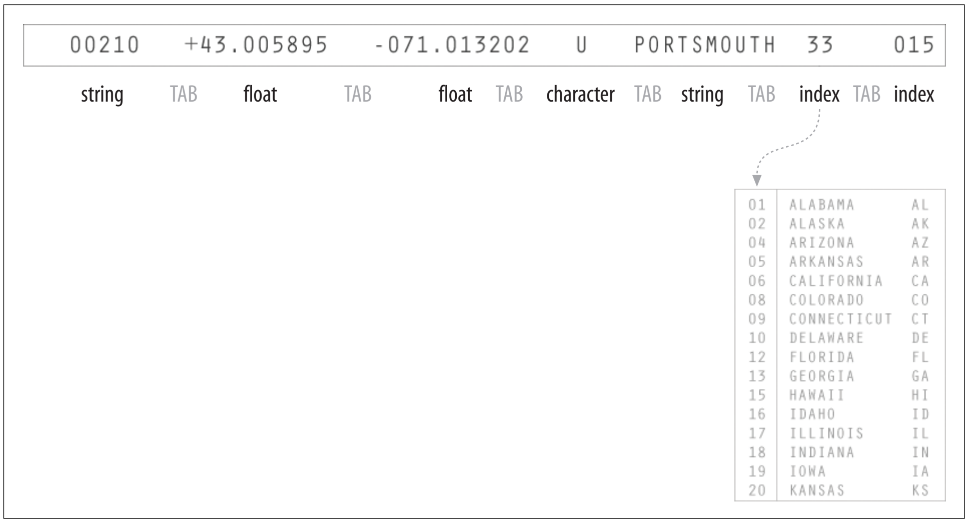
```
00210   +43.005895   -071.013202   U   PORTSMOUTH   33   015
00211   +43.005895   -071.013202   U   PORTSMOUTH   33   015
00212   +43.005895   -071.013202   U   PORTSMOUTH   33   015
00213   +43.005895   -071.013202   U   PORTSMOUTH   33   015
00214   +43.005895   -071.013202   U   PORTSMOUTH   33   015
00215   +43.005895   -071.013202   U   PORTSMOUTH   33   015
00501   +40.922326   -072.637078   U   HOLTSVILLE   36   103
00544   +40.922326   -072.637078   U   HOLTSVILLE   36   103
00601   +18.165273   -066.722583       ADJUNTAS     72   001
00602   +18.393103   -067.180953       AGUADA       72   003
00603   +18.455913   -067.145780       AGUADILLA    72   005
00604   +18.493520   -067.135883       AGUADILLA    72   005
00605   +18.465162   -067.141486   P   AGUADILLA    72   005
00606   +18.172947   -066.944111       MARICAO      72   093
00610   +18.288685   -067.139696       ANASCO       72   011
00611   +18.279531   -066.802170   P   ANGELES      72   141
00612   +18.450674   -066.698262       ARECIBO      72   013
00613   +18.458093   -066.732732   P   ARECIBO      72   013
00614   +18.429675   -066.674506   P   ARECIBO      72   013
00616   +18.444792   -066.640678       BAJADERO     72   013
```

Figure 1-1. Zip codes in the format provided by the U.S. Census Bureau

Acquisition concerns how the user downloads your data as well as how you obtained the data in the first place. If the final project will be distributed over the Internet, as you design the application, you have to take into account the time required to download data into the browser. And because data downloaded to the browser is probably part of an even larger data set stored on the server, you may have to structure the data on the server to facilitate retrieval of common subsets.

Parse

After you acquire the data, it needs to be parsed—changed into a format that tags each part of the data with its intended use. Each line of the file must be broken along its individual parts; in this case, it must be delimited at each tab character. Then, each piece of data needs to be converted to a useful format. Figure 1-2 shows the layout of each line in the census listing, which we have to understand to parse it and get out of it what we want.



```
00210   +43.005895   -071.013202   U   PORTSMOUTH   33   015
```

01	ALABAMA	AL
02	ALASKA	AK
04	ARIZONA	AZ
05	ARKANSAS	AR
06	CALIFORNIA	CA
08	COLORADO	CO
09	CONNECTICUT	CT
10	DELAWARE	DE
12	FLORIDA	FL
13	GEORGIA	GA
15	HAWAII	HI
16	IDAHO	ID
17	ILLINOIS	IL
18	INDIANA	IN
19	IOWA	IA
20	KANSAS	KS

Figure 1-2. Structure of acquired data

Each field is formatted as a data type that we'll handle in a conversion program:

String
: A set of characters that forms a word or a sentence. Here, the city or town name is designated as a string. Because the zip codes themselves are not so much numbers as a series of digits (if they were numbers, the code 02139 would be stored as 2139, which is not the same thing), they also might be considered strings.

Float
: A number with decimal points (used for the latitudes and longitudes of each location). The name is short for *floating point*, from programming nomenclature that describes how the numbers are stored in the computer's memory.

Character

A single letter or other symbol. In this data set, a character sometimes designates special post offices.

Integer

A number without a fractional portion, and hence no decimal points (e.g., –14, 0, or 237).

Index

Data (commonly an integer or string) that maps to a location in another table of data. In this case, the index maps numbered codes to the names and two-digit abbreviations of states. This is common in databases, where such an index is used as a pointer into another table, sometimes as a way to compact the data further (e.g., a two-digit code requires less storage than the full name of the state or territory).

With the completion of this step, the data is successfully tagged and consequently more useful to a program that will manipulate or represent it in some way.

Filter

The next step involves filtering the data to remove portions not relevant to our use. In this example, for the sake of keeping it simple, we'll be focusing on the contiguous 48 states, so the records for cities and towns that are not part of those states—Alaska, Hawaii, and territories such as Puerto Rico—are removed. Another project could require significant mathematical work to place the data into a mathematical *model* or normalize it (convert it to an acceptable range of numbers).

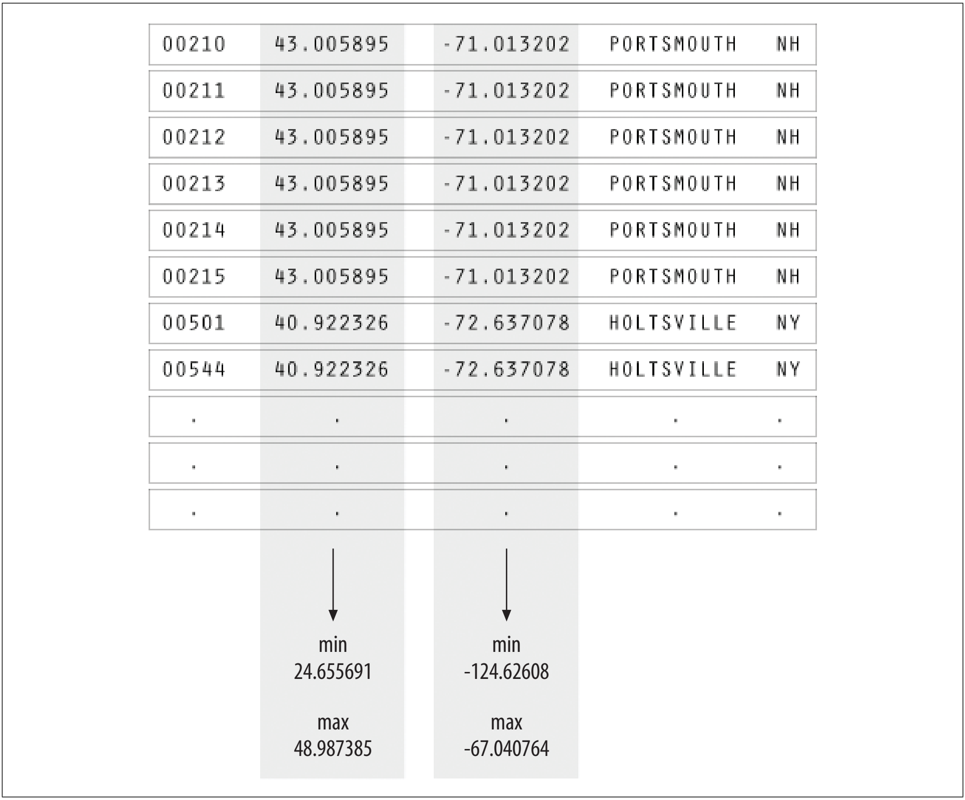
Mine

This step involves math, statistics, and data mining. The data in this case receives only a simple treatment: the program must figure out the minimum and maximum values for latitude and longitude by running through the data (as shown in Figure 1-3) so that it can be presented on a screen at a proper scale. Most of the time, this step will be far more complicated than a pair of simple math operations.

Represent

This step determines the basic form that a set of data will take. Some data sets are shown as lists, others are structured like trees, and so forth. In this case, each zip code has a latitude and longitude, so the codes can be mapped as a two-dimensional plot, with the minimum and maximum values for the latitude and longitude used for the start and end of the scale in each dimension. This is illustrated in Figure 1-4.

The Represent stage is a linchpin that informs the single most important decision in a visualization project and can make you rethink earlier stages. How you choose to represent the data can influence the very first step (what data you acquire) and the third step (what particular pieces you extract).



00210	43.005895	-71.013202	PORTSMOUTH	NH
00211	43.005895	-71.013202	PORTSMOUTH	NH
00212	43.005895	-71.013202	PORTSMOUTH	NH
00213	43.005895	-71.013202	PORTSMOUTH	NH
00214	43.005895	-71.013202	PORTSMOUTH	NH
00215	43.005895	-71.013202	PORTSMOUTH	NH
00501	40.922326	-72.637078	HOLTSVILLE	NY
00544	40.922326	-72.637078	HOLTSVILLE	NY
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
	min 24.655691 max 48.987385	min -124.62608 max -67.040764		

Figure 1-3. Mining the data: just compare values to find the minimum and maximum

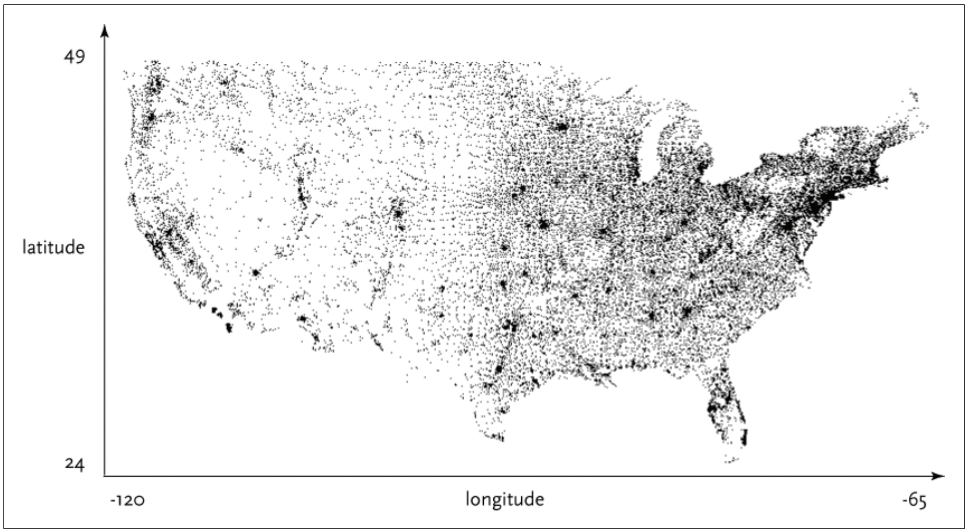


Figure 1-4. Basic visual representation of zip code data

Refine

In this step, graphic design methods are used to further clarify the representation by calling more attention to particular data (establishing hierarchy) or by changing attributes (such as color) that contribute to readability.

Hierarchy is established in Figure 1-5, for instance, by coloring the background deep gray and displaying the selected points (all codes beginning with four) in white and the deselected points in medium yellow.

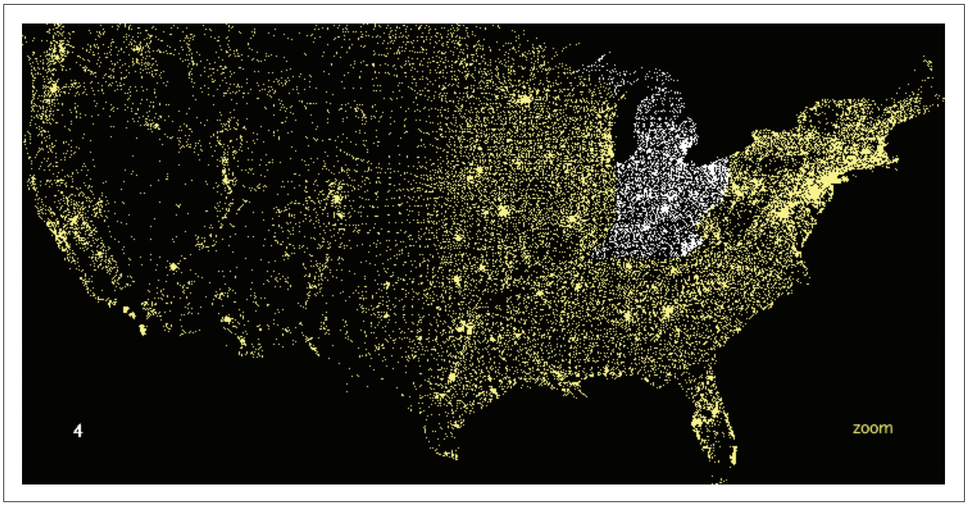


Figure 1-5. Using color to refine the representation

Interact

The next stage of the process adds interaction, letting the user control or explore the data. Interaction might cover things like selecting a subset of the data or changing the viewpoint. As another example of a stage affecting an earlier part of the process, this stage can also affect the refinement step, as a change in viewpoint might require the data to be designed differently.

In the Zipdecode project, typing a number selects all zip codes that begin with that number. Figures 1-6 and 1-7 show all the zip codes beginning with zero and nine, respectively.

Another enhancement to user interaction (not shown here) enables the users to traverse the display laterally and run through several of the prefixes. After typing part or all of a zip code, holding down the Shift key allows users to replace the last number typed without having to hit the Delete key to back up.

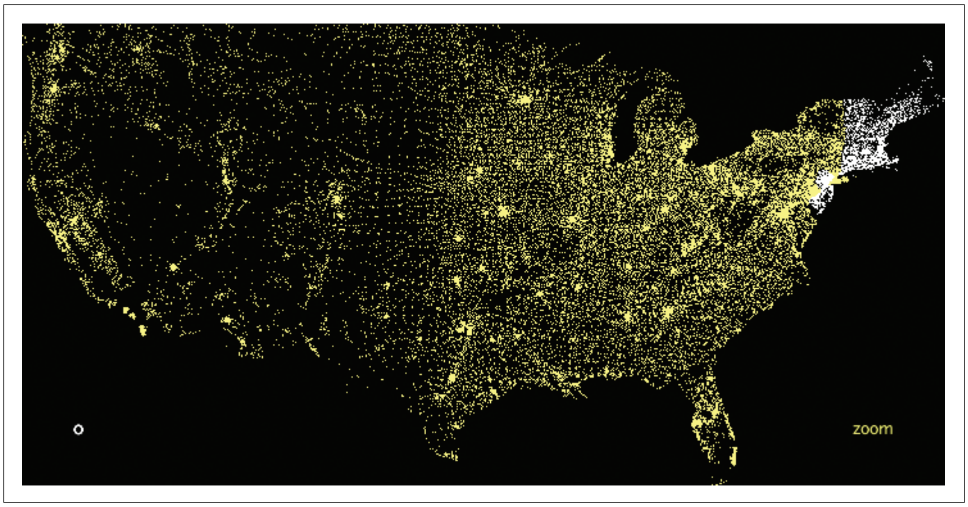


Figure 1-6. The user can alter the display through choices (zip codes starting with 0)

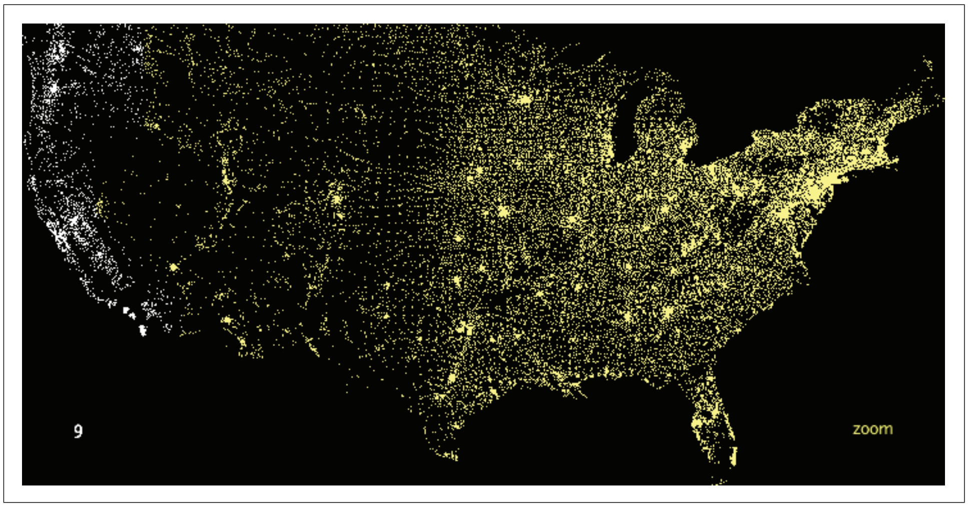


Figure 1-7. The user can alter the display through choices (zip codes starting with 9)

Typing is a very simple form of interaction, but it allows the user to rapidly gain an understanding of the zip code system's layout. Just contrast this sample application with the difficulty of deducing the same information from a table of zip codes and city names.

The viewer can continue to type digits to see the area covered by each subsequent set of prefixes. Figure 1-8 shows the region highlighted by the two digits 02, Figure 1-9 shows the three digits 021, and Figure 1-10 shows the four digits 0213. Finally, Figure 1-11 shows what you get by entering a full zip code, 02139—a city name pops up on the display.

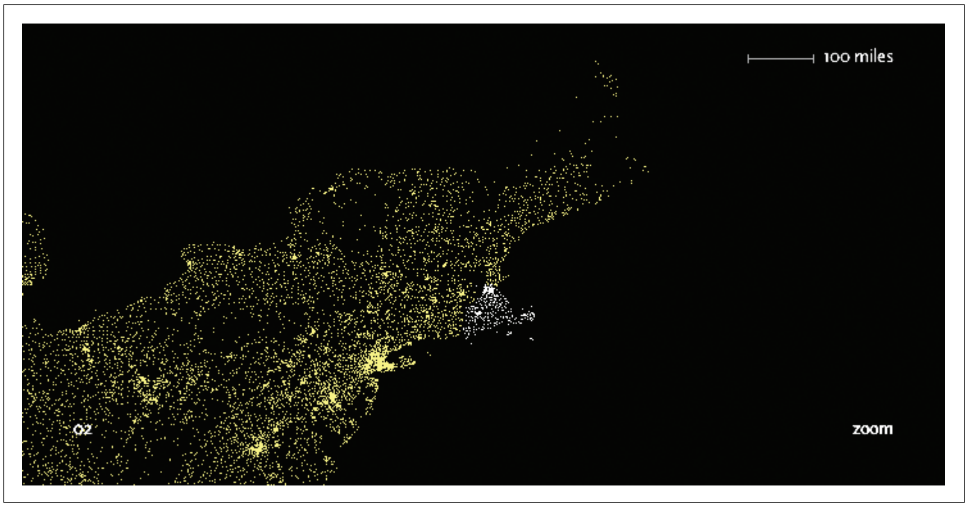


Figure 1-8. Honing in with two digits (02)

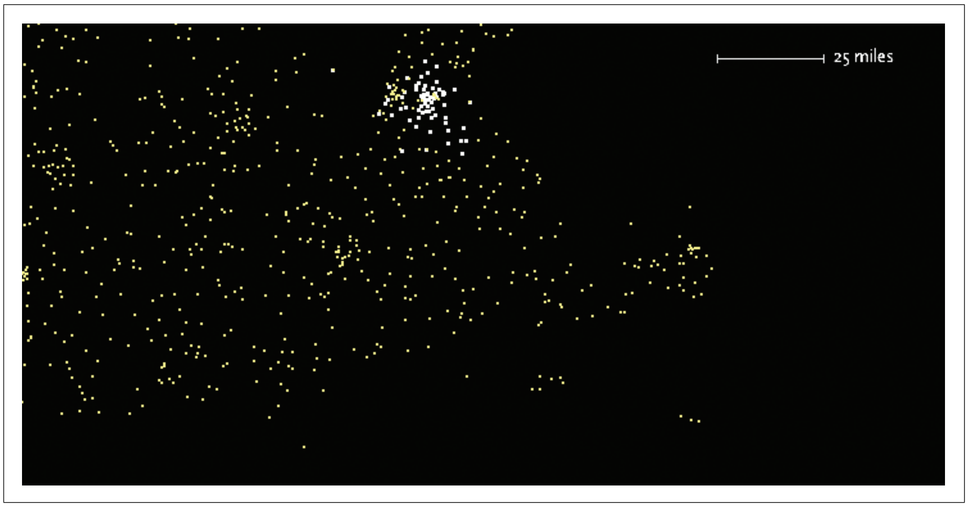


Figure 1-9. Honing in with three digits (021)

In addition, users can enable a "zoom" feature that draws them closer to each subsequent digit, revealing more detail around the area and showing a constant rate of detail at each level. Because we've chosen a map as a representation, we could add more details of state and county boundaries or other geographic features to help viewers associate the "data" space of zip code points with what they know about the local environment.

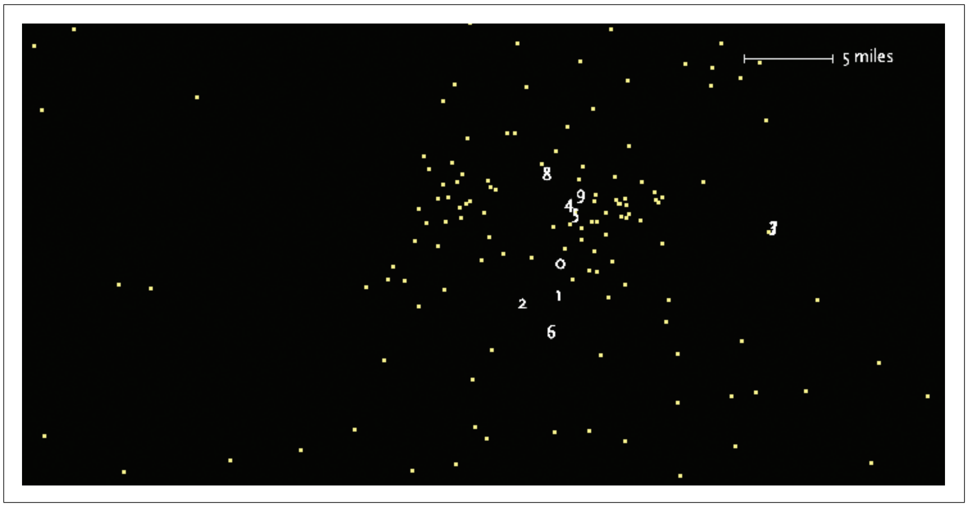


Figure 1-10. Honing in further with four digits (0213)

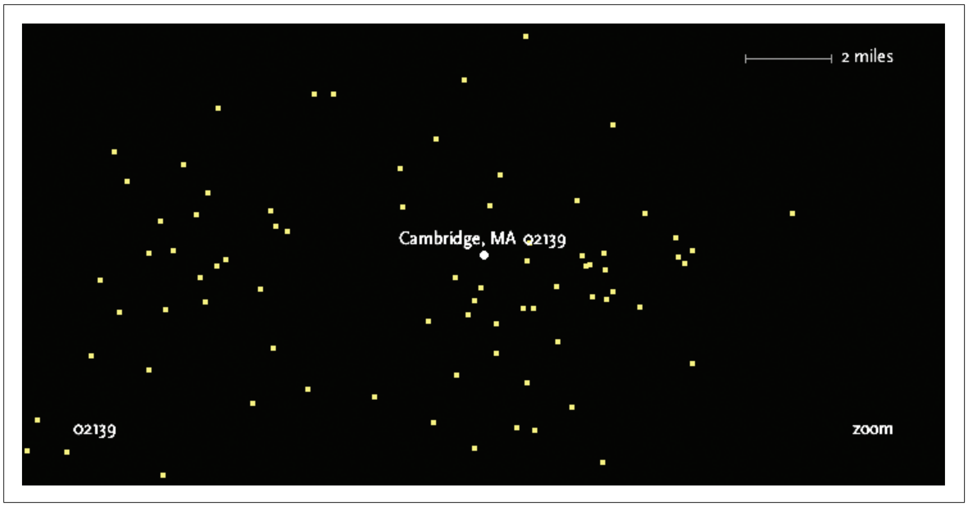


Figure 1-11. Honing in even further with the full zip code (02139)

Iteration and Combination

Figure 1-12 shows the stages in order and demonstrates how later decisions commonly reflect on earlier stages. Each step of the process is inextricably linked because of how the steps affect one another. In the Zipdecode application, for instance:

- The need for a compact representation on the screen led me to refilter the data to include only the contiguous 48 states.
- The representation step affected acquisition because after I developed the application I modified it so it could show data that was downloaded over a slow

Internet connection to the browser. My change to the structure of the data allows the points to appear slowly, as they are first read from the data file, employing the data itself as a "progress bar."

- Interaction by typing successive numbers meant that the colors had to be modified in the visual refinement step to show a slow transition as points in the display are added or removed. This helps the user maintain context by preventing the updates on-screen from being too jarring.

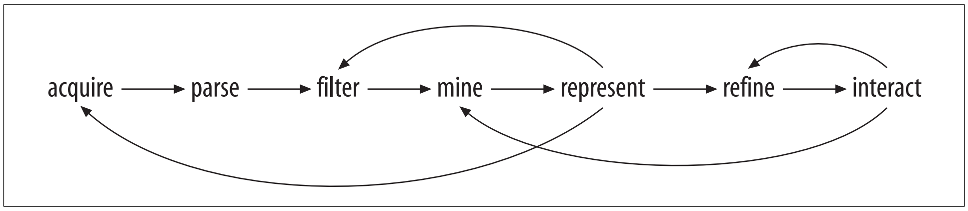


Figure 1-12. Interactions between the seven stages

The connections between the steps in the process illustrate the importance of the individual or team in addressing the project as a whole. This runs counter to the common fondness for assembly-line style projects, where programmers handle the technical portions, such as acquiring and parsing data, and visual designers are left to choose colors and typefaces. At the intersection of these fields is a more interesting set of properties that demonstrates their strength in combination.

When acquiring data, consider how it can change, whether sporadically (such as once a month) or continuously. This expands the notion of graphic design that's traditionally focused on solving a specific problem for a specific data set, and instead considers the meta-problem of how to handle a certain *kind* of data that might be updated in the future.

In the filtering step, data can be filtered in real time, as in the Zipdecode application. During visual refinement, changes to the design can be applied across the entire system. For instance, a color change can be automatically applied to the thousands of elements that require it, rather having to make such a tedious modification by hand. This is the strength of a computational approach, where tedious processes are minimized through automation.

Principles

I'll finish this general introduction to visualization by laying out some ways of thinking about data and its representation that have served me well over many years and many diverse projects. They may seem abstract at first, or of minor importance to the job you're facing, but I urge you to return and reread them as you practice visualization; they just may help you in later tasks.

Each Project Has Unique Requirements

A visualization should convey the unique properties of the data set it represents. This book is not concerned with providing a handful of ready-made "visualizations" that can be plugged into any data set. Ready-made visualizations can help produce a quick view of your data set, but they're inflexible commodity items that can be implemented in packaged software. Any bar chart or scatter plot made with Excel will look like a bar chart or scatter plot made with Excel. Packaged solutions can provide only packaged answers, like a pull-string toy that is limited to a handful of canned phrases, such as "Sales show a slight increase in each of the last five years!" Every problem is unique, so capitalize on that uniqueness to solve the problem.

Chapters in this book are divided by types of data, rather than types of display. In other words, we're not saying, "Here's how to make a bar graph," but "Here are several ways to show a correlation." This gives you a more powerful way to think about maximizing what can be said about the data set in question.

I'm often asked for a library of tools that will automatically make attractive representations of any given data set. But if each data set is different, the point of visualization is to expose that fascinating aspect of the data and make it self-evident. Although readily available representation toolkits are useful starting points, they must be customized during an in-depth study of the task.

Data is often stored in a generic format. For instance, databases used for annotation of genomic data might consist of enormous lists of start and stop positions, but those lists vary in importance depending on the situation in which they're being used. We don't view books as long abstract sequences of words, yet when it comes to information, we're often so taken with the enormity of the information and the low-level abstractions used to store it that the narrative is lost. Unless you stop thinking about databases, everything looks like a table—millions of rows and columns to be stored, queried, and viewed.

In this book, we use a small collection of simple helper classes as starting points. Often, we'll be targeting the Web as a delivery platform, so the classes are designed to take up minimal time for download and display. But I will also discuss more robust versions of similar tools that can be used for more in-depth work.

This book aims to help you learn to understand data as a tool for human decision-making—how it varies, how it can be used, and how to find what's unique about your data set. We'll cover many standard methods of visualization and give you the background necessary for making a decision about what sort of representation is suitable for your data. For each representation, we consider its positive and negative points and focus on customizing it so that it's best suited to what you're trying to convey about your data set.

Avoid the All-You-Can-Eat Buffet

Often, less detail will actually convey more information because the inclusion of overly specific details causes the viewer to miss what's most important or disregard the image entirely because it's too complex. Use as little data as possible, no matter how precious it seems.

Consider a weather map, with curved bands of temperatures across the country. The designers avoid giving each band a detailed edge (particularly because the data is often fuzzy). Instead, they convey a broader pattern in the data.

Subway maps leave out the details of surface roads because the additional detail adds more complexity to the map than necessary. Before maps were created in Beck's style, it seemed that knowing street locations was essential to navigating the subway. Instead, individual stations are used as waypoints for direction finding. The important detail is that your target destination is near a particular station. Directions can be given in terms of the last few turns to be taken after you exit the station, or you can consult a map posted at the station that describes the immediate area aboveground.

It's easy to collect data, and some people become preoccupied with simply accumulating more complex data or data in mass quantities. But more data is not implicitly better, and often serves to confuse the situation. Just because it can be measured doesn't mean it should. Perhaps making things simple is worth bragging about, but making complex messes is not. Find the smallest amount of data that can still convey something meaningful about the contents of the data set. As with Beck's underground map, focusing on the question helps define those minimum requirements.

The same holds for the many "dimensions" that are found in data sets. Web site traffic statistics have many dimensions: IP address, date, time of day, page visited, previous page visited, result code, browser, machine type, and so on. While each of these might be examined in turn, they relate to distinct questions. Only a few of the variables are required to answer a typical question, such as "How many people visited page *x* over the last three months, and how has that figure changed each month?" Avoid trying to show a burdensome multidimensional space that maps too many points of information.

Know Your Audience

Finally, who is your audience? What are their goals when approaching a visualization? What do they stand to learn? Unless it's accessible to your audience, why are you doing it? Making things simple and clear doesn't mean assuming that your users are idiots and "dumbing down" the interface for them.

In what way will your audience use the piece? A mapping application used on a mobile device has to be designed with a completely different set of criteria than one used on a desktop computer. Although both applications use maps, they have little to do with each other. The focus of the desktop application may be finding locations and print maps, whereas the focus of the mobile version is actively following the directions to a particular location.

Onward

In this chapter, we covered the process for attacking the common modern problems of having too much data and having data that changes. In the next chapter, we'll discuss Processing, the software tool used to handle data sets in this book.

CHAPTER 2

Getting Started with Processing

The Processing project began in the spring of 2001 and was first used at a workshop in Japan that August. Originally built as a domain-specific extension to Java targeted at artists and designers, Processing has evolved into a full-blown design and prototyping tool used for large-scale installation work, motion graphics, and complex data visualization. Processing is a simple programming environment that was created to make it easier to develop visually oriented applications with an emphasis on animation and provide users with instant feedback through interaction. As its capabilities have expanded over the past six years, Processing has come to be used for more advanced production-level work in addition to its sketching role.

Processing is based on Java, but because program elements in Processing are fairly simple, you can learn to use it from this book even if you don't know any Java. If you're familiar with Java, it's best to forget that Processing has anything to do with it for a while, at least until you get the hang of how the API works. We'll cover how to integrate Java and Processing toward the end of the book.

The latest version of Processing can be downloaded at:

http://processing.org/download

An important goal for the project was to make this type of programming accessible to a wider audience. For this reason, Processing is free to download, free to use, and open source. But projects developed using the Processing environment and core libraries can be used for any purpose. This model is identical to GCC, the GNU Compiler Collection. GCC and its associated libraries (e.g., libc) are open source under the GNU Public License (GPL), which stipulates that changes to the code must be made available. However, programs created with GCC (examples too numerous to mention) are not themselves required to be open source.

Processing consists of:

- The Processing Development Environment (PDE). This is the software that runs when you double-click the Processing icon. The PDE is an Integrated Development Environment with a minimalist set of features designed as a simple introduction to programming or for testing one-off ideas.
- A collection of commands (also referred to as functions or methods) that make up the "core" programming interface, or API, as well as several libraries that support more advanced features, such as drawing with OpenGL, reading XML files, and saving complex imagery in PDF format.
- A language syntax, identical to Java but with a few modifications. The changes are laid out in detail toward the end of the book.
- An active online community, hosted at *http://processing.org*.

For this reason, references to "Processing" can be somewhat ambiguous. Are we talking about the API, the development environment, or the web site? I'll be careful to differentiate them when referring to each.

Sketching with Processing

A Processing program is called a *sketch*. The idea is to make Java-style programming feel more like scripting, and adopt the process of scripting to quickly write code. Sketches are stored in the *sketchbook*, a folder that's used as the default location for saving all of your projects. When you run Processing, the sketch last used will automatically open. If this is the first time Processing is used (or if the sketch is no longer available), a new sketch will open.

Sketches that are stored in the sketchbook can be accessed from File → Sketchbook. Alternatively, File → Open... can be used to open a sketch from elsewhere on the system.

Advanced programmers need not use the PDE and may instead use its libraries with the Java environment of choice. (This is covered toward the end of the book.) However, if you're just getting started, it's recommended that you use the PDE for your first few projects to gain familiarity with the way things are done. Although Processing is based on Java, it was never meant to be a Java IDE with training wheels. To better address our target audience, its conceptual model (how programs work, how interfaces are built, and how files are handled) is somewhat different from Java's.

Hello World

Programming languages are often introduced with a simple program that prints "Hello World" to the console. The Processing equivalent is simply to draw a line:

```
line(15, 25, 70, 90);
```

Enter this example and press the Run button, which is an icon that looks like the Play button on any audio or video device. The result will appear in a new window, with a gray background and a black line from coordinate (15, 25) to (70, 90). The (0, 0) coordinate is the upper-lefthand corner of the display window. Building on this program to change the size of the display window and set the background color, type in the code from Example 2-1.

Example 2-1. Simple sketch

```
size(400, 400);
background(192, 64, 0);
stroke(255);
line(150, 25, 270, 350);
```

This version sets the window size to 400×400 pixels, sets the background to an orange-red, and draws the line in white, by setting the stroke color to 255. By default, colors are specified in the range 0 to 255. Other variations of the parameters to the `stroke()` function provide alternate results:

```
stroke(255);               // sets the stroke color to white
stroke(255, 255, 255);     // identical to stroke(255)
stroke(255, 128, 0);       // bright orange (red 255, green 128, blue 0)
stroke(#FF8000);           // bright orange as a web color
stroke(255, 128, 0, 128); // bright orange with 50% transparency
```

The same alternatives work for the `fill()` command, which sets the fill color, and the `background()` command, which clears the display window. Like all Processing methods that affect drawing properties, the fill and stroke colors affect all geometry drawn to the screen until the next fill and stroke commands are executed.

It's also possible to use the editor of your choice instead of the built-in editor. Simply select "Use External Editor" in the Preferences window (Processing → Preferences on Mac OS X, or File → Preferences on Windows and Linux). When using an external editor, editing will be disabled in the PDE, but the text will reload whenever you press Run.

Hello Mouse

A program written as a list of statements (like the previous examples) is called a *basic* mode sketch. In basic mode, a series of commands are used to perform tasks or create a single image without any animation or interaction. Interactive programs are drawn as a series of frames, which you can create by adding functions titled `setup()` and `draw()`, as shown in the *continuous* mode sketch in Example 2-2. They are built-in functions that are called automatically.

Example 2-2. Simple continuous mode sketch

```
void setup( ) {
  size(400, 400);
  stroke(255);
  background(192, 64, 0);
}

void draw( ) {
  line(150, 25, mouseX, mouseY);
}
```

Example 2-2 is identical in function to Example 2-1, except that now the line follows the mouse. The `setup( )` block runs once, and the `draw( )` block runs repeatedly. As such, `setup( )` can be used for any initialization; in this case, it's used for setting the screen size, making the background orange, and setting the stroke color to white. The `draw( )` block is used to handle animation. The `size( )` command must always be the first line inside `setup( )`.

Because the `background( )` command is used only once, the screen will fill with lines as the mouse is moved. To draw just a single line that follows the mouse, move the `background( )` command to the `draw( )` function, which will clear the display window (filling it with orange) each time `draw( )` runs:

```
void setup( ) {
  size(400, 400);
  stroke(255);
}

void draw( ) {
  background(192, 64, 0);
  line(150, 25, mouseX, mouseY);
}
```

Basic mode programs are most commonly used for extremely simple examples, or for scripts that run in a linear fashion and then exit. For instance, a basic mode program might start, draw a page to a PDF file, and then exit.

Most programs employ continuous mode, which uses the `setup( )` and `draw( )` blocks. More advanced mouse handling can also be introduced; for instance, the `mousePressed( )` method will be called whenever the mouse is pressed. So, in the following example, when the mouse is pressed, the screen is cleared via the `background( )` command:

```
void setup( ) {
  size(400, 400);
  stroke(255);
}

void draw( ) {
  line(150, 25, mouseX, mouseY);
}
```

```
void mousePressed( ) {
  background(192, 64, 0);
}
```

More about basic versus continuous mode programs can be found in the Programming Modes section of the Processing reference, which can be viewed from Help → Getting Started or online at *http://processing.org/reference/environment*.

Exporting and Distributing Your Work

One of the most significant features of the Processing environment is its ability to bundle your sketch into an applet or application with just one click. Select File → Export to package your current sketch as an applet. This will create a folder named *applet* inside your sketch folder. Opening the *index.html* file inside that folder will open your sketch in a browser. The applet folder can be copied to a web site intact and will be viewable by users who have Java installed on their systems. Similarly, you can use File → Export Application to bundle your sketch as an application for Windows, Mac OS X, and Linux.

The applet and application folders are overwritten whenever you export—make a copy or remove them from the sketch folder before making changes to the *index.html* file or the contents of the folder.

More about the export features can be found in the reference; see *http://processing.org/reference/environment/export.html*.

Saving Your Work

If you don't want to distribute the actual project, you might want to create images of its output instead. Images are saved with the `saveFrame( )` function. Adding `saveFrame( )` at the end of `draw( )` will produce a numbered sequence of TIFF-format images of the program's output, named *screen-0001.tif*, *screen-0002.tif*, and so on. A new file will be saved each time `draw( )` runs. Watch out because this can quickly fill your sketch folder with hundreds of files. You can also specify your own name and file type for the file to be saved with a command like:

```
saveFrame("output.png")
```

To do the same for a numbered sequence, use #s (hash marks) where the numbers should be placed:

```
saveFrame("output-####.png");
```

For high-quality output, you can write geometry to PDF files instead of the screen, as described in the section "More About the size( ) Method," later in this chapter.

Examples and Reference

While many programmers learn to code in school, others teach themselves. Learning on your own involves looking at lots of other code: running, altering, breaking, and enhancing it until you can reshape it into something new. With this learning model in mind, the Processing software download includes dozens of examples that demonstrate different features of the environment and API.

The examples can be accessed from the File → Examples menu. They're grouped into categories based on their functions (such as Motion, Typography, and Image) or the libraries they use (such as PDF, Network, and Video).

Find an interesting topic in the list and try an example. You'll see commands that are familiar, such as `stroke()`, `line()`, and `background()`, as well as others that have not yet been covered. To see how a function works, select its name, and then right-click and choose Find in Reference from the pop-up menu (Find in Reference can also be found beneath the Help menu). That will open the reference for that function in your default web browser.

In addition to a description of the function's syntax, each reference page includes an example that uses the function. The reference examples are much shorter (usually four or five lines apiece) and easier to follow than the longer code examples.

More About the size( ) Method

The `size()` command also sets the global variables `width` and `height`. For objects whose size is dependent on the screen, always use the `width` and `height` variables instead of a number (this prevents problems when the `size()` line is altered):

```
size(400, 400);

// The wrong way to specify the middle of the screen
ellipse(200, 200, 50, 50);

// Always the middle, no matter how the size() line changes
ellipse(width/2, height/2, 50, 50);
```

In the earlier examples, the `size()` command specified only a width and height for the new window. An optional parameter to the `size()` method specifies how graphics are rendered. A *renderer* handles how the Processing API is implemented for a particular output method (whether the screen, or a screen driven by a high-end graphics card, or a PDF file). Several renderers are included with Processing, and each has a unique function. At the risk of getting too far into the specifics, here are examples of how to specify them with the `size()` command along with descriptions of their capabilities.

`size(400, 400, JAVA2D);`
: The Java2D renderer is used by default, so this statement is identical to `size(400, 400)`. The Java2D renderer does an excellent job with high-quality 2D vector graphics, but at the expense of speed. In particular, working with pixels is slower compared to the P2D and P3D renderers.

`size(400, 400, P2D);`
: The Processing 2D renderer is intended for simpler graphics and fast pixel operations. It lacks niceties such as stroke caps and joins on thick lines, but makes up for it when you need to draw thousands of simple shapes or directly manipulate the pixels of an image or video.

`size(400, 400, P3D);`
: Similar to P2D, the Processing 3D renderer is intended for speed and pixel operations. It also produces 3D graphics inside a web browser, even without the use of a library like Java3D. Image quality is poorer (the `smooth( )` command is disabled, and image accuracy is low), but you can draw thousands of triangles very quickly.

`size(400, 400, OPENGL);`
: The OpenGL renderer uses Sun's Java for OpenGL (JOGL) library for faster rendering, while retaining Processing's simpler graphics APIs and the PDE's easy applet and application export. To use OpenGL graphics, you must select Sketch → Import Library → OpenGL in addition to altering your `size( )` command. OpenGL applets also run within a web browser without additional modification, but a dialog box will appear asking users whether they trust "Sun Microsystems, Inc." to run Java for OpenGL on their computers. If this poses a problem, the P3D renderer is a simpler, if less full-featured, solution.

`size(400, 400, PDF, "output.pdf");`
: The PDF renderer draws all geometry to a file instead of the screen. Like the OpenGL library, you must import the PDF library before using this renderer. This is a cousin of the Java2D renderer, but instead writes directly to PDF files.

Each renderer has a specific role. P2D and P3D are great for pixel-based work, while the JAVA2D and PDF settings will give you the highest quality 2D graphics. When the Processing project first began, the P2D and P3D renderers were a single choice (and, in fact, the only available renderer). This was an attempt to offer a unified mode of thinking about drawing, whether in two or three dimensions. However, this became too burdensome because of the number of tradeoffs that must be made between 2D and 3D. A very different expectation of quality exists for 2D and 3D, for instance, and trying to cover both sides in one renderer meant doing both poorly.

Loading and Displaying Data

One of the unique aspects of the Processing API is the way files are handled. The `loadImage()` and `loadStrings()` functions each expect to find a file inside a folder named *data*, which is a subdirectory of the *sketch* folder.

The data Folder

The *data* folder addresses a common frustration when dealing with code that is tested locally but deployed over the Web. Like Java, software written with Processing is subject to security restrictions that determine how a program can access resources such as the local hard disk or other servers via the Internet. This prevents malicious developers from writing code that could harm your computer or compromise your data.

The security restrictions can be tricky to work with during development. When running a program locally, data can be read directly from the disk, though it must be placed relative to the user's "working directory," generally the location of the application. When running online, data must come from a location on the same server. It might be bundled with the code itself (in a JAR archive, discussed later, or from another URL on the same server). For a local file, Java's `FileInputStream` class can be used. If the file is bundled in a JAR archive, the `getResource()` function is used. For a file on the server, `URL.openStream()` might be employed. During the journey from development to deployment, it may be necessary to use all three of these methods.

With Processing, these scenarios (and some others) are handled transparently by the file API methods. By placing resources in the *data* folder, Processing packages the files as necessary for online and offline use.

File handling functions include `loadStrings()`, which reads a text file into an array of `String` objects, and `loadImage()`, which reads an image into a `PImage` object, the container for image data in Processing.

```
// Examples of loading a text file and a JPEG image
// from the data folder of a sketch.
String[] lines = loadStrings("something.txt");
PImage image = loadImage("picture.jpg");
```

These examples may be a bit easier to read if you know the programming concepts of data types and classes. Each variable has to have a data type, such as `String` or `PImage`.

The `String[]` syntax means "an array of data of the class `String`." This array is created by the `loadStrings` command and is given the name `lines`; it will presumably be used later in the program under that name. The reason `loadStrings` creates an array is that it splits the *something.txt* file into its individual lines. The second command creates a single variable of class `PImage`, with the name `image`.

To add a file to a Processing sketch, use the Sketch → Add File command, or drag the file into the editor window of the PDE. The *data* folder will be created if it does not exist already.

To view the contents of the *sketch* folder, use the Sketch → Show Sketch Folder command. This opens the sketch window in your operating system's file browser.

In the file commands, it's also possible to use full path names to local files, or URLs to other locations if the *data* folder is not suitable:

```
// Load a text file and an image from the specified URLs
String[] lines = loadStrings("http://benfry.com/writing/map/locations.tsv");
PImage image = loadImage("http://benfry.com/writing/map/map.png");
```

Functions

The steps of the process outlined in the first chapter are commonly associated with specific functions in the Processing API. For instance:

Acquire
: `loadStrings( )`, `loadBytes( )`

Parse
: `split( )`

Filter
: `for( )`, `if (item[`*i*`].startsWith( ))`

Mine
: `min( )`, `max( )`, `abs( )`

Represent
: `map( )`, `beginShape( )`, `endShape( )`

Refine
: `fill( )`, `strokeWeight( )`, `smooth( )`

Interact
: `mouseMoved( )`, `mouseDragged( )`, `keyPressed( )`

This is not an exhaustive list, but simply another way to frame the stages of visualization for those more familiar with code.

Libraries Add New Features

A *library* is a collection of code in a specified format that makes it easy to use within Processing. Libraries have been important to the growth of the project because they let developers make new features accessible to users without making them part of the core Processing API.

Several core libraries come with Processing. These can be seen in the Libraries section of the online reference (also available from the Help menu from within the PDE); see *http://processing.org/reference/libraries*.

One example is the XML import library. This is an extremely minimal XML parser (based on the open source project NanoXML) with a small download footprint (approximately 30KB) that makes it ideal for online use.

To use the XML library in a project, choose Sketch → Import Library → xml. This will add the following line to the top of the sketch:

```
import processing.xml.*;
```

Java programmers will recognize the `import` command. In Processing, this line also determines what code is packaged with a sketch when it is exported as an applet or application.

Now that the XML library is imported, you can issue commands from it. For instance, the following line loads an XML file named *sites.xml* into a variable named `xml`:

```
XMLElement xml = new XMLElement(this, "sites.xml");
```

The `xml` variable can now be manipulated as necessary to read the contents. The full example can be seen in the reference for its class, `XMLElement`, at *http://processing.org/reference/libraries/xml/XMLElement.html*.

The `this` variable is used frequently with library objects because it lets the library make use of the core API functions to draw to the screen or load files. The latter case applies to the XML library, allowing XML files to be read from the *data* folder or other locations supported by the file API methods.

Other libraries provide features such as writing QuickTime movie files, sending and receiving MIDI commands, sophisticated 3D camera control, and access to MySQL databases.

Sketching and Scripting

Processing sketches are made up of one or more tabs, with each tab representing a piece of code. The environment is designed around projects that are a few pages of code, and often three to five tabs in total. This covers a significant number of projects developed to test and prototype ideas, often before embedding them into a larger project or building a more robust application for broader deployment.

This small-scale development style is useful for data visualization in two primary scenarios. The most common scenario is when you have a data set in mind, or a question that you're trying to answer, and you need a quick way to load the data, represent it, and see what's there. This is important because it lets you take an inventory of the data in question. How many elements are there? What are the largest and smallest values? How many dimensions are we looking at? We'll return to this notion of exploring data in future chapters.

In the second scenario, the desired outcome is known, but the correct means of representing the data and interacting with it have not yet been determined.

The idea of sketching is identical to that of scripting, except that you're not working in an interpreted scripting language, but rather gaining the performance benefit of compiling to Java class files. Of course, strictly speaking, Java itself is an interpreted language, but its bytecode compilation brings it much closer to the "metal" than languages such as JavaScript, ActionScript, Python, or Ruby.

Processing was never intended as the ultimate language for visual programming; instead, we set out to make something that was:

- A sketchbook for our own work, simplifying the majority of tasks that we undertake
- A programming environment suitable for teaching programming to a non-traditional audience
- A stepping stone from scripting languages to more complicated or difficult languages such as full-blown Java or C++

At the intersection of these points is a tradeoff between speed and simplicity of use. If we didn't care about speed, it might make sense to use Python, Ruby, or many other scripting languages. That is especially true for the education side. If we didn't care about making a transition to more advanced languages, we'd probably avoid a C++ or Java-style syntax. But Java is a nice starting point for a sketching language because it's far more forgiving than C/C++ and also allows users to export sketches for distribution via the Web.

Processing assembles our experience in building software of this kind (sketches of interactive works or data-driven visualization) and simplifies the parts that we felt should be easier, such as getting started quickly, and insulates new users from issues like those associated with setting up Java.

Don't Start by Trying to Build a Cathedral

If you're already familiar with programming, it's important to understand how Processing differs from other development environments and languages. The Processing project encourages a style of work that builds code quickly, understanding that either the code will be used as a quick sketch or that ideas are being tested before developing a final project. This could be misconstrued as software engineering heresy. Perhaps we're not far from "hacking," but this is more appropriate for the roles in which Processing is used. Why force students or casual programmers to learn about graphics contexts, threading, and event handling methods before they can show something on the screen that interacts with the mouse? The same goes for advanced developers; why should they always need to start with the same two pages of code whenever they begin a project?

In another scenario, if you're doing scientific visualization, the ability to try things out quickly is a far higher priority than sophisticated code structure. Usually you don't know what the outcome will be, so you might build something one week to try an initial hypothesis and build something new the next based on what was learned in the first week. To this end, remember the following considerations as you begin visualizing data with Processing:

- Be careful about creating unnecessary structures in your code. As you learn about encapsulating your code into classes, it's tempting to make ever-smaller classes because data can always be distilled further. Do you need classes at the level of molecules, atoms, or quarks? Just because atoms go smaller doesn't mean that we need to work at a lower level of abstraction. If a class is half a page long, does it make sense to have six additional subclasses that are each half a page long? Could the same thing be accomplished with a single class that is a page and a half in total?
- Consider the scale of the project. It's not always necessary to build enterprise-level software on the first day. We're asking questions about data, so figure out the minimum code necessary to help answer that question.
- Do you really need to use a database? If you're manipulating half a gigabyte of data and have a gigabyte of RAM, can you shove the data into memory and play with it directly? If so, use that option; it lets you avoid developing a schema for the database before you actually know what you're doing (or want to do) with the data.
- Do you need to start with *all* the data? Having collected precious terabytes of potentially useful information, do you need all of it to answer your first round of questions? A small percentage, which will require less infrastructure, is usually enough to indicate whether a larger project is even worth pursuing.

The point is to delay engineering work until it's appropriate. The threshold for where to begin engineering a piece of visualization software is much later than for traditional programming projects because there is a kind of "art" to the early process of quick iteration.

Of course, once things are working, avoid the urge to rewrite for its own sake. A rewrite should be used when addressing a completely different problem. If you've managed to hit the nail on the head, you should refactor to clean up method names and class interactions. But a full rewrite of already finished code is almost always a bad idea, no matter how "ugly" it seems.

Ready?

In this chapter, we covered the basics of the Processing environment, as well as a bit of the philosophy behind the environment itself and the type of software built with the language. In the next chapter, we'll get started representing our first data set.