CHAPTER 6
Scatterplot Maps

In this chapter, we cover the seven steps as laid out in Chapter 1 and apply them to the question, "How do zip codes relate to geography?" (The background for this project was introduced in Chapter 1.)

Preprocessing

Data is always dirty, and once you've found your data set, you'll need to clean it up. As in the previous chapter, we'll go through the steps of acquiring and parsing in detail. None of this is rocket science, but again, it's meant to familiarize you with the various formats in which you'll find data, and alert you to some of the common issues you'll encounter along the way. If you just want to start playing with locations and maps, you can download the finished *zips.tsv* file from the book web site (*http://benfry.com/writing/zipdecode/zips.tsv*) and jump ahead to the next section.

Data from the U.S. Census Bureau (Acquire)

The acronym ZIP stands for Zoning Improvement Plan, a 1963 initiative to simplify the delivery of mail in the United States. Personal correspondence, once the majority of all mail, was rapidly being overtaken by business mail, which by the 1960s accounted for 80% of the post. Faced with an ever-increasing amount of mail to process, the U.S. Postal Service initiated the zip system to specify more accurately the geographic area of the mail's destination. The U.S. Postal Service's web site features a lengthier history of the system at *http://www.usps.com/history*.

Versions of the zip code database are available from a variety of sources. The data is public and therefore freely available on government web sites. Government sites often contain a wealth of information for those willing to take the time to dig for it. The terminology can sometimes be archaic and the documentation poor, so it often takes a while to figure out exactly how things work. In the spirit of capitalism, resellers have jumped in to provide you with "value added" versions of the data.

Clearly, the term “value” varies widely—some companies are happy to charge your credit card for the honor of their knowing the right search terms to use with Google, or of having clicked through the census bureau web site for you. Others subscribe to the official data from the U.S. Postal Service and curate a useful, working copy of the data.

Free services have also emerged, such as *http://geocoder.us* (described online and in O’Reilly’s *Mapping Hacks* by Schuyler Erle, Rich Gibson, and Jo Walsh), which maintains a working data set as well as open source software that you can use to formulate zip code and address information.

For our purposes, we’ll use a listing from the U.S. Census Bureau, found at *http://www.census.gov/geo/www/tiger/zip1999.html*. The data is outdated by a few years, but it will be sufficient for our short-term purpose.

That page provides a link to a compressed archive (with the seemingly redundant title *zip1999.zip*) that contains a DBF file with the data set and a Microsoft Word document that describes each of the fields (columns) in the data set. (For information about DBF files, see Chapter 10.)

Both OpenOffice and Microsoft Excel can open a DBF file. OpenOffice might even register the *.dbf* extension explicitly, but in Excel you’ll have to use the “All files” option in the File → Open dialog box before the DBF file shows up.

The file contains approximately 42,000 lines, one for each zip code. The following is a small sample:

```
ZIP_CODE  LATITUDE      LONGITUDE   ZIP_CLASS  PONAME          STATE    COUNTY
95466    +39.056598    -123.525375             PHILO            06       045
95468    +38.919145    -123.540572             POINT ARENA      06       045
95469    +39.360935    -123.106751             POTTER VALLEY    06       045
95470    +39.302446    -123.462532             REDWOOD VALLEY   06       045
95471    +38.523472    -122.982142    P        RIO NIDO         06       097
95472    +38.407222    -122.869654             SEBASTOPOL       06       097
95473    +38.325851    -122.505846    P        SEBASTOPOL       06       097
95476    +38.255943    -122.476819             SONOMA           06       097
95480    +38.676694    -123.372059             STEWARTS POINT   06       097
95481    +39.127247    -123.164533    P        TALMAGE          06       045
95482    +39.403699    -123.321202             UKIAH            06       045
95485    +39.252489    -122.856430             UPPER LAKE       06       033
95486    +38.464487    -123.037996    P        VILLA GRANDE     06       097
95487    +38.463088    -122.989975    P        VINEBURG         06       097
95488    +39.660425    -123.786385             WESTPORT         06       045
95490    +39.525958    -123.365730             WILLITS          06       045
95492    +38.532827    -122.804100             WINDSOR          06       097
95493    +39.185033    -122.965163             WITTER SPRINGS   06       033
95494    +38.934552    -123.268378             YORKVILLE        06       045
95497    +38.717318    -123.463976    P        THE SEA RANCH    06       097
95501    +40.646324    -124.025773             EUREKA           06       023
```

Dealing with the Zip Code Database File (Parse and Filter)

After opening the file with OpenOffice or Excel, save the file as tab-delimited (TSV) or comma-separated (CSV) values for easier parsing. For our purposes, we'll save it as CSV (title it *zipnov99.csv*), resulting in a file whose first 10 lines look like:

```
"ZIP_CODE,C,5","LATITUDE,C,11","LONGITUDE,C,11","ZIP_
CLASS,C,1","PONAME,C,28","STATE,C,2","COUNTY,C,3"
"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"
```

The data in its current format is not quite ready to go. It is almost always the case that you'll need to do additional work to clean the data before it is ready to be included with an application. Often, you'll run the acquire stage and the parse and filter stages twice, as we do in this chapter. With our zip code data, one can observe that:

- The useful columns for our purposes are `ZIP_CODE`, `LATITUDE`, `LONGITUDE`, `PONAME`, and `STATE`. The `ZIP_CLASS` and `COUNTY` columns can be removed to save some disk space (if we intend to run this locally) or download time (if we distribute this application over the Web).
- The `STATE` column is encoded as a FIPS (Federal Information Processing Standards) number, which we'll want to convert to a two-digit state abbreviation.
- Not all of the data rows are necessary for our example. For the time being, we'll cheat and use only the contiguous 48 states, omitting Alaska, Hawaii, and American territories.
- While we're at it, the city names are listed in ALL CAPS, which looks garish and aggressive. It's important to realize that this is a limitation of the particular data set, not all data of this kind. If we were to get a better zip code list, the names might not be capitalized. As such, it's better to clean the data first, rather than include a workaround for the problem in the final project. That is not to say that we should be obsessed with generalization (see the discussion of "sketching" in Chapter 2), but this is a case where the generalization doesn't cost us anything in terms of time or efficiency.
- Excel and OpenOffice tend to introduce a lot of extra rubbish, such as unnecessary quotes, into TSV files. (To be fair, OpenOffice allows you to tweak these parameters, though the nomenclature for the export interface can be confusing.)
- Because latitude and longitude values reflect points on a globe, a *projection* will be used to convert the coordinates to positions that more closely resemble how the United States are typically portrayed (slightly curved at the top, rather than following a latitude line straight across).

- We'll need to know the range of latitudes and longitudes in order to plot them in a proper range to the screen. For this, we'll keep track of the minimum and maximum values after they've been projected.
- Because downloading the zip codes from the network may take some time, the file will also specify the number of total lines, so that we can calculate progress during the download.

Each of these issues is easy to handle. We'll simply write a short bit of code to turn our data into a more usable and compact format. Addressing each issue is straightforward:

1. We'll make a second version of the data file that leaves out the unnecessary columns. By placing constants at the beginning of the code for each of the columns, we'll make the code easier to follow because `arrayName[LONGITUDE]` is more self-explanatory than `arrayName[2]`.

   ```
   // Indices for each of the columns
   int ZIP_CODE = 0;
   int LATITUDE = 1;
   int LONGITUDE = 2;
   int ZIP_CLASS = 3;
   int PONAME = 4;
   int STATE_FIPS = 5;
   int COUNTY = 6;
   ```

2. In the second file, the FIPS code will be replaced with a two-letter state abbreviation. The codes can be found at the Federal Information Processing Standards site, specifically Publication 5-2: *http://www.itl.nist.gov/fipspubs/fip5-2.htm*.

   A clean version of this data is available from the book web site at *http://benfry.com/writing/zipdecode/fips.tsv*. For the curious, the clean version was created by saving the HTML file, opening it with OpenOffice, copying the data from the state tables, and pasting it into a blank spreadsheet file. The file was then saved as TSV by selecting the "Text CSV (.csv)" option, and in the "Export of text files" dialog box, setting the "Field delimiter" to "{Tab}" (which can be chosen from the drop-down list) and the "Text delimiter" to nothing.

   The following code loads the *fips.tsv* file and places it into a `Hashtable` so that we can look up individual values. With this in place, `fipsTable.get( )` will provide the state abbreviation for any FIPS code value.

   ```
   // Load the state FIPS codes into a table.
   Hashtable fipsTable = new Hashtable( );
   String[] fipsLines = loadStrings("fips.tsv");
   for (int i = 0; i < fipsLines.length; i++) {
     // Split each line on the tab characters.
     String[] pieces = split(fipsLines[i], TAB);
     // The FIPS code is in column 1,
     // and the state abbreviation in column 2
     // (keep in mind that columns are numbered from zero).
     fipsTable.put(pieces[1], pieces[2]);
   }
   ```

The clean version also omits the first 0 in each code to match the two-digit codes used in the *zipsnov99.csv* file. If each FIPS code were converted to an integer, this wouldn't be necessary, but storing this field as integer data is not worthwhile. It is tedious to convert (and later restore) the integer values, and not really necessary when `String` objects will work fine and save a few steps.

3. Gleaning the contiguous 48 states from the full list is a straightforward task. Other territories have been left out of *fips.tsv*, meaning that if no state abbreviation is found for a code, it can be skipped. Cutting out Alaska and Hawaii is also a matter of skipping lines whose FIPS code maps to AK or HI. Inside the `for( )` loop, the code will look like this:

```
    String stateAbbrev = (String) fipsTable.get(data[STATE_FIPS]);
    // If the abbreviation was not found, skip this line,
    // because that means it's an outlying territory.
    if (stateAbbrev == null) continue;
    // For now, skip Alaska and Hawaii.
    if (stateAbbrev.equals("AK") || stateAbbrev.equals("HI")) continue;
```

4. For the city names (the `PONAME` column), we can capitalize just the first letter of each word, which isn't perfect, but it's better than shouting all the time. Further, because the city and state abbreviation are always used together (e.g., "Sebastopol, CA"), that information can go into a single column. That also gives us more flexibility if we want to use a different data set, such as the postal codes for Germany or Australia—which don't specify locations the same way as the U.S. but have similar numbering systems.

   The following code is a general-purpose method that takes a `String` as input, breaks it into individual characters, and then capitalizes the characters that follow spaces (while making all other characters lowercase).

```
// Capitalize the first letter of each word in a string.
String fixCapitals(String title) {
  char[] text = title.toCharArray( );
  // If set to true, the next letter will be capitalized.
  boolean capitalizeNext = true;

  for (int i = 0; i < text.length; i++) {
    if (Character.isSpace(text[i])) {
      capitalizeNext = true;
    } else if (capitalizeNext) {
      text[i] = Character.toUpperCase(text[i]);
      capitalizeNext = false;
    } else {
      text[i] = Character.toLowerCase(text[i]);
    }
  }
  return new String(text);
}
```

If you are dealing with an enormous amount of data and know for a fact that your data is simply ASCII, other tricks can be used to capitalize more quickly than the `Character.toUpperCase( )` and `Character.toLowerCase( )` functions (which take into account Unicode capitalization).

5. Extraneous quotes and commas can be thrown out, converting the information to a more minimal TSV file. More background regarding CSV and TSV files (including an explanation for the quotes and commas) can be found in Chapter 10. This conversion is handled with a `scrubQuotes( )` function:

```
// Parse quotes from CSV data. Quotes around a column are common,
// and actual double quotes (") are specified by two double quotes ("").
void scrubQuotes(String[] array) {
  for (int i = 0; i < array.length; i++) {
    if (array[i].length( ) > 2) {
      // Remove quotes at start and end, if present.
      if (array[i].startsWith("\"") && array[i].endsWith("\"")) {
        array[i] = array[i].substring(1, array[i].length( ) - 1);
      }
    }
    // Make double quotes into single quotes.
    array[i] = array[i].replaceAll("\"\"", "\"");
  }
}
```

6. The Albers Equal-Area Conic is a useful projection when dealing with the United States. Several different map projections applied to the U.S. can be seen on the U.S. Geological Survey's web site at *http://erg.usgs.gov/isb/pubs/booklets/mapsofus/mapsofus.html*. In our case, the specifics of the chosen projection can be found on the helpful MathWorld web site run by Wolfram Research. The following code is an adaptation of the algorithm found at *http://mathworld.wolfram.com/AlbersEqual-AreaConicProjection.html*:

```
// USGS uses standard parallels at 45.5°N and 29.5°N
// with a central meridian value of 96°W.
// Latitude value is phi, longitude is lambda.
float phi0 = 0;
float lambda0 = radians(-96);
float phi1 = radians(29.5f);
float phi2 = radians(45.5f);

float phi = radians(lat);
float lambda = radians(lon);

float n = 0.5f * (sin(phi1) + sin(phi2));
float theta = n * (lambda - lambda0);
float c = sq(cos(phi1)) + 2*n*sin(phi1);
float rho = sqrt(c - 2*n*sin(phi)) / n;
float rho0 = sqrt(c - 2*n*sin(phi0)) / n;

float x = rho * sin(theta);
float y = rho0 - rho*cos(theta);
```

7. As the preprocessor runs, we can keep a running account of the minimum and maximum ranges for the coordinates in question. That is done by setting the maximum arbitrarily small and the minimum arbitrarily high:

   ```
   float minX = MAX_FLOAT;
   float maxX = MIN_FLOAT;
   float minY = MAX_FLOAT;
   float maxY = MIN_FLOAT;
   ```

   and checking the values on each iteration through the loop:

   ```
   if (x > maxX) maxX = x;
   if (x < minX) minX = x;
   if (y > maxY) maxY = y;
   if (y < minY) minY = y;
   ```

   The values will be written to the preprocessor output file so that the boundaries of the shape are known before the file has finished loading in the interactive applet. That allows us to show points as they load from the network, which also serves as a indicator of the progress of the file download.

8. To keep track of the number of locations, the `placeCount` variable is incremented as each new location is parsed.

Building the Preprocessor

Open Processing and start a new sketch. Add the *fips.tsv* file to your sketch by dragging it into the editor window or selecting Sketch.

Pulling all these steps together, the preprocessor code appears as the following:

```
// Indices for each of the columns
int ZIP_CODE = 0;
int LATITUDE = 1;
int LONGITUDE = 2;
int ZIP_CLASS = 3;
int PONAME = 4;
int STATE_FIPS = 5;
int COUNTY = 6;

void setup( ) {
  // Load the state FIPS codes into a table.
  Hashtable fipsTable = new Hashtable( );
  String[] fipsLines = loadStrings("fips.tsv");
  for (int i = 0; i < fipsLines.length; i++) {
    // Split each line on the tab characters.
    String[] pieces = split(fipsLines[i], TAB);
    // The FIPS code is in column 1,
    // and the state abbreviation in column 2
    // (keep in mind that columns are numbered from zero).
    fipsTable.put(pieces[1], pieces[2]);
  }

  String[] lines = loadStrings("zipnov99.csv");
```

```
    // Set the minimum and maximum values arbitrarily large.
    float minX = 1;
    float maxX = -1;
    float minY = 1;
    float maxY = -1;

    // Set up an array for the cleaned data.
    String[] cleaned = new String[lines.length];
    // Number of cleaned entries found
    int placeCount = 0;

    // Start at row 1, because the first row is the column titles.
    for (int row = 1; row < lines.length; row++) {
      // Split the row into pieces on each comma.
      String[] data = split(lines[row], ',');
      scrubQuotes(data);
      // Remove extra whitespace on either side of each column.
      data = trim(data);

      String stateAbbrev = (String) fipsTable.get(data[STATE_FIPS]);
      // If the abbreviation was not found, skip this line,
      // because that means it's an outlying territory.
      if (stateAbbrev == null) continue;
      // For now, also skip Alaska and Hawaii.
      if (stateAbbrev.equals("AK") || stateAbbrev.equals("HI")) continue;

      // Attempt to fix the capitalization of the city/town name.
      String placeName = fixCapitals(data[PONAME]) + ", " + stateAbbrev;

      float lat = float(data[LATITUDE]);
      float lon = float(data[LONGITUDE]);
  \
      // Albers equal-area conic projection.
      // USGS uses standard parallels at 45.5°N and 29.5°N
      // with a central meridian value of 96°W.
      // Latitude value is phi, longitude is lambda.
      float phi0 = 0;
      float lambda0 = radians(-96);
      float phi1 = radians(29.5f);
      float phi2 = radians(45.5f);

      float phi = radians(lat);
      float lambda = radians(lon);

      float n = 0.5f * (sin(phi1) + sin(phi2));
      float theta = n * (lambda - lambda0); //radians(lon - lambda0);
      float c = sq(cos(phi1)) + 2*n*sin(phi1);
      float rho = sqrt(c - 2*n*sin(phi)) / n;
      float rho0 = sqrt(c - 2*n*sin(phi0)) / n;

      float x = rho * sin(theta);
      float y = rho0 - rho*cos(theta);
      if (x > maxX) maxX = x;
      if (x < minX) minX = x;
      if (y > maxY) maxY = y;
      if (y < minY) minY = y;

      // Add a cleaned version of the line, separated by tabs, to the list.
      cleaned[placeCount++] = data[ZIP_CODE] + "\t" +
                              x + "\t"
                              y + "\t"
                              placeName;
    }

    // Write to a file called "zips.tsv" in the sketch folder.
    PrintWriter tsv = createWriter("zips.tsv");

    // Use the first line to specify the number of data points in the file,
    // along with the minimum and maximum latitude and longitude coordinates.
    // Use a # to mark the line as different from the other data.
    tsv.println("# " + placeCount +
                "," + minX + "," + maxX + "," + minY + "," + maxY);

    // Write each line of the cleaned data.
    for (int i = 0; i < placeCount; i++) {
      tsv.println(cleaned[i]);
    }

    // Flush and close the file buffer.
    tsv.flush( );
    tsv.close( );

    // Finished; quit the program.
    println("Finished.");
    exit( );
  }

  // Parse quotes from CSV or TSV data. Quotes around a column are common,
  // and actual double quotes (") are specified by two double quotes ("").
  void scrubQuotes(String[] array) {
    for (int i = 0; i < array.length; i++) {
      if (array[i].length( ) > 2) {
        // Remove quotes at start and end, if present.
        if (array[i].startsWith("\"") && array[i].endsWith("\"")) {
          array[i] = array[i].substring(1, array[i].length( ) - 1);
        }
      }
      // Make double quotes into single quotes.
      array[i] = array[i].replaceAll("\"\"", "\"");
    }
  }

  // Capitalize the first letter of each word in a string.
  String fixCapitals(String title) {
    char[] text = title.toCharArray( );
    // If set to true, the next letter will be capitalized.
    boolean capitalizeNext = true;
```

```
      for (int i = 0; i < text.length; i++) {
        if (Character.isSpace(text[i])) {
          capitalizeNext = true;
        } else if (capitalizeNext) {
          text[i] = Character.toUpperCase(text[i]);
          capitalizeNext = false;
        } else {
          text[i] = Character.toLowerCase(text[i]);
        }
      }
      return new String(text);
    }
```

The resulting file is much easier on the eyes and far simpler to parse in our next step:

```
# 41556,-0.3667764,0.35192886,0.4181981,0.87044954
00210    0.3135056    0.7633538    Portsmouth, NH
00211    0.3135056    0.7633538    Portsmouth, NH
00212    0.3135056    0.7633538    Portsmouth, NH
00213    0.3135056    0.7633538    Portsmouth, NH
00214    0.3135056    0.7633538    Portsmouth, NH
00215    0.3135056    0.7633538    Portsmouth, NH
00501    0.30247012   0.7226447    Holtsville, NY
00544    0.30247012   0.7226447    Holtsville, NY
01001    0.29536617   0.742954     Agawam, MA
01002    0.29843047   0.7478273    Amherst, MA
01003    0.29629046   0.74733305   Amherst, MA
01004    0.29775193   0.7479712    Amherst, MA
01005    0.302632     0.7482096    Barre, MA
01007    0.29958177   0.7465452    Belchertown, MA
01008    0.29309207   0.7430645    Blandford, MA
01009    0.30066824   0.74547637   Bondsville, MA
01010    0.302785     0.74424756   Brimfield, MA
01011    0.29267046   0.74506783   Chester, MA
01012    0.29383755   0.74713147   Chesterfield, MA
01013    0.29678938   0.74368894   Chicopee, MA
01014    0.29752877   0.7440418    Chicopee, MA
01020    0.29802647   0.74428964   Chicopee, MA
```

What about a binary data file or a database?

Of course, one could pack the data into a more sophisticated binary format so that it would be an even smaller file. I'll leave that as an exercise for the reader. Unless the need for space and speed is acute, I prefer to avoid dealing with binary formats. Dealing with such data is tricky because you can't just open a binary file in a text editor to see what's going on. A text file can be run compressed with GZIP and read as a stream, and often is in the neighborhood of the size of a binarized version of the data, while retaining the convenience of text.

Java's *serialization* capabilities are another possibility for storing the data. Serialization allows you to write the contents of the current state of a Java object to the disk; it's a "just add water" approach to storing information from a Java application to be retrieved next time the application is run. Unfortunately, serialization tends to be slower than actually parsing simple information. Furthermore, you have to avoid changing the structure of your Java class because it would mean rewriting the serialized version of the data. No thanks.

Whenever considering a “lot” of data, people tend to start thinking “database” right away. But even though there are almost 42,000 zip code records, entering them into a database is excessive—the data is only 2–3 megabytes—and a database would also prevent the sort of immediate interaction we want as the user types a postal code. This will happen in many scenarios, where shoving the data into RAM is so much more expedient (and helps improve the interaction to such an extent) that databases should be avoided until absolutely necessary. Even in cases where the data might be a few gigabytes, clever use of subsets of the data can help the additional work to set up a database.

Loading the Data (Acquire and Parse)

Fire up Processing, start a new sketch, and add the cleaned version of *zips.tsv* to the sketch.

We start with constants that define the indices for each column:

```
// column numbers in the data file
static final int CODE = 0;
static final int X = 1;
static final int Y = 2;
static final int NAME = 3;
```

The main tab of each sketch is represented by Processing as a class. A single location will be defined using a second class named `Place`. To create this class, use the arrow located at the righthand side of the tab bar and select New Tab from the pop-up menu. Name the tab `Place`. For now, the class has a simple constructor and only keeps track of the name, zip code, and coordinates for each location:

```
class Place {
  int code;
  String name;
  float x;
  float y;

  public Place(int code, String name, float x, float y) {
    this.code = code;
    this.name = name;
    this.x = x;
    this.y = y;
  }
}
```

Back in the main tab, a few variables are necessary to keep track of the number of total places, the number loaded, and the objects themselves:

```
int totalCount; // total number of places
Place[] places;
int placeCount; // number of places loaded
```

The placeCount and totalCount variables are separate because one will be used to allocate room for the total number of locations, and the other will keep track of how many have been loaded so far from the data source. This becomes more important later, when we will load the data asynchronously.

The filtering process has already covered the preprocessing step, and the only thing resembling mining for this project was handled when we calculated the values for the minimum and maximum coordinates in the preprocessing step:

```
// min/max boundary of all points
float minX, maxX;
float minY, maxY;
```

Now the *zips.tsv* file can be parsed in a straightforward fashion. Because we've already cleaned the data, there's no additional code to validate individual rows or to rid gremlins from the columns. The readData( ) method orchestrates the data acquisition and parsing. The parseInfo( ) method reads the header line from the file, and parsePlace( ) converts a single line of the data file into a Place object:

```
public void setup( ) {
  readData( );
}

void readData( ) {
  String[] lines = loadStrings("zips.tsv");
  parseInfo(lines[0]); // read the header line

  places = new Place[totalCount];
  for (int i = 1; i < lines.length; i++) {
    places[placeCount] = parsePlace(lines[i]);
    placeCount++;
  }
}

void parseInfo(String line) {
  String infoString = line.substring(2); // remove the #
  String[] infoPieces = split(infoString, ',');
  totalCount = int(infoPieces[0]);
  minX = float(infoPieces[1]);
  maxX = float(infoPieces[2]);
  minY = float(infoPieces[3]);
  maxY = float(infoPieces[4]);
}

Place parsePlace(String line) {
  String pieces[] = split(line, TAB);

  int zip = int(pieces[CODE]);
  float x = float(pieces[X]);
```

```
    float y = float(pieces[Y]);
    String name = pieces[NAME];

    return new Place(zip, name, x, y);
  }
```

Running this program is not particularly satisfying; if everything is working properly, nothing will happen when the sketch runs.

Drawing a Scatterplot of Zip Codes (Mine and Represent)

After parsing your data, you must give some consideration to how the data is mapped to the screen. The x and y coordinates from the data file do not correspond to locations on the screen, so the `map( )` function is used to remap them to a useful coordinate space. As with previous examples, we'll set up coordinates for the bounding box where the map should be drawn. A modified `setup( )` method sets the values in slightly from the width and height of the plot:

```
// Border of where the map should be drawn on screen
float mapX1, mapY1;
float mapX2, mapY2;

public void setup( ) {
  size(720, 453, P3D);

  mapX1 = 30;
  mapX2 = width - mapX1;
  mapY1 = 20;
  mapY2 = height - mapY1;

  readData( );
}
```

Next, add a method named `draw( )` to the `Place` class to draw a single location:

```
  void draw( ) {
    int xx = (int) TX(x);
    int yy = (int) TY(y);
    set(xx, yy, #000000);
  }
```

And back in the host application, add an umbrella `draw( )` method that will handle calling the `draw( )` method for each place. In addition, functions named `TX( )` and `TY( )` (for *transform* x and y) handle calling `map( )` to map the points to the screen:

```
public void draw( ) {
  background(255);
  for (int i = 0; i < placeCount; i++) {
    places[i].draw( );
  }
}
```

```
    float TX(float x) {
      return map(x, minX, maxX, mapX1, mapX2);
    }

    float TY(float y) {
      return map(y, minY, maxY, mapY2, mapY1);
    }
```

Running this version of the code plots thousands of locations. It essentially yields a population density map of the United States because higher populated areas have more postal codes; see Figure 6-1.

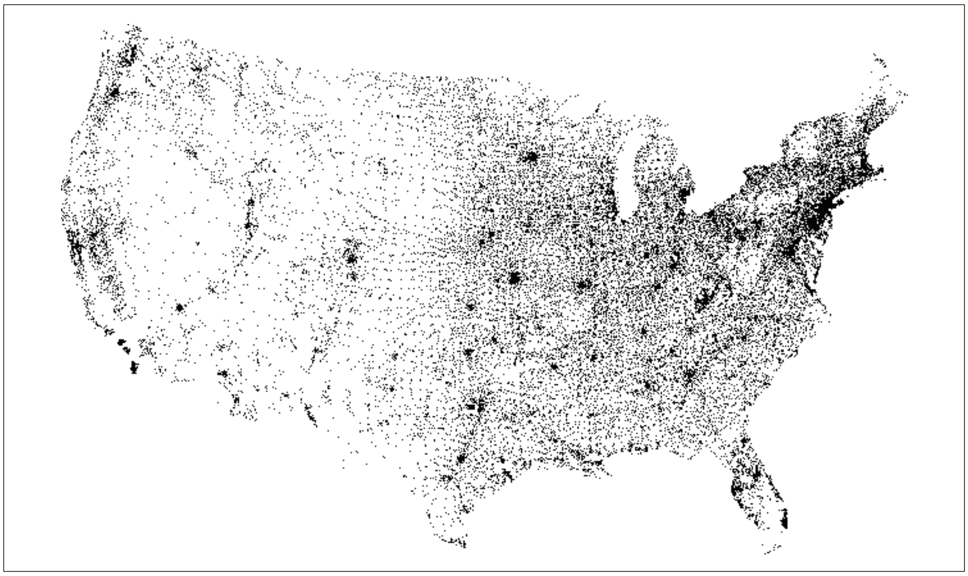

Figure 6-1. Geographic locations of postal zip codes

Highlighting Points While Typing (Refine and Interact)

Returning to the questions that started this chapter, the focus is now to add interaction so that users can explore how the postal codes relate to geography. The user will be asked to type a series of digits, and as she types each digit, locations will light or dim based on whether they are part of a zip code typed so far. For instance, typing `0` and then `2` will dim any locations not part of `02XXX`.

The refinement stage begins with choosing a set of colors. First, we choose a better background than white, followed by an initial color that the map will have when no numbers have been typed. After that, we choose colors that highlight locations whose codes include the numbers already typed, and an additional color to indicate when there are no available zip codes that use the digits typed so far:

```
color backgroundColor    = #333333; // dark background color
color dormantColor       = #999966; // initial color of the map
color highlightedColor   = #CBCBCB; // color for selected points
color unhighlightedColor = #66664C; // color for points that are not selected
color badColor           = #FFFF66; // text color when nothing found
```

A font is needed for the text we use to provide the user with feedback on what she has typed so far. The `typedChars` array will contain letters typed, and `typedCount` keeps track of the number of digits entered. The `messageX` and `messageY` values will be the location where the text should be drawn, and `foundCount` will be the number of locations currently selected. The `typedPartials` variable is used to make selection fast (more about this later):

```
PFont font;
String typedString = "";
char typedChars[] = new char[5];
int typedCount;
int typedPartials[] = new int[6];
float messageX, messageY;
int foundCount;
```

Inside `setup( )`, add the following lines to load the font and set the message text location. Use Tools → Create Font to replace *ScalaSans-Regular-14.vlw* with the name of the font that you create. The `textMode(SCREEN)` line specifies that the text be drawn at its original size, in screen space (no transformations):

```
  font = loadFont("ScalaSans-Regular-14.vlw");
  textFont(font);
  textMode(SCREEN);

  messageX = 40;
  messageY = height - 40;
```

The letters typed by the user are handled by a `keyPressed( )` method, along with an additional method that is called whenever changes are made to the current selection:

```
void keyPressed( ) {
  if ((key == BACKSPACE) || (key == DELETE)) {
    if (typedCount > 0) {
      typedCount--;
    }
    updateTyped( );

  } else if ((key >= '0') && (key <= '9')) {
    if (typedCount != 5) { // Stop at 5 digits.
      if (foundCount > 0) { // If nothing found, ignore further typing.
        typedChars[typedCount++] = key;
      }
    }
  }
  updateTyped( );
}
```

```
void updateTyped( ) {
  typedString = new String(typedChars, 0, typedCount);
  typedPartials[typedCount] = int(typedString);
  for (int j = typedCount-1; j > 0; --j) {
    typedPartials[j] = typedPartials[j + 1] / 10;
  }

  foundCount = 0;
  for (int i = 0; i < placeCount; i++) {
    // Update boundaries of selection
    // and identify whether a particular place is chosen.
    places[i].check( );
  }
```

Inside updateTyped, the typedString value is updated by creating a new String object from the typedChars array, one that's based on the number of characters that have been typed so far. The typedPartials array divides each zip code by 10. For instance, when the user types 15232, println(typedPartials) produces:

```
[0] 0
[1] 1
[2] 15
[3] 152
[4] 1523
[5] 15232
```

This creates a quick way to see how well each location matches. The Place class includes a lot of work to handle the new changes:

```
class Place {
  int code;
  String name;
  float x;
  float y;

  int partial[];
  int matchDepth;


  public Place(int code, String name, float x, float y) {
    this.code = code;
    this.name = name;
    this.x = x;
    this.y = y;

    partial = new int[6];
    partial[5] = code;
    partial[4] = partial[5] / 10;
    partial[3] = partial[4] / 10;
    partial[2] = partial[3] / 10;
    partial[1] = partial[2] / 10;
  }
```

```
    void check( ) {
      // Default to zero levels of depth that match
      matchDepth = 0;

      if (typedCount != 0) {
        // Start from the greatest depth, and work backwards to see how many
        // items match. Want to figure out the maximum match, so better to
        // begin from the end.
        for (int j = typedCount; j > 0; --j) {
          if (typedPartials[j] == partial[j]) {
            matchDepth = j;
            break; // Since starting at end, can stop now.
          }
        }
      }

      if (matchDepth == typedCount) {
        foundCount++;
      }
    }

    void draw( ) {
      int xx = (int) TX(x);
      int yy = (int) TY(y);

      color c = dormantColor;
      if (typedCount != 0) {
        if (matchDepth == typedCount) {
          c = highlightedColor;
        } else {
          c = unhighlightedColor;
        }
      }
      set(xx, yy, c);
    }
  }
```

The check( ) method calculates whether a point is selected and increments the foundCount variable to keep track of how many locations are still valid. Inside draw( ), a point can be one of three colors: the dormantColor when nothing has been typed, the highlightedColor when the point matches, and the unhighlightedColor when the point does not match.

The new version of the code answers our initial question and makes it easy to compare regions against one another by typing different numbers. Typing 4 as the first digit produces the image shown in Figure 6-2.

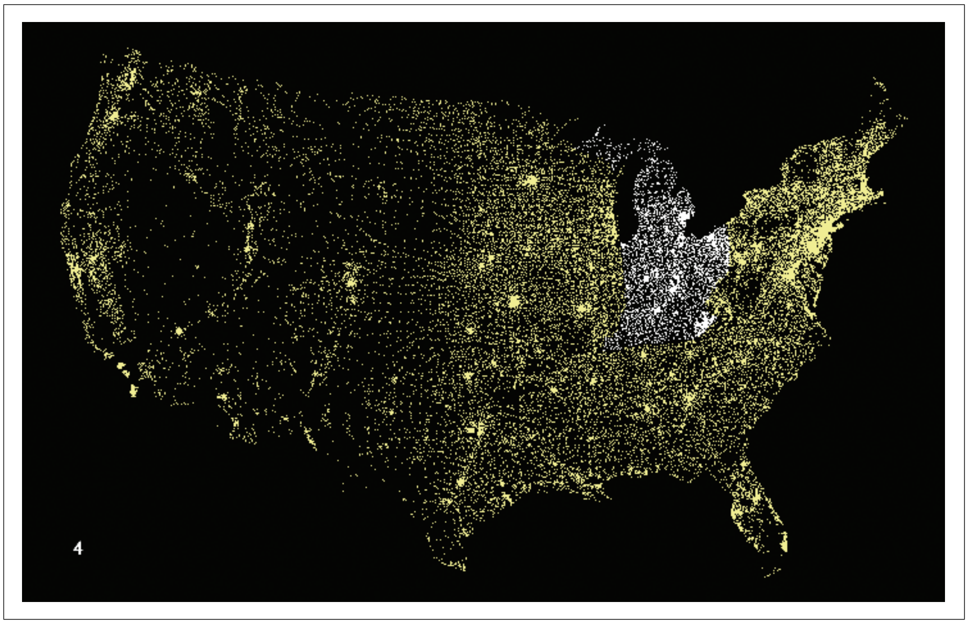

Figure 6-2. Selecting a region of zip codes

Show the Currently Selected Point (Refine)

When five digits have been typed, the point should appear different and include the text for the location's name. In this section, we'll show the location of a fully typed-out, five-digit zip code as a rectangle and add text to its upper-right corner that names the location. That is done with the `drawChosen( )` method:

```
void drawChosen( ) {
  noStroke( );
  fill(highlightColor);
  int size = 4;
  rect(TX(x), TY(y), size, size);

  // Calculate position to draw the text, offset slightly from the main point.
  float textX = TX(x);
  float textY = TY(y) - size - 4;

  // Don't go off the top (e.g., 59544).
  if (textY < 20) {
    textY = TY(y) + 20;
  }
```

```
        // Don't run off the bottom (e.g., 33242).
        if (textY > height - 5) {
          textY = TY(y) - 20;
        }

        String location = name + " " + nf(code, 5);
        float wide = textWidth(location);

        if (textX > width/3) {
          textX -= wide + 8;
        } else {
          textX += 8;
        }

        textAlign(LEFT);
        fill(highlightColor);
        text(location, textX, textY);
      }
    }
```

Because the text will be shown adjacent to the point, it's necessary to also make sure that the text does not leave the edge of the screen. The middle lines of the method cover this logic.

In the main tab, a variable named `chosen` keeps track of the current `Place` object (if any):

```
Place chosen;
```

The chosen point is drawn with a rectangle. To center it, add the following line to `setup( )`:

```
  rectMode(CENTER);
```

The following modifications to `draw( )` carry out the extra step to draw the chosen point:

```
public void draw( ) {
  background(backgroundColor);

  for (int i = 0; i < placeCount; i++) {
    places[i].draw( );
  }

  if (typedCount != 0) {
    if (foundCount > 0) {
      if (typedCount == 4) {
        // Redraw the chosen ones, because they're often occluded
        // by the non-selected points.
        for (int i = 0; i < placeCount; i++) {
          if (places[i].matchDepth == typedCount) {
            places[i].draw( );
          }
        }
      }
```

```
          if (chosen != null) {
            chosen.drawChosen( );
          }

          fill(highlightColor);
          textAlign(LEFT);
          text(typedString, messageX, messageY);

        } else {
          fill(badColor);
          text(typedString, messageX, messageY);
        }
      }
    }
```

And in updateTyped( ), chosen should be set to null after foundCount is set to 0, which resets the current selection whenever typing occurs.

The result is shown in Figure 6-3.

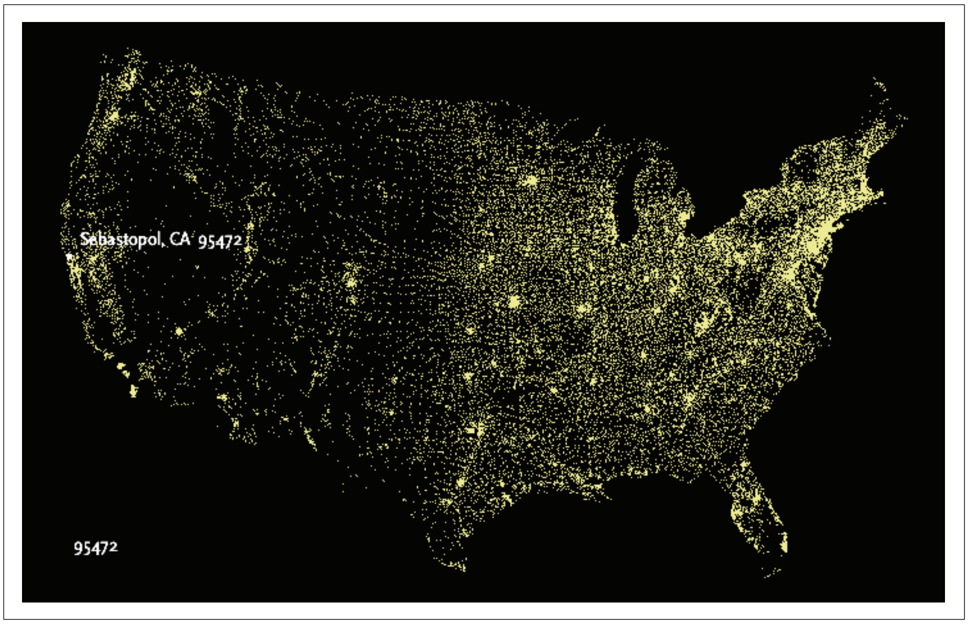

Figure 6-3. Special handling of a full five-digit user entry

Progressively Dimming and Brightening Points (Refine)

In the previous examples, typing makes the screen update instantaneously, which can be disorienting for users trying to compare different areas. In Chapter 3, the `Integrator` class was used to create a more gradual interpolation between two data values. For this chapter, a class called `ColorIntegrator` does the same, but interpolates between two colors. Instead of a `float`, the `ColorIntegrator.target( )` method takes a color (in web color format, or created with the `color( )` function). Both classes can be downloaded from the book's site at:

http://benfry.com/writing/zipdecode/Integrator.java
http://benfry.com/writing/zipdecode/ColorIntegrator.java

To handle a smooth interpolation, a set of six `ColorIntegrator` objects are used, one for each possible number typed plus an initial value for cases in which no digits are typed. When no digits have been typed, `faders[0]` is used to determine the color. When the user begins typing, all points that match one digit are drawn with the color from `faders[1]`. All points that match two digits are drawn with `faders[2]`, and so on. For instance, as the user types `021`, the following occurs:

- With no digits typed, all points are drawn with the color value in `faders[0]`. Inside `setup( )`, `faders[0]` is set to `unhighlightedColor`, but its `target( )` method is set to `dormantColor`, a pale yellow. This has the effect of fading all the points from the duller `unhighlightedColor` into the brighter `dormantColor`. This effect has no functional purpose, but letting the points fade in is more pleasing than making 42,000 points appear instantaneously.
- When the user presses 0, `faders[1]` is told through its `target( )` method to target `highlightedColor`, which causes the points that match to transition gradually from their previous color to `highlightedColor`. In the same code sequence, `faders[0]` is set to target `unhighlightedColor`, causing all points that don't match the user's 0 to begin fading out. All points that match the first digit are drawn using `faders[1]`, and all points that match zero digits are drawn with `faders[0]`.
- When the user presses 2 (so that the screen now reads `02`), `faders[2]` is set to target `highlightedColor`, whereas `faders[0]` and `faders[1]` are set to target `unhighlightedColor`. `faders[0]` has probably already reached its target value of `unhighlightedColor`, and `faders[1]` will arrive shortly.

Each fader is handled individually because the transition from one set of points to another may overlap. That is, if the user presses a first digit, a set of points begins to dim. Those points may not have dimmed completely when the next digit is typed. Separate `ColorIntegrator` objects support the overlap by independently handling each transition.

The array of `ColorIntegrator` objects is created as a global variable:

```
ColorIntegrator faders[];
```

The objects are created inside setup( ):

```
    faders = new ColorIntegrator[6];

    // When nothing is typed, all points are shown with a color called
    // "dormant," which is brighter than when not highlighted, but
    // not as bright as the highlight color for a selection.
    faders[0] = new ColorIntegrator(unhighlightedColor);
    faders[0].setAttraction(0.5);
    faders[0].target(dormantColor);

    for (int i = 1; i < 6; i++) {
      faders[i] = new ColorIntegrator(unhighlightedColor);
      faders[i].setAttraction(0.5);
      faders[i].target(highlightedColor);
    }
```

As with any time animation, it is a good idea to use frameRate( ) to ensure that the transitions behave identically on faster or slower machines:

```
  frameRate(15);
```

Next, add an updateAnimation( ) method that updates the value for each fader:

```
  void updateAnimation( ) {
    for (int i = 0; i < 6; i++) {
      faders[i].update( );
    }
  }
```

Call this method at the beginning of draw( ).

The fading values are handled in updateTyped( ) by targeting the highlightedColor or unhighlightedColor based on how many digits have been typed so far:

```
  void updateTyped( ) {
    typedString = new String(typedChars, 0, typedCount);

      if (typedCount == 0) {
        faders[0].target(dormantColor);

    } else {
      // Un-highlight areas already typed past.
      for (int i = 0; i < typedCount; i++) {
        faders[i].target(unhighlightedColor);
      }
      // Highlight remaining points that match what's typed.
      for (int i = typedCount; i < 6; i++) {
        faders[i].target(highlightedColor);
      }
    }

    typedPartials[typedCount] = int(typedString);
    for (int j = typedCount-1; j > 0; --j) {
      typedPartials[j] = typedPartials[j + 1] / 10;
    }
    foundCount = 0;
    chosen = null;

    for (int i = 0; i < placeCount; i++) {
      // Update boundaries of selection
      // and identify whether a particular place is chosen.
      places[i].check( );
    }
  }
```

Inside the draw( ) method for Place, the color value is now set based on how far things have faded for the depth at which the location matches:

```
      set(xx, yy, faders[matchDepth].value);
```

For instance, if four digits have been typed but this location possesses only the first two, the color of this point will be based on how bright or dim the faders[2] element is.

You will have to try the code directly to see for yourself; the result is difficult to show in a figure because it involves a subtle fading effect.

Zooming In (Interact)

Because interaction in our zip code example involves honing in on an ever-smaller area of locations, the application begs for a capability to zoom closer as the area becomes more specific. The effect of zooming is striking, but its implementation is actually quite straightforward and can be based on methods already in use.

So far, the crux of the representation is the map( ) function, which remaps a series of coordinates (with a predefined range) to a specific location on the screen (with a new range). To allow for zooming, instead set the ranges themselves as Integrator objects:

```
color backgroundColor    = #333333; // dark background color
color dormantColor       = #999966; // initial color of the map
color highlightedColor   = #CBCBCB; // color for selected points
color unhighlightedColor = #66664C; // color for points that are not selected
color badColor           = #FFFF66; // text color when nothing found

ColorIntegrator faders[];

// Border of where the map should be drawn on screen
float mapX1, mapY1;
float mapX2, mapY2;

// Column numbers in the data file
static final int CODE = 0;
static final int X = 1;
static final int Y = 2;
static final int NAME = 3;

int totalCount; // total number of places
```

```
Place[] places;
int placeCount; // number of places loaded

// Min/max boundary of all points
float minX, maxX;
float minY, maxY;

// Typing and selection
PFont font;
String typedString = "";
char typedChars[] = new char[5];
int typedCount;
int typedPartials[] = new int[6];

float messageX, messageY;

int foundCount;
Place chosen;

// Zoom
boolean zoomEnabled = false;
Integrator zoomDepth = new Integrator( );

Integrator zoomX1;
Integrator zoomY1;
Integrator zoomX2;
Integrator zoomY2;

float targetX1[] = new float[6];
float targetY1[] = new float[6];
float targetX2[] = new float[6];
float targetY2[] = new float[6];

// Boundary of currently valid points at this typedCount
float boundsX1, boundsY1;
float boundsX2, boundsY2;


public void setup( ) {
  size(720, 453, P3D);

  mapX1 = 30;
  mapX2 = width - mapX1;
  mapY1 = 20;
  mapY2 = height - mapY1;

  font = loadFont("ScalaSans-Regular-14.vlw");
  textFont(font);
  textMode(SCREEN);

  messageX = 40;
  messageY = height - 40;

  faders = new ColorIntegrator[6];
  // When nothing is typed, all points are shown with a color called
  // "dormant," which is brighter than when not highlighted, but
  // not as bright as the highlight color for a selection.
  faders[0] = new ColorIntegrator(unhighlightedColor);
  faders[0].setAttraction(0.5);
  faders[0].target(dormantColor);

  for (int i = 1; i < 6; i++) {
    faders[i] = new ColorIntegrator(unhighlightedColor);
    faders[i].setAttraction(0.5);
    faders[i].target(highlightedColor);
  }

  readData( );

  zoomX1 = new Integrator(minX);
  zoomY1 = new Integrator(minY);
  zoomX2 = new Integrator(maxX);
  zoomY2 = new Integrator(maxY);

  targetX1[0] = minX;
  targetX2[0] = maxX;
  targetY1[0] = minY;
  targetY2[0] = maxY;

  rectMode(CENTER);
  ellipseMode(CENTER);
  frameRate(15);
}


void readData( ) {
  String[] lines = loadStrings("zips.tsv");
  parseInfo(lines[0]); // Read the header line

  places = new Place[totalCount];
  for (int i = 1; i < lines.length; i++) {
    places[placeCount] = parsePlace(lines[i]);
    placeCount++;
  }
}


void parseInfo(String line) {
  String infoString = line.substring(2); // Remove the #
  String[] infoPieces = split(infoString, ',');
  totalCount = int(infoPieces[0]);
  minX = float(infoPieces[1]);
  maxX = float(infoPieces[2]);
  minY = float(infoPieces[3]);
  maxY = float(infoPieces[4]);
}
```

```
Place parsePlace(String line) {
  String pieces[] = split(line, TAB);

  int zip = int(pieces[CODE]);
  float x = float(pieces[X]);
  float y = float(pieces[Y]);
  String name = pieces[NAME];

  return new Place(zip, name, x, y);
}


public void draw( ) {
  background(backgroundColor);

  updateAnimation( );

  for (int i = 0; i < placeCount; i++) {
    places[i].draw( );
  }

  if (typedCount != 0) {
    if (foundCount > 0) {
      if (!zoomEnabled && (typedCount == 4)) {
        // Redraw the chosen ones, because they're often occluded
        // by the non-selected points.
        for (int i = 0; i < placeCount; i++) {
          if (places[i].matchDepth == typedCount) {
            places[i].draw( );
          }
        }
      }

      if (chosen != null) {
        chosen.drawChosen( );
      }

      fill(highlightColor);
      textAlign(LEFT);
      text(typedString, messageX, messageY);

    } else {
      fill(badColor);
      text(typedString, messageX, messageY);
    }
  }

  // Draw "zoom" text toggle.
  textAlign(RIGHT);
  fill(zoomEnabled ? highlightColor : unhighlightColor);
  text("zoom", width - 40, height - 40);
  textAlign(LEFT);
}
```

```
void updateAnimation( ) {
  for (int i = 0; i < 6; i++) {
    faders[i].update( );
  }

  if (foundCount > 0) {
    zoomDepth.target(typedCount);
  } else {
    // If no points were found, use the previous zoom depth
    // (which will be the last depth where foundCount was > 0).
    zoomDepth.target(typedCount-1);
  }
  zoomDepth.update( );

  zoomX1.update( );
  zoomY1.update( );
  zoomX2.update( );
  zoomY2.update( );
}


float TX(float x) {
  if (zoomEnabled) {
    return map(x, zoomX1.value, zoomX2.value, mapX1, mapX2);

  } else {
    return map(x, minX, maxX, mapX1, mapX2);
  }
}


float TY(float y) {
  if (zoomEnabled) {
    return map(y, zoomY1.value, zoomY2.value, mapY2, mapY1);

  } else {
    return map(y, minY, maxY, mapY2, mapY1);
  }
}


void mousePressed( ) {
  // If the user clicks the "zoom" text, toggle zoomEnabled.
  if ((mouseX > width-100) && (mouseY > height - 50)) {
    zoomEnabled = !zoomEnabled;
  }
}


void keyPressed( ) {
  if ((key == BACKSPACE) || (key == DELETE)) {
    if (typedCount > 0) {
      typedCount--;
    }
  } else if ((key >= '0') && (key <= '9')) {
    if (typedCount != 5) { // only 5 digits
      if (foundCount > 0) { // don't allow to keep typing bad
        typedChars[typedCount++] = key;
      }
    }
  }
  updateTyped( );
}


void updateTyped( ) {
  typedString = new String(typedChars, 0, typedCount);

  if (typedCount == 0) {
    faders[0].target(dormantColor);

  } else {
    // Un-highlight areas already typed past.
    for (int i = 0; i < typedCount; i++) {
      faders[i].target(unhighlightedColor);
    }
    // Highlight remaining points that match what's typed.
    for (int i = typedCount; i < 6; i++) {
      faders[i].target(highlightedColor);
    }
  }

  typedPartials[typedCount] = int(typedString);
  for (int j = typedCount-1; j > 0; --j) {
    typedPartials[j] = typedPartials[j + 1] / 10;
  }

  foundCount = 0;
  chosen = null;

  boundsX1 = maxX;
  boundsY1 = maxY;
  boundsX2 = minX;
  boundsY2 = minY;

  for (int i = 0; i < placeCount; i++) {
    // Update boundaries of selection
    // and identify whether a particular place is chosen.
    places[i].check( );
  }
  calcZoom( );
}


void calcZoom( ) {
  if (foundCount != 0) {
    // Given a set of min/max coords, expand in one direction so that the
    // selected area includes the range with the proper aspect ratio.
```

```
      float spanX = (boundsX2 - boundsX1);
      float spanY = (boundsY2 - boundsY1);

      float midX = (boundsX1 + boundsX2) / 2;
      float midY = (boundsY1 + boundsY2) / 2;

      if ((spanX != 0) && (spanY != 0)) {
        float screenAspect = width / float(height);
        float spanAspect = spanX / spanY;

        if (spanAspect > screenAspect) {
          spanY = (spanX / width) * height; // wide

        } else {
          spanX = (spanY / height) * width; // tall
        }
      } else { // if span is zero
        // use the span from one level previous
        spanX = targetX2[typedCount-1] - targetX1[typedCount-1];
        spanY = targetY2[typedCount-1] - targetY1[typedCount-1];
      }
      targetX1[typedCount] = midX - spanX/2;
      targetX2[typedCount] = midX + spanX/2;
      targetY1[typedCount] = midY - spanY/2;
      targetY2[typedCount] = midY + spanY/2;

    } else if (typedCount != 0) {
      // Nothing found at this level, so set the zoom identical to the previous.
      targetX1[typedCount] = targetX1[typedCount-1];
      targetY1[typedCount] = targetY1[typedCount-1];
      targetX2[typedCount] = targetX2[typedCount-1];
      targetY2[typedCount] = targetY2[typedCount-1];
    }

    zoomX1.target(targetX1[typedCount]);
    zoomY1.target(targetY1[typedCount]);
    zoomX2.target(targetX2[typedCount]);
    zoomY2.target(targetY2[typedCount]);

    if (!zoomEnabled) {
      zoomX1.set(zoomX1.target);
      zoomY1.set(zoomY1.target);
      zoomX2.set(zoomX2.target);
      zoomY2.set(zoomY2.target);
    }
  }
```

The zoomX1, zoomY1, zoomX2, and zoomY2 variables are the new ranges to be used with the map( ) function. When zoom is not enabled, the horizontal coordinate of a location on screen is calculated by using map( ) to convert the value from the range minX to maxX into the range mapX1 to mapX2. When zooming, we replace minX and maxX with the minimum and maximum values that we want to be visible onscreen. That way, coordinates inside that range will be mapped between mapX1 and mapX2, while map( )

will place the others somewhere to the left of `mapX1` (probably offscreen to the left), or an x value greater than `mapX2` (offscreen to the right).

The `targetX1`, `targetY1`, `targetX2`, and `targetY2` arrays contain the boundaries for each zoom level (that is, the number of digits typed so far, which the code maintains in the `typedCount` variable). Inside `calcZoom()`, the span of the currently valid points is calculated for the current zoom level.

The `spanAspect` and `screenAspect` variables are used to expand the target range vertically or horizontally so that it fits the aspect ratio of the screen. For instance, after typing 0, the application will zoom to the northeastern United States. This set of points is taller than it is wide, so the height of the points will be used to determine the vertical span, and the horizontal span (`spanX`) will be expanded to stay in proportion.

If the span is zero, as is the case when a single zip code is selected, the boundary is instead set to the previous zoom level, but with the final coordinate centered inside that range. The target levels are based on the previous boundary of all points. For instance, after typing 940, the boundary for the zoom is based on the minimum and maximum locations when only 9 and 4 had been typed.

A text label onscreen handles toggling the zoom mode. The `mousePressed()` method checks to see whether the mouse has been pressed inside the range of this label's location. After typing 4, the image looks like Figure 6-4.

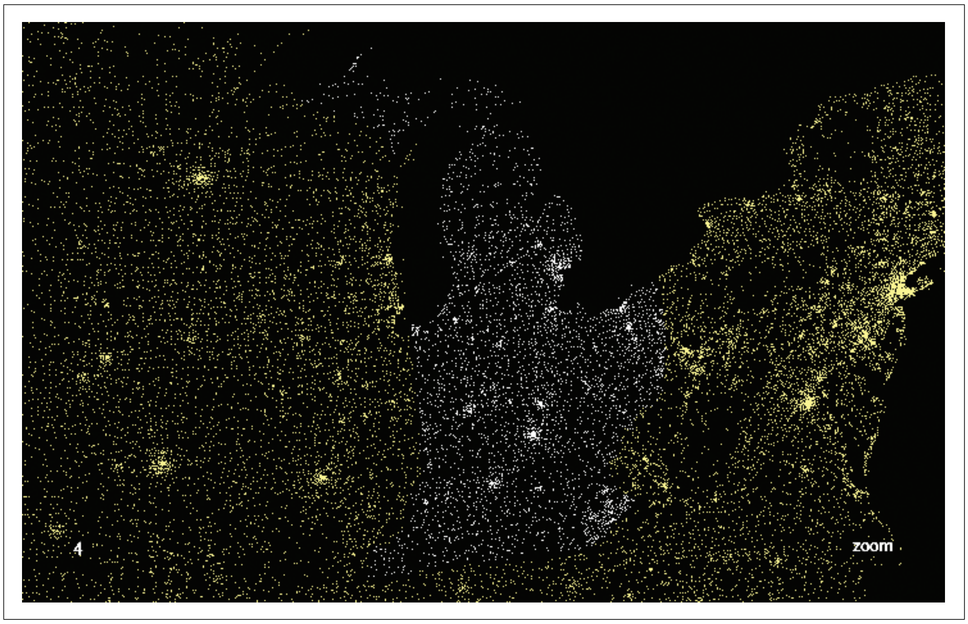

Figure 6-4. Appearance after typing 4 with zoom enabled

Additional minor changes inside the Place class handle calculating the boundary inside the check( ) method and drawing the selected point slightly differently. Because the changes are small and spread throughout, the entire code is shown here, with the modifications highlighted:

```
class Place {
  int code;
  String name;
  float x;
  float y;

  int partial[];
  int matchDepth;


  public Place(int code, String name, float lon, float lat) {
    this.code = code;
    this.name = name;
    this.x = lon;
    this.y = lat;

    partial = new int[6];
    partial[5] = code;
    partial[4] = partial[5] / 10;
    partial[3] = partial[4] / 10;
    partial[2] = partial[3] / 10;
    partial[1] = partial[2] / 10;
  }


  void check( ) {
    // Default to zero levels of depth that match
    matchDepth = 0;

    if (typedCount != 0) {
      // Start from the greatest depth, and work backwards to see how many
      // items match. Want to figure out the maximum match, so better to
      // begin from the end.
      // The multiple levels of matching are important because more than one
      // depth level might be fading at a time.
      for (int j = typedCount; j > 0; --j) {
        if (typedPartials[j] == partial[j]) {
          matchDepth = j;
          break; // since starting at end, can stop now
        }
      }
    }

    //if (partial[typedCount] == partialCode) {
    if (matchDepth == typedCount) {
      foundCount++;
      if (typedCount == 5) {
        chosen = this;
      }
      if (x < boundsX1) boundsX1 = x;
      if (y < boundsY1) boundsY1 = y;
      if (x > boundsX2) boundsX2 = x;
      if (y > boundsY2) boundsY2 = y;
    }
  }

  void draw( ) {
    int xx = (int) TX(x);
    int yy = (int) TY(y);

    if ((xx < 0) || (yy < 0) || (xx >= width) || (yy >= height)) return;

    set(xx, yy, faders[matchDepth].colorValue);
  }


  void drawChosen( ) {
    noStroke( );
    fill(faders[matchDepth].colorValue);
    // The chosen point has to be a little larger when zooming.
    int size = zoomEnabled ? 6 : 4;
    rect(TX(x), TY(y), size, size);

    // Calculate position to draw the text, slightly offset from the main point.
    float textX = TX(x);
    float textY = TY(y) - size - 4;

    // Don't go off the top (e.g., 59544).
    if (textY < 20) {
      textY = TY(y) + 20;
    }

    // Don't run off the bottom (e.g., 33242).
    if (textY > height - 5) {
      textY = TY(y) - 20;
    }

    String location = name + " " + nf(code, 5);

    if (zoomEnabled) {
      // Center the single point onscreen when zooming.
      textAlign(CENTER);
      text(location, textX, textY);

    } else {
      float wide = textWidth(location);

      if (textX > width/3) {
        textX -= wide + 8;
      } else {
        textX += 8;
      }

      textAlign(LEFT);
```

```
        fill(highlightColor);
        text(location, textX, textY);
      }
    }
  }
```

Changing How Points Are Drawn When Zooming (Refine)

A bit of time spent playing with the zoom mode makes it clear that points are quickly lost because the single pixels spread out and become difficult to see. This problem springs from how we perceive images, and it requires that points be handled differently as the zoom shifts.

To resolve the issue, points should become larger as more digits are typed. Even better would be to add more information along the way, keeping the density of information on the screen constant as the zoom occurs. For instance, after typing three digits, map data depicting state outlines and major interstates could be included. Although the data retrieval and formatting for that enhancement are too complicated to show in this chapter, we can at least implement one simple way to add a little more information: showing the remaining possibilities for the last digit after the user has typed the first four.

An updated version of the `draw( )` method found inside the `Place` class handles two refinements: changing points to rectangles and showing possible final digits:

```
void draw( ) {
  float xx = TX(x);
  float yy = TY(y);

  if ((xx < 0) || (yy < 0) || (xx >= width) || (yy >= height)) return;

  if ((zoomDepth.value < 2.8f) || !zoomEnabled) { // show simple dots
    set((int)xx, (int)yy, faders[matchDepth].colorValue);

  } else { // show slightly more complicated dots
    noStroke( );

    fill(faders[matchDepth].colorValue);

    if (matchDepth == typedCount) {
      if (typedCount == 4) { // on the fourth digit, show possibilities
        text(code % 10, TX(x), TY(y));
      } else { // show a larger box for selections
        rect(xx, yy, zoomDepth.value, zoomDepth.value);
      }
    } else { // show a slightly smaller box for unselected
      rect(xx, yy, zoomDepth.value-1, zoomDepth.value-1);
    }
  }
}
```

Figure 6-5 shows a selection, with the adjacent locations being drawn using rectangles instead of single pixels as a result of the new `draw()` method.

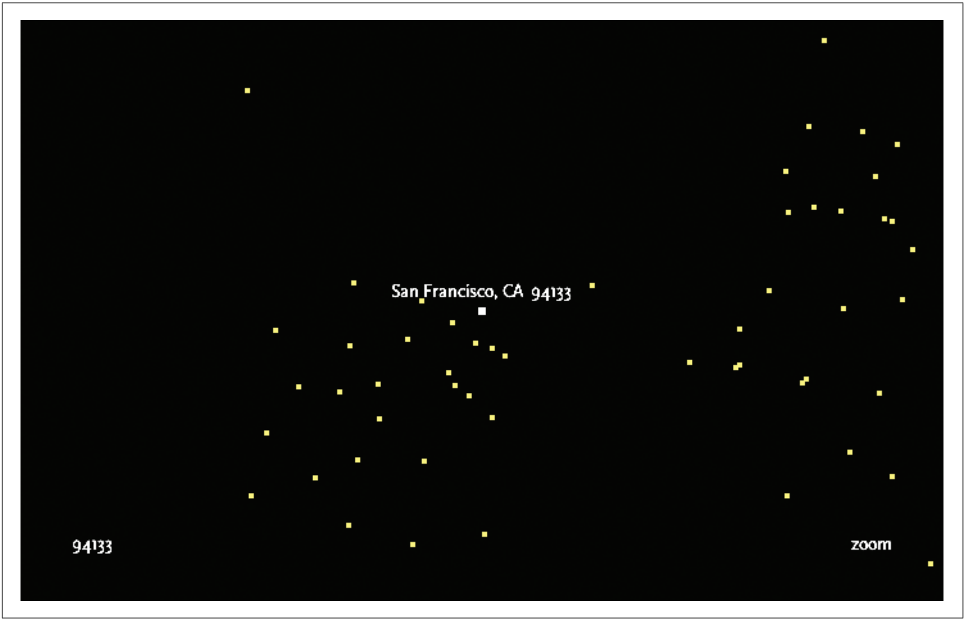

Figure 6-5. After typing five digits, a city name appears for that location

Deployment Issues (Acquire and Refine)

For an online project, downloading two megabytes of data is likely to be a problem. In its current iteration, the program will stop until all data has been downloaded. A better alternative is to use the built-in `Thread` class to load the data asynchronously. The thread acts independently of the rest of the program, gradually adding locations by incrementing `placeCount`. When `placeCount` and `totalCount` are identical, the data has finished loading.

A *thread* provides a way to bundle a function in your program so that it can run at the same time as another part of the program. In this case, rather than waiting for the data to download, a thread can be used for the download while the main `draw()` method of the program continues to run. Because the program is still running, it remains responsive to user input, which makes it feel faster than if the program halted until the download was finished.

The change also means that the data file should be moved *out* of the data folder, and will instead be placed adjacent to the sketch when on your server. In the sketch folder, move the *zips.tsv* file one level up from its previous location. This way, the much smaller download will happen quickly, and the applet will start almost immediately (drawing the background and making it clear to the user that things are working).

The data should also be gzip compressed to save some space (and therefore time downloading). That can be done using a utility that will gzip-encode your file, but for the purposes of this chapter, you can download a version that's already been compressed:

http://benfry.com/writing/zipdecode/zips.gz

The code to handle asynchronous loading should be placed in a new file (a new tab in the Processing interface) called `Slurper`. The contents should be as follows:

```
class Slurper implements Runnable {

  Slurper( ) {
    Thread thread = new Thread(this);
    thread.start( );
  }

  public void run( ) {
    try {
      BufferedReader reader = createReader("zips.gz");

      // First get the info line.
      String line = reader.readLine( );
      parseInfo(line);

      places = new Place[totalCount];

      // Parse each of the rest of the lines.
      while ((line = reader.readLine( )) != null) {
        places[placeCount] = parsePlace(line);
        placeCount++;
      }
    } catch (IOException e) {
      e.printStackTrace( );
    }
  }
}
```

A `Runnable` is a class that has a `run( )` method. The `Thread` class takes a `Runnable` and begins executing it independently of the code from which it was called. The `createReader( )` method will automatically uncompress the gzip-compressed data as it is read. The `BufferedReader` syntax is described in Chapter 9, but the rest is largely similar to the original `readData( )` method. Back in the main tab, the `readData( )` method now need only start the thread:

```
void readData( ) {
  new Slurper( );
}
```

This code will create the new `Slurper` object, which will take care of loading the points as they become available from the network.

A nice feature of this method is that the points themselves serve as a kind of progress bar to alert the user to the progress of the download. Over a slow connection, the points will gradually appear, moving from right to left as the data begin to load.

Next Steps

From this point, there are several directions in which to take this project. For instance:

- Using the technique described in Chapter 5, we can check to see whether the `Integrator` objects have actually changed during `update()`. If no change has occurred, we can disable animation with `noLoop()` to make the program less demanding on the CPU (and therefore make other running programs more responsive).
- Depending on browser configuration and how the HTML for the exported applet is written, the program may not have keyboard focus when it first loads. This can cause confusion for the user if they begin typing but nothing happens. To handle such a situation, show the user a message that says, "Click inside the applet to begin" whenever the built-in Boolean variable `focused` is set to `false`.
- By redoing the preprocessing steps (Acquire, Parse, Filter), this same method can be used with the codes of other countries. Germany and the UK both use similar postal numbering systems, and data files for each are available online. The opengeodb project is one good place to start; see *http://sourceforge.net/projects/opengeodb*.
- For another variation of this project, I used town names instead of zip code digits. Typing "F" highlights all locations whose names begin with F, and typing "Fargo" shows the distribution of Fargos throughout the U.S. Also interesting about this modification is that it begins to show migration patterns for settlers from different countries as they moved across the U.S.
- And as long as we're looking at names, you can use street names as another option. It's a bit too much information to do the entire U.S., but it's possible to use the same principles to show a map of all the street names in a single state, and progressively select them based on names typed by the user. The street data can be found at the U.S. Census Bureau's TIGER/LINE site at *http://www.census.gov/geo/www/tiger*.
- What about area codes? I have often received requests for the same application applied to area codes or other sets of information that are geographically oriented. Such a system would not be consistent with the original question about how postal codes relate to geography, but it would still be a helpful and interesting tool.

- Also, from the tool front, it too would be possible to address other questions such as, "What locations are within 25 miles of 94133?" This can be done by converting latitude/longitude distance to miles and changing the interface to support selecting a zip code and a distance.
- One of the more interesting modifications would be to bring in additional data sets—whether satellite photography, geographic boundaries, interstate highways, or map images. By removing the projection conversion from the first step, any latitude/longitude-based data could be used as an additional layer. The nicest application would be to show different layers of information at each zoom level (e.g., showing state outlines once the user has typed a single digit, and after two digits, start to show city names or interstates). A good goal in any visualization project is to maintain the density of visual information at each level.

Having mastered the basic version of this project, there are lots of options for taking the project further. Have fun!