CHAPTER 3

Mapping

This chapter covers the basics of reading, displaying, and interacting with a data set. As an example, we'll use a map of the United States, and a set of data values for all 50 states. Drawing such a map is a simple enough task that could be done without programming—either with mapping software or by hand—but it gives us an example upon which to build. The process of designing with data involves a great deal of iteration: small changes that help your project evolve in usefulness and clarity. And as this project evolves through the course of the chapter, it will become clear how software can be used to create representations that automatically update themselves, or how interaction can be used to provide additional layers of information.

Drawing a Map

Some development environments separate work into projects; the equivalent term for Processing is a *sketch*. Start a new Processing sketch by selecting File → New.

For this example, we'll use a map of the United States to use as a background image. The map can be downloaded from *http://benfry.com/writing/map/map.png*.

Drag and drop the *map.png* file into the Processing editor window. A message at the bottom will appear confirming that the file has been added to the sketch. You can also add files by selecting Sketch → Add File. A sketch is organized as a folder, and all data files are placed in a subfolder named *data*. (The *data* folder is covered in Chapter 2.)

Then, enter the following code:

```
PImage mapImage;

void setup( ) {
  size(640, 400);
  mapImage = loadImage("map.png");
}
```

```
void draw( ) {
  background(255);
  image(mapImage, 0, 0);
}
```

Finally, click the Run button. Assuming everything was entered correctly, a map of the United States will appear in a new window.

Explanation of the Processing Code

Processing API functions are named to make their uses as obvious as possible. Method names, such as `loadImage( )`, convey the purpose of the calls in simple language. What you may need to get used to is dividing your code into functions such as `setup( )` and `draw( )`, which determine how the code is handled. After clicking the Run button, the `setup( )` method executes once. After `setup( )` has completed, the `draw( )` method runs repeatedly. Use the `setup( )` method to load images, fonts, and set initial values for variables. The `draw( )` method runs at 60 frames per second (or slower if it takes longer than 1/60th of a second to run the code inside the `draw( )` method); it can be used to update the screen to show animation or respond to mouse movement and other types of input.

Our first function calls are very basic. The `loadImage( )` function reads an image from the data folder (URLs or absolute paths also work). The `PImage` class is a container for image data, and the `image( )` command draws it to the screen at a specific location.

Locations on a Map

The next step is to specify some points on the map. To simplify this, a file containing the coordinates for the center of each state can be found at *http://benfry.com/writing/map/locations.tsv*.

In future chapters, we'll explore how this data is read. In the meantime, some code to read the location data file can be found at *http://benfry.com/writing/map/Table.pde*.

Add both of these files to your sketch the same way that you added the *map.png* file earlier.

The `Table` class is just two pages of code, and we'll get into its function later. In the meantime, suffice it to say that it reads a file as a grid of rows and columns. The class has methods to get an `int`, `float`, or `String` for a specific row and column. To get float values, for instance, use the following format:

```
table.getFloat(row, column)
```

Rows and columns are numbered starting at zero, so the column titles (if any) will be row 0, and the row titles will be column 0.

In the previous section, we saw how displaying a map in Processing is a two-step process:

1. Load the data.
2. Display the data in the desired format.

Displaying the centers of states follows the same pattern, although a little more code is involved:

1. Create `locationTable` and use the `locationTable.getFloat( )` function to read each location's coordinates (x and y values).
2. Draw a circle using those values. Because a circle, geometrically speaking, is just an ellipse whose width and height are the same, graphics libraries provide an ellipse-drawing function that covers circle drawing as well.

A new version of the code follows, with modifications highlighted:

```
PImage mapImage;
Table locationTable;
int rowCount;

void setup( ) {
  size(640, 400);
  mapImage = loadImage("map.png");
  // Make a data table from a file that contains
  // the coordinates of each state.
  locationTable = new Table("locations.tsv");
  // The row count will be used a lot, so store it globally.
  rowCount = locationTable.getRowCount( );
}

void draw( ) {
  background(255);
  image(mapImage, 0, 0);

  // Drawing attributes for the ellipses.
  smooth( );
  fill(192, 0, 0);
  noStroke( );

  // Loop through the rows of the locations file and draw the points.
  for (int row = 0; row < rowCount; row++) {
    float x = locationTable.getFloat(row, 1);  // column 1
    float y = locationTable.getFloat(row, 2);  // column 2
    ellipse(x, y, 9, 9);
  }
}
```

The `smooth( )`, `fill( )`, and `noStroke( )` functions apply to any drawing we subsequently do in the `draw( )` function. Later, we'll look at the aspects of drawing you can control; here I'll just mention that the `fill( )` function assigns red, green, and blue elements to the color. I've chosen to show all of the circles in red.

Figure 3-1 shows the map and points for each location.

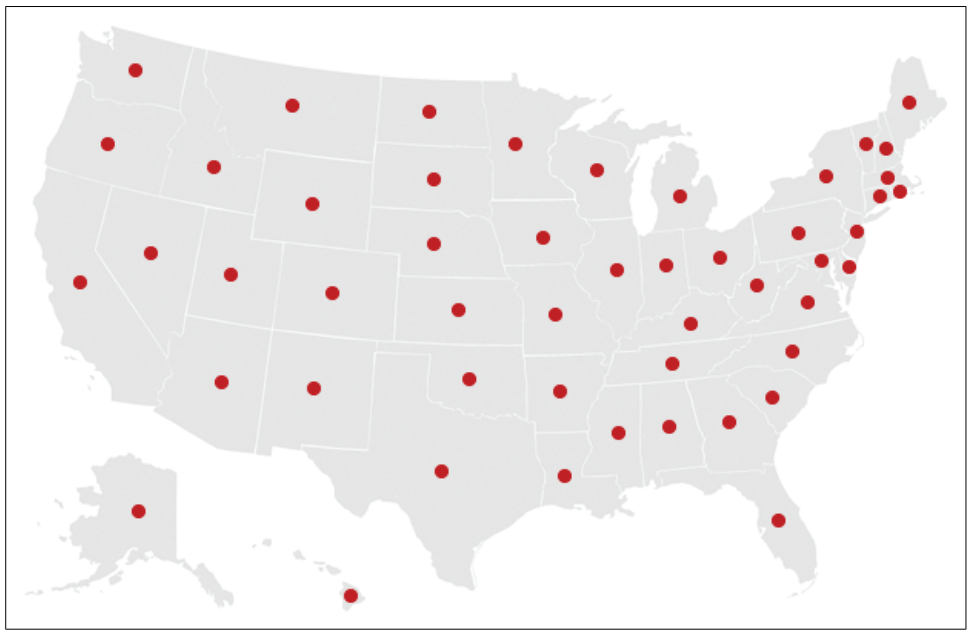


Figure 3-1. U.S. map and centers of states

Data on a Map

Next we want to load a set of values that will appear on the map itself. For this, we add another `Table` object and load the data from a file called *random.tsv*, available at *http://benfry.com/writing/map/random.tsv*.

It's always important to find the minimum and maximum values for the data, because that range will need to be mapped to other features (such as size or color) for display. To do this, use a `for` loop to walk through each line of the data table and check to see whether each value is bigger than the maximum found so far, or smaller than the minimum. To begin, the `dataMin` variable is set to `MAX_FLOAT`, a built-in value for the maximum possible `float` value. This ensures that `dataMin` will be replaced with the first value found in the table. The same is done for `dataMax`, by setting it to `MIN_FLOAT`. Using 0 instead of `MIN_FLOAT` and `MAX_FLOAT` will not work in cases where the minimum value in the data set is a positive number (e.g., 2.4) or the maximum is a negative number (e.g., –3.75).

The data table is loaded in the same fashion as the location data, and the code to find the minimum and maximum immediately follows:

```
PImage mapImage;
Table locationTable;
int rowCount;
```

```
Table dataTable;
float dataMin = MAX_FLOAT;
float dataMax = MIN_FLOAT;

void setup( ) {
  size(640, 400);
  mapImage = loadImage("map.png");
  locationTable = new Table("locations.tsv");
  rowCount = locationTable.getRowCount( );

  // Read the data table.
  dataTable = new Table("random.tsv");

  // Find the minimum and maximum values.
  for (int row = 0; row < rowCount; row++) {
    float value = dataTable.getFloat(row, 1);
    if (value > dataMax) {
      dataMax = value;
    }
    if (value < dataMin) {
      dataMin = value;
    }
  }
}
```

The other half of the program (shown later) draws a data point for each location. A `drawData( )` function is introduced, which takes x and y coordinates as parameters, along with an abbreviation for a state. The `drawData( )` function grabs the float value from column 1 based on a state abbreviation (which can be found in column 0).

The `getRowName( )` function gets the name of a particular row. This is just a convenience function because the row name is usually in column 0, so it's identical to `getString(row, 0)`. The row titles for this data set are the two-letter state abbreviations. In the modified example, `getRowName( )` is used to get the state abbreviation for each row of the data file.

The `getFloat( )` function can also use a row name instead of a row number, which simply matches the `String` supplied against the abbreviation found in column 0 of the *random.tsv* data file. The results are shown in Figure 3-2.

The rest of the program follows:

```
void draw( ) {
  background(255);
  image(mapImage, 0, 0);

  smooth( );
  fill(192, 0, 0);
  noStroke( );
```

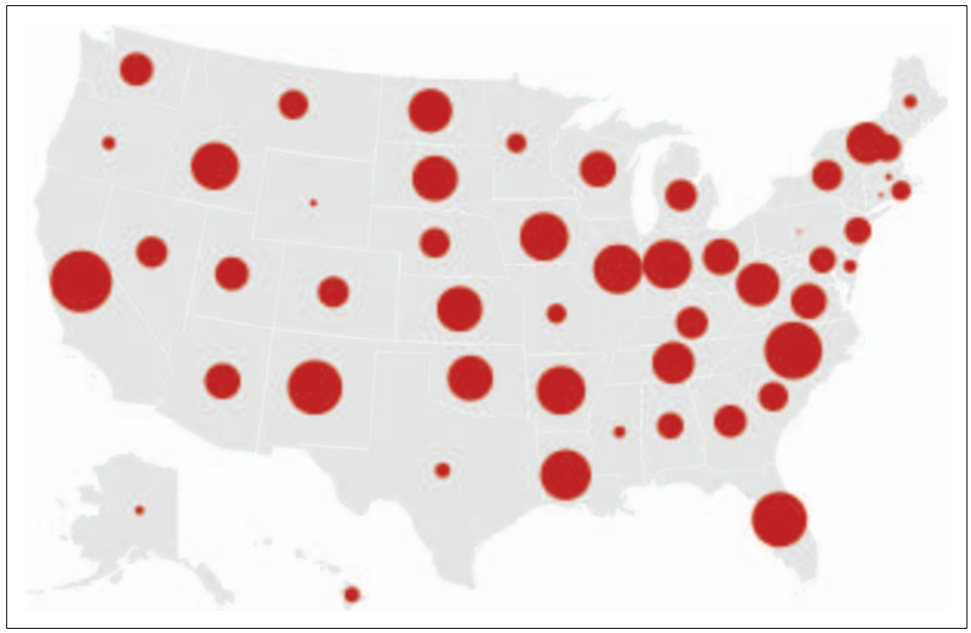


Figure 3-2. Varying data by size

```
  for (int row = 0; row < rowCount; row++) {
    String abbrev = dataTable.getRowName(row);
    float x = locationTable.getFloat(abbrev, 1);
    float y = locationTable.getFloat(abbrev, 2);
    drawData(x, y, abbrev);
  }
}

// Map the size of the ellipse to the data value
void drawData(float x, float y, String abbrev) {
  // Get data value for state
  float value = dataTable.getFloat(abbrev, 1);
  // Re-map the value to a number between 2 and 40
  float mapped = map(value, dataMin, dataMax, 2, 40);
  // Draw an ellipse for this item
  ellipse(x, y, mapped, mapped);
}
```

The `map( )` function converts numbers from one range to another. In this case, `value` is expected to be somewhere between `dataMin` and `dataMax`. Using `map( )` reproportions `value` to be a number between 2 and 40. The `map( )` function is useful for hiding the math involved in the conversion, which makes code quicker to write and easier to read. A lot of visualization problems revolve around mapping data from one range to another (e.g., from the min and max of the input data to the width or height of a plot), so the `map( )` method is used frequently in this book.

Another refinement option is to keep the ellipse the same size but interpolate between two different colors for high and low values. The `norm( )` function maps values from a user-specified range to a *normalized* range between 0.0 and 1.0. The `percent` value is a percentage of where `value` lies in the range from `dataMin` to `dataMax`. For instance, a `percent` value of 0.5 represents 50%, or halfway between `dataMin` and `dataMax`:

```
float percent = norm(value, dataMin, dataMax);
```

The `lerp( )` function converts a normalized value to another range (`norm( )` and `lerp( )` together make up the `map( )` function), and the `lerpColor( )` function does the same, except it interpolates between two colors. The syntax:

```
color between = lerpColor(color1, color2, percent)
```

returns a `between` value based on the percentage (a number between 0.0 and 0.1) specified. To make the colors interpolate between red and blue for low and high values, replace the `drawData( )` function with the following:

```
void drawData(float x, float y, String abbrev) {
  float value = dataTable.getFloat(abbrev, 1);
  float percent = norm(value, dataMin, dataMax);
  color between = lerpColor(#FF4422, #4422CC, percent);  // red to blue
  fill(between);
  ellipse(x, y, 15, 15);
}
```

Results are shown in Figure 3-3.

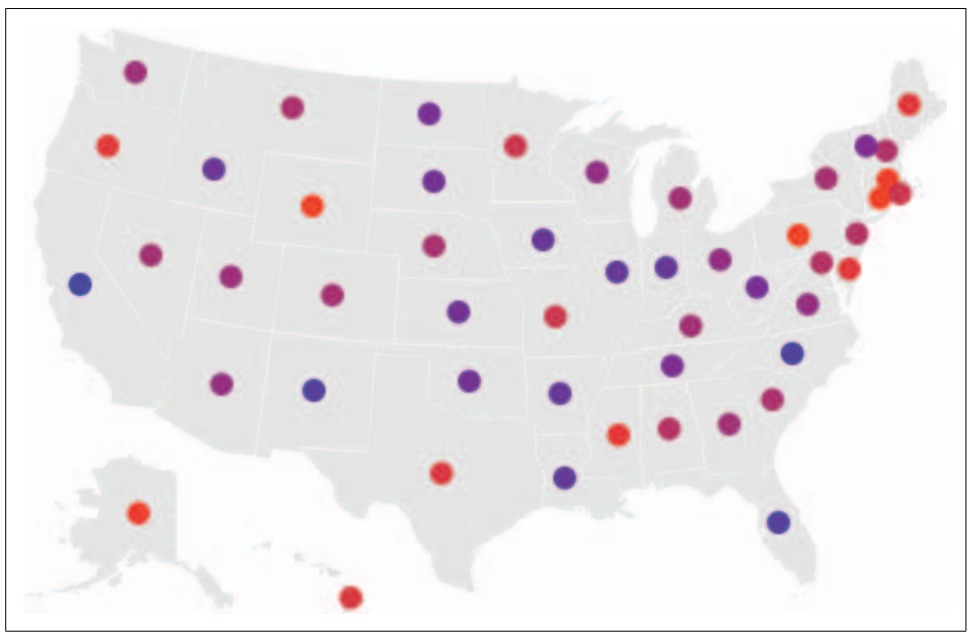


Figure 3-3. Varying data by color

This illustrates the problem with interpolating between two unrelated colors. Two separate colors make sense for positive and negative values (see the next section "Two-Sided Data Ranges" and Figure 3-6), but the idea of purple as an in-between value is confusing to read because it's difficult to say just how red or blue the values are—everything becomes a muddy purple. If using two different colors, a better option is to provide a neutral value in between the two colors, such as white or black.

On the other hand, to make color interpolation work, it's better to employ a pair of similar colors. For instance, blue and green provide an alternative gradation of values that is easier to read than the red-to-purple-to-blue range. Replace the `lerpColor` line with the following:

```
color between = lerpColor(#296F34, #61E2F0, percent);
```

and take a look at the result in Figure 3-4.

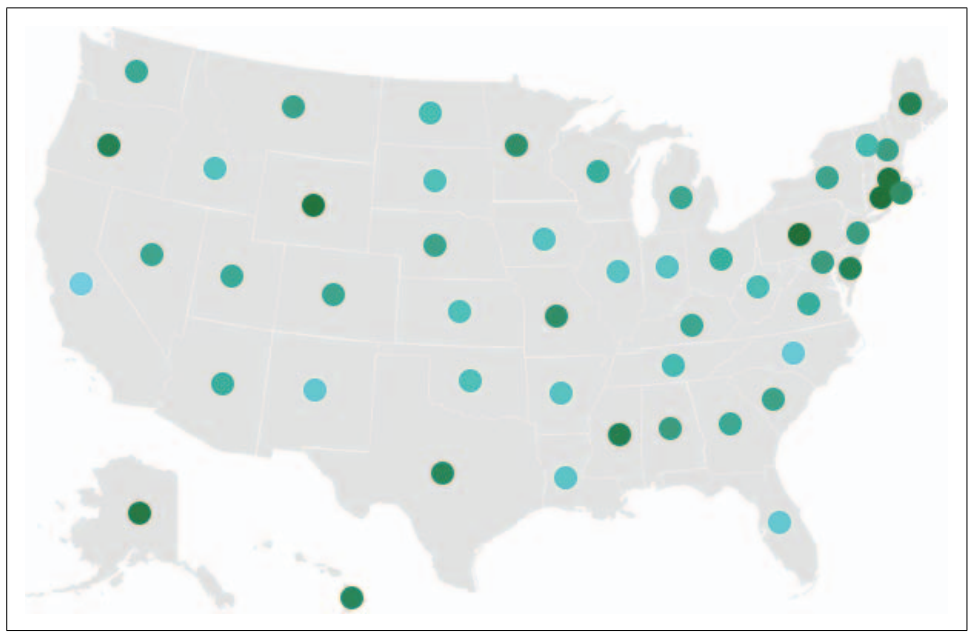


Figure 3-4. Varying data by color: better color choices

The interpolated values tend to be "muddy" because the interpolation is calculated in RGB color space. To preserve the saturation and brightness of colors, a better option is the HSB color space, particularly when dealing with colors that are similar in hue. A fourth parameter to `lerpColor( )` allows you to change the color space used for interpolation:

```
color between = lerpColor(#296F34, #61E2F0, percent, HSB);
```

Changing to the HSB color model improves brightness and contrast; see Figure 3-5.

RGB and HSB Color Spaces

The HSB color space defines colors based on hue, saturation, and brightness instead of the red, green, and blue values used in RGB. The RGB color space has more to do with how color is represented by computer screens than how we actually perceive color. Intermediate steps in each of the hue, saturation, and brightness components of a color provide better interpolation because each of the perceptual aspects of the color are broken apart—the shift in the hue component is separated from the shift in saturation and the shift in brightness.

In the RGB color space, a gray value occurs whenever the R, G, and B components are identical (or at least similar). In RGB space, the color halfway between orange (255, 128, 0) and light blue (0, 128, 255) is gray (128, 128, 128). So using `lerpColor()` in RGB mode would cause the orange to become more gray at each step, until it reaches gray; then, it would slowly move from gray to blue. Not too pleasing to look at.

On the other hand, RGB mode is preferred when there are significant changes in hue. For instance, if you begin at red and interpolate to green in HSB space, you'll iterate through all the spectrum colors in between: from red, to orange, then to yellow, and finally to green.

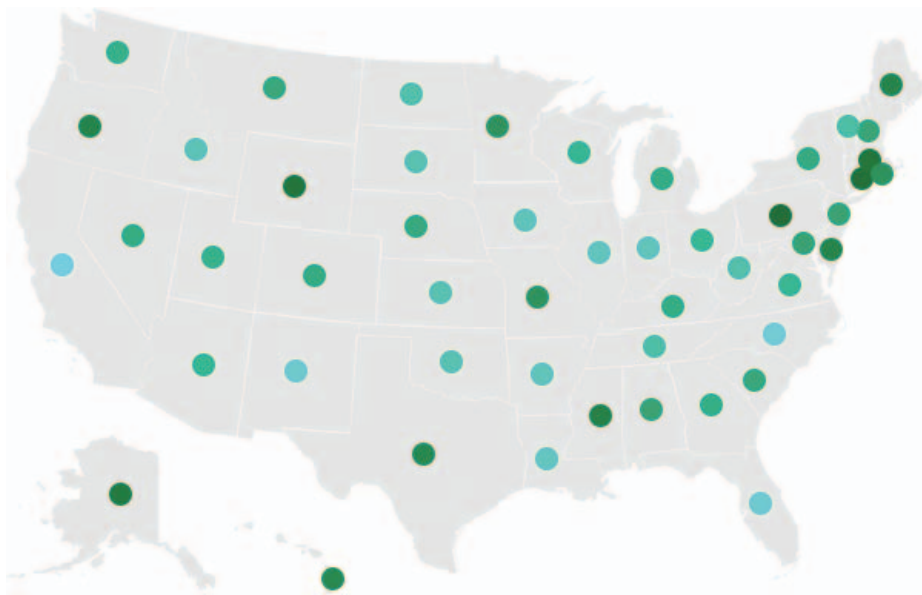


Figure 3-5. Varying data by color: better color space

But the color mix is still lacking, so next let's look at other options.

Two-Sided Data Ranges

Because the values in the data set are positive and negative, a better option would be to use separate colors for positive or negative while changing the size of each ellipse to reflect the range. The following replacement for `drawData( )` separates positive and negative values as well as indicating the magnitude (absolute value) of each value.

In this case, positive values are remapped from 0 through the maximum data value into a value between 3 and 30 for the diameter of the ellipse. Negative values are mapped in a similar fashion, where the most negative (`dataMin`) will be mapped to size 30, and the least negative (0) will be mapped to size 3. Positive values are drawn with blue ellipses and negative values with red; see Figure 3-6.

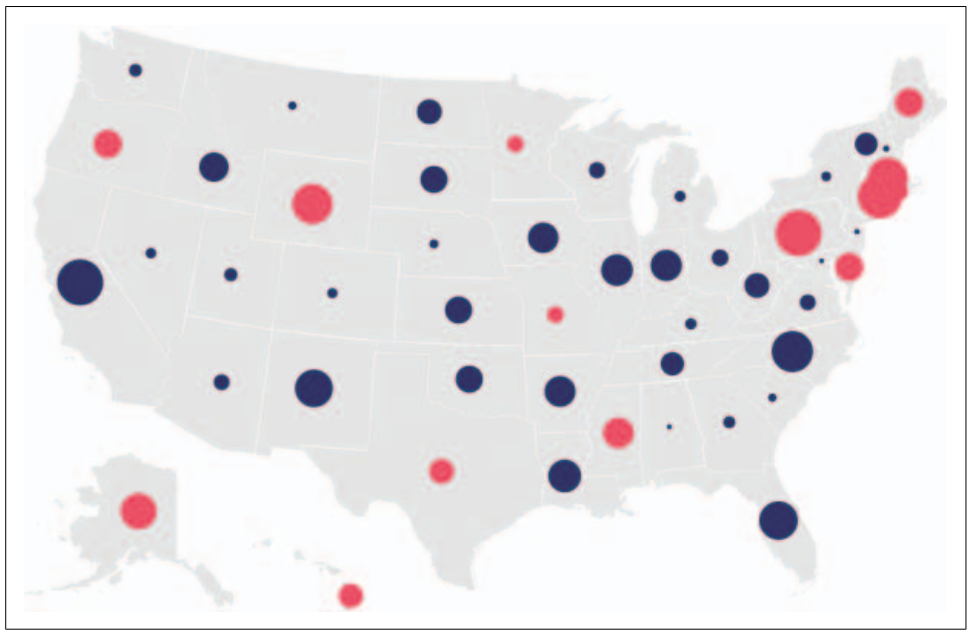


Figure 3-6. Magnitude and positive/negative

```
void drawData(float x, float y, String abbrev) {
  float value = dataTable.getFloat(abbrev, 1);
  float diameter = 0;
  if (value >= 0) {
    diameter = map(value, 0, dataMax, 3, 30);
    fill(#333366);  // blue
  } else {
    diameter = map(value, 0, dataMin, 3, 30);
    fill(#EC5166);  // red
  }
  ellipse(x, y, diameter, diameter);
}
```

Figure 3-6 is much easier to read and interpret than the previous representations; however, we've used up two visual features (size and color) on a single dimension of the data. For a simple data set such as this one, it's not a problem, but if we look at a pair of data values, we would want to use color for one dimension of the data and size for the other. In some cases, showing one variable two ways can be helpful for reinforcing the meaning of the values, but it's often done without consideration. The approach used in Figure 3-6 would be an appropriate solution if the difference between positive and negative values was our primary or secondary interest.

To preserve size for another aspect of the data, another option would be to map the *transparency* of the ellipses to their relative values. Transparency is also referred to as *alpha transparency* or usually just *alpha*. It's controlled by an optional second parameter to the `fill()` function; 0 means that the entire background shows though the object, whereas 255 means the object is totally opaque.

Yet another revision of the `drawData()` function shows how transparency is controlled; see Figure 3-7.

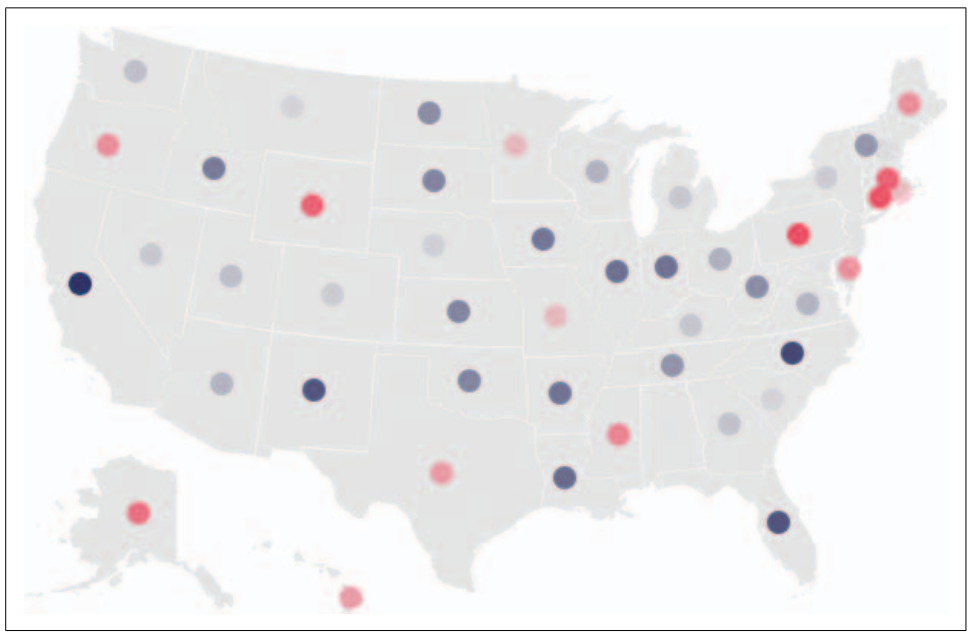


Figure 3-7. Magnitude and positive/negative using transparency

```
void drawData(float x, float y, String abbrev) {
  float value = dataTable.getFloat(abbrev, 1);
  if (value >= 0) {
    float a = map(value, 0, dataMax, 0, 255);
    fill(#333366, a);
  } else {
    float a = map(value, 0, dataMin, 0, 255);
```

```
      fill(#EC5166, a);
    }
    ellipse(x, y, 15, 15);
  }
```

Transparency can be useful for features at a glance, but it doesn't provide a lot of differentiation between the values (as can be seen here). For this particular data set, Figures 3-2 and 3-6 provide the best solutions in terms of visual design. The others are included to show alternatives and demonstrate the thinking process behind visual refinements.

Provide More Information with a Mouse Rollover (Interact)

Adding a small amount of interaction can help make a display more useful, and this feature shows how Processing makes mouse information readily available. As the mouse hovers above a particular state, additional information about that location can be revealed.

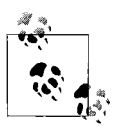
To show the extra information as text, a font is required. Use the Create Font option under the Tools menu. For this example, a typeface named Univers Bold was chosen from the list, and the size was set to 12. (Univers may not be available on your machine, so choose any font you'd like.)

Clicking OK adds a file named *Univers-Bold-12.vlw* to the *data* folder. Now when the sketch is exported as an applet or application, the font can be used on other machines, even if Univers is not installed on them.

Adding these lines to `setup()` sets the font:

```
PFont font = loadFont("Univers-Bold-12.vlw");
textFont(font);
```

Because fonts are loaded from files, `loadFont()` should be used only inside `setup()` (and not from the `draw()` method); otherwise, the font will be loaded repeatedly and slow down the program.



The location of the `loadFont()` call reiterates a valuable principle that may guide you when placing other function calls. The `setup()` method runs only once, when a browser or other program loads a sketch. The `draw()` method runs repeatedly (several times a second) so that sketches can be updated over time and animated.

The `textFont()` command sets the current font.

The rollover itself is handled by checking the distance between the mouse and the data point. If the mouse is within a certain range of the point, the text appears. The distance is calculated with the `dist()` function, and because this will calculate the radius between the location and the mouse, we issue `ellipseMode(RADIUS)` to draw the data points before `dist()`. Using the radius also helps because it can be used to

place the text above the data point: the distance to the bottom of the text will be the radius plus a few pixels of space. As the default `ellipseMode` is `DIAMETER` (and a radius is half a diameter), we'll adjust the preceding `map( )` function to use the values 1.5 and 15 instead of 3 and 30:

```
void drawData(float x, float y, String abbrev) {
  float value = dataTable.getFloat(abbrev, 1);
  float radius = 0;
  if (value >= 0) {
    radius = map(value, 0, dataMax, 1.5, 15);
    fill(#4422CC);  // blue
  } else {
    radius = map(value, 0, dataMin, 1.5, 15);
    fill(#FF4422);  // red
  }
  ellipseMode(RADIUS);
  ellipse(x, y, radius, radius);

  if (dist(x, y, mouseX, mouseY) < radius+2) {
    fill(0);
    textAlign(CENTER);
    // Show the data value and the state abbreviation in parentheses.
    text(value + " (" + abbrev + ")", x, y-radius-4);
  }
}
```

The parameters to `text( )` are a bit complex. The first argument—which is the text to display—combines the data itself (`value`) with the state abbreviation (`abbrev`), enclosing the abbreviation in parentheses. The second and third parameters specify the x and y position of the text. The vertical location is `y-radius-4`, which is the y-coordinate at the center of the circle, minus the radius to get to the edge of the circle, minus four more pixels.

Because the mouse cursor extends to the right and downward, we've placed the text above the circle to prevent the arrow from covering up the data itself. This is a general problem with rollovers that we'll return to later: rollover text or the means for selecting it can obscure the data underneath.

With the rollover, you might want to bring in additional types of data as well. An additional data file that maps abbreviations to the full state name can be found at *http://benfry.com/writing/map/names.tsv*.

The table in this data file will be used in conjunction with the others to look up names for the individual states. More commonly, this particular data would be found in the same data file as the others, but it's useful to introduce the idea of joining multiple sets of data. Joining data frequently is necessary, and the opportunity to combine data sets from different sources is a powerful aspect of data visualization.

First, add a `nameTable` declaration after the `locationTable` declaration at the beginning of the code:

```
Table nameTable;
```

Handling Mouse Interaction

In more sophisticated programs, it's common to package how elements are drawn into an individual *class*—a set of code that groups together related functions and variables. For instance, each state could be an instance of a class that contains the name, data value, and location on screen for each state. Mouse interaction would be handled by a function inside the class that checks whether the mouse location is near the location value for the state. We'll see this method used in later chapters.

The Processing API provides few high-level elements—there is currently no `Shape` class that handles such things automatically. Instead, the API is designed so that such classes are either unnecessary or simple for others to build into their own projects.

Next, load the `nameTable` along with the others inside the `setup()` function:

```
nameTable = new Table("names.tsv");
```

Finally, when drawing the text for the rollover, grab the full name from the table and display it:

```
String name = nameTable.getString(abbrev, 1);
text(name + " " + value, x, y-radius-4);
```

With the longer text showing, sometimes the parts of the text will appear behind other points because the states are all drawn in order. Because Massachusetts is drawn after Connecticut, the dot for Massachusetts will cover the rollover text for Connecticut. To change this behavior, first draw the data points for all the states, and then draw the rollover text for the current selection. When drawing each state, we'll check to see whether the mouse is in the vicinity, and if so, that state is a candidate for having its name drawn.

You might also notice that when using the mouse in the area of smaller states (such as those in the Northeast), several names will show up. To handle this, we'll keep track of the state that is closest to the mouse and show only the information for that state. As each state is drawn, we'll check whether the distance from the state's location to the mouse is the smallest found so far, and if so, store its X and Y position along with the text to show at that position. These three variables will be updated as each data point within range of the mouse is drawn. After the data points have finished drawing, the text can be drawn at that particular X and Y point.

The `mouseX` and `mouseY` variables are updated on each trip through `draw()`. Because the `draw()` method runs repeatedly (at around 60 frames per second), updates for the rollover happen almost instantaneously.

The top of the program remains unchanged, but the `draw()` and `drawData()` functions are replaced with the following:

```
// Global variables set in drawData( ) and read in draw( )
float closestDist;
String closestText;
float closestTextX;
float closestTextY;

void draw( ) {
  background(255);
  image(mapImage, 0, 0);

  closestDist = MAX_FLOAT;

  for (int row = 0; row < rowCount; row++) {
    String abbrev = dataTable.getRowName(row);
    float x = locationTable.getFloat(abbrev, 1);
    float y = locationTable.getFloat(abbrev, 2);
    drawData(x, y, abbrev);
  }

  // Use global variables set in drawData( )
  // to draw text related to closest circle.
  if (closestDist != MAX_FLOAT) {
    fill(0);
    textAlign(CENTER);
    text(closestText, closestTextX, closestTextY);
  }
}

void drawData(float x, float y, String abbrev) {
  float value = dataTable.getFloat(abbrev, 1);
  float radius = 0;
  if (value >= 0) {
    radius = map(value, 0, dataMax, 1.5, 15);
    fill(#4422CC);  // blue
  } else {
    radius = map(value, 0, dataMin, 1.5, 15);
    fill(#FF4422);  // red
  }
  ellipseMode(RADIUS);
  ellipse(x, y, radius, radius);

  float d = dist(x, y, mouseX, mouseY);
  // Because the following check is done each time a new
  // circle is drawn, we end up with the values of the
  // circle closest to the mouse.
  if ((d < radius + 2) && (d < closestDist)) {
    closestDist = d;
    String name = nameTable.getString(abbrev, 1);
    closestText = name + " " + value;
    closestTextX = x;
    closestTextY = y-radius-4;
  }
}
```

Updating Values over Time (Acquire, Mine)

A static map isn't particularly interesting, especially when there's the possibility of an interactive environment. In addition, data values are often dynamic. They might change every second, every hour, or every year, but in each case, we'll want to depict the change over time. For example, we can replace the data with random values each time a key is pressed. Again, because `draw()` is called repeatedly, the values shown on screen will update immediately.

The code in this section provides an initial illustration of how to handle user interaction, a theme I'll expand in the book.

One problem with changing data is that the minimum and maximum values need to stay fixed. You'll need to figure out the absolute values for each because recalculating `dataMin` and `dataMax` each time new data is found will make the data points appear out of proportion to the previous set of values. For this example, we'll set the minimum and maximum values to –10 and 10, when the variables are first declared at the beginning of the program:

```
Table dataTable;
float dataMin = -10;
float dataMax = 10;
```

This change means that the code to find the minimum and maximum values can be removed from the `setup()` method.

The following code builds on the previous step and adds a function to randomize the data values each time the Space bar is pressed:

```
void keyPressed() {
  if (key == ' ') {
    updateTable();
  }
}

void updateTable() {
  for (int row = 0; row < rowCount; row++) {
    float newValue = random(dataMin, dataMax);
    dataTable.setFloat(row, 1, newValue);
  }
}
```

The `random()` function takes a minimum and maximum value, and returns a value starting with the minimum, up to but not including the maximum. The `setFloat()` function overwrites the old value from the data table with the new random value.

When running this code, press the Space bar once to see a new set of data appear. You might also notice that the rollovers now look a bit silly because the randomized values might have six or seven digits of precision. This can be changed with the `nf()` function, which is used to format numbers for printing.

The basic form of `nf( )` specifies the number of digits to the left and right of the decimal point. Specifying 0 for either position means "any" number of digits. So, to allow any number of digits to the left of the decimal point and two digits to the right, the line that sets `closestText` changes from:

```
closestText = name + " " + value;
```

to the following:

```
closestText = name + " " + nf(value, 0, 2);
```

The name for `nf( )` is intentionally terse (if a bit cryptic) because it's almost always used in situations when it's being concatenated to another part of a `String`. Two related functions are:

`nfp( )`
: Requires each number to be shown with a + or – sign.

`nfs( )`
: Pads values with spaces to fill out the number of digits specified. This is useful in some situations for lining up values in vertical columns.

Because we care about positive and negative, `nfp( )` is probably best suited for our purpose. This turns "North Dakota 6.15234534" into "North Dakota +6.15", which is far more readable.

Instead of randomizing the data, `updateTable( )` could be used to load a new set of values from another data source, whether another file or a location online. For instance, the following URL can be used to read a new set of data from the Web:

http://benfry.com/writing/map/random.cgi

At this location is a simple Perl script that generates a new set of values and sends them over a CGI connection. The code follows, with comments for those not familiar with Perl syntax:

```
#!/usr/bin/perl

# An array of the 50 state abbreviations
@states = ('AL', 'AK', 'AZ', 'AR', 'CA', 'CO', 'CT', 'DE', 'FL', 'GA',
           'HI', 'ID', 'IL', 'IN', 'IA', 'KS', 'KY', 'LA', 'ME', 'MD',
           'MA', 'MI', 'MN', 'MS', 'MO', 'MT', 'NE', 'NV', 'NH', 'NJ',
           'NM', 'NY', 'NC', 'ND', 'OH', 'OK', 'OR', 'PA', 'RI', 'SC',
           'SD', 'TN', 'TX', 'UT', 'VT', 'VA', 'WA', 'WV', 'WI', 'WY');

# A CGI script must identify the type of data it's sending;
# this line specifies that plain text data will follow.
print "Content-type: text/plain\n\n";

# Loop through each of the state abbreviations in the array.
foreach $state (@states) {
```

```
    # Pick a random number between -10 and 10. (rand( ) returns a
    # number between 0 and 1; multiply that by 20 and subtract 10.)
    $r = (rand( ) * 20) - 10;

    # Print the state name, followed by a tab,
    # then the random value, followed by a new line.
    print "$state\t$r\n";
}
```

To use this URL, the code for the `updateTable( )` function changes to the following:

```
void updateTable( ) {
  dataTable = new Table("http://benfry.com/writing/map/random.cgi");
}
```

Even though this script just produces randomized data, the same model could be used in actual practice, where a data set is generated online—perhaps from a database or something else that is accessible only through a network connection.

Smooth Interpolation of Values over Time (Refine)

When updating data, it's important to show users the transition over time. Interpolating between values helps users track where the changes occur and provides context for the change as it happens. The way to think about interpolation is that your data values are never "equal" to some number; rather, they're always "becoming" or "transitioning to" another value.

For this, we use another class called an `Integrator`. The contents of the code will be explained shortly, but for the time being, it can be downloaded from *http://benfry.com/writing/map/Integrator.pde*.

This code implements a simple physics-based interpolator. A force is exerted by which a value can "target" another, in the manner of a spring (more about this later). The important thing to understand is that an `Integrator` object represents a single data value. When the `Integrator` is constructed, an initial value is set:

```
Integrator changingNumber = new Integrator(4);
```

To make the value transition from 4 (its initial value) to –2, use the `target( )` method:

```
changingNumber.target(-2);
```

This has no effect yet on the display. For the value to update, you must call the `update( )` method of the `Integrator`:

```
changingNumber.update( );
```

Usually this is done at the beginning of `draw( )`. The `target( )` method is called whenever a state changes, and the constructor is used inside `setup( )`. The `Integrator` has a `value` field that always holds its current value. To set the diameter of an ellipse based on the changing value, a line like this would be used inside `draw( )`:

```
ellipse(x, y, changingNumber.value, changingNumber.value);
```

Because our state example uses 50 values, we need to create an array of `Integrator` objects inside `setup()`, update each of them at the beginning of `setup()`, and `target()` them to new values each time the display changes, effectively producing an animation. Instead of using `getFloat()` to read values from the `dataTable` object, the `dataTable` object will be used to `target()` the `Integrator` list.

The modified code looks like this:

```
PImage mapImage;
Table locationTable;
Table nameTable;
int rowCount;

Table dataTable;
float dataMin = -10;
float dataMax = 10;

Integrator[] interpolators;

void setup() {
  size(640, 400);
  mapImage = loadImage("map.png");
  locationTable = new Table("locations.tsv");
  nameTable = new Table("names.tsv");
  rowCount = locationTable.getRowCount();

  dataTable = new Table("random.tsv");

  // Setup: load initial values into the Integrator.
  interpolators = new Integrator[rowCount];
  for (int row = 0; row < rowCount; row++) {
    float initialValue = dataTable.getFloat(row, 1);
    interpolators[row] = new Integrator(initialValue);
  }

  PFont font = loadFont("Univers-Bold-12.vlw");
  textFont(font);

  smooth();
  noStroke();
}

float closestDist;
String closestText;
float closestTextX;
float closestTextY;


void draw() {
  background(255);
  image(mapImage, 0, 0);
```

```
    // Draw: update the Integrator with the current values,
    // which are either those from the setup( ) function
    // or those loaded by the target( ) function issued in
    // updateTable( ).
    for (int row = 0; row < rowCount; row++) {
      interpolators[row].update( );
    }

    closestDist = MAX_FLOAT;

    for (int row = 0; row < rowCount; row++) {
      String abbrev = dataTable.getRowName(row);
      float x = locationTable.getFloat(abbrev, 1);
      float y = locationTable.getFloat(abbrev, 2);
      drawData(x, y, abbrev);
    }

    if (closestDist != MAX_FLOAT) {
      fill(0);
      textAlign(CENTER);
      text(closestText, closestTextX, closestTextY);
    }
  }

  void drawData(float x, float y, String abbrev) {
    // Figure out what row this is.
    int row = dataTable.getRowIndex(abbrev);
    // Get the current value.
    float value = interpolators[row].value;

    float radius = 0;
    if (value >= 0) {
      radius = map(value, 0, dataMax, 1.5, 15);
      fill(#4422CC);  // blue
    } else {
      radius = map(value, 0, dataMin, 1.5, 15);
      fill(#FF4422);  // red
    }
    ellipseMode(RADIUS);
    ellipse(x, y, radius, radius);

    float d = dist(x, y, mouseX, mouseY);
    if ((d < radius + 2) && (d < closestDist)) {
      closestDist = d;
      String name = nameTable.getString(abbrev, 1);
      // Use target (not current) value for showing the data point.
      String val = nfp(interpolators[row].target, 0, 2);
      closestText = name + " " + val;
      closestTextX = x;
      closestTextY = y-radius-4;
    }
  }
```

```
void keyPressed( ) {
  if (key == ' ') {
    updateTable( );
  }
}

void updateTable( ) {
  for (int row = 0; row < rowCount; row++) {
    float newValue = random(-10, 10);
    interpolators[row].target(newValue);
  }
}
```

The changes will transition very quickly, and there are two ways to handle this. The first is to adjust the frame rate of your application, which may be very high. By default, the frame rate is capped to 60 frames per second (fps). When building animated graphics, it's important to keep an eye on the frame rate to avoid situations in which a faster computer runs your code too quickly. Simply setting the maximum frame rate lower results in a more visually pleasing presentation. This line, added to the end of `setup( )`, sets the maximum to 30 fps:

```
frameRate(30);
```

Another option is to use the `Integrator` class's own parameters. We mentioned that the `Integrator` class uses math for simple physics that simulate a spring. The target value is the resting length of the spring. The other parameters are defined in terms of physical properties, which include the damping (how much friction exists to prevent the changes from being too wobbly) and the degree of attraction (how quickly the value will become another). You can set the damping and attraction in the constructor. The default damping is 0.5 and the attraction is 0.2. Even without modifying the `frameRate( )` setting, changing the constructor can make things move much more slowly:

```
interpolators[row] = new Integrator(initialValue, 0.5, 0.01);
```

Cutting down the damping makes things bouncy:

```
interpolators[row] = new Integrator(initialValue, 0.9, 0.1);
```

Using Your Own Data

The file format presented in this chapter is straightforward, so try replacing the *random.tsv* file with your own data based on the 50 states. It's remarkably easy to plot your own values to individual locations. You'll probably still use the `map( )` function, but you don't have to use ellipses or colors to plot your data points. You could draw an image at each location, varying its size based on the data. Or some points could be hidden or reorganize themselves in various ways. The points might refer to anything from chain coffee shops per capita to poverty levels in each state.

Taking Data from the User

Not everyone wants to employ data relating to the United States, but the same technique is sound for any type of data mapped to particular points. In later chapters, we'll get into mapping latitude and longitude coordinates, as well as using shape data for locations, but even the simple example presented in this chapter can be used in many other ways.

The following code reads from the *names.tsv* file and asks the user to indicate a location for each in turn, by clicking the mouse where the user wants the data to be placed. Start this example as a separate sketch. It requires a *map.png* file, a *names.tsv* file, and the *Table.pde* file used throughout this chapter. The map image and names file can be replaced with data of your choice, and this code produces a *locations.tsv* file that can be added to the *data* folder of the new sketch:

```
PImage mapImage;
Table nameTable;

int currentRow = -1;
PrintWriter writer;

void setup( ) {
  size(640, 400);
  mapImage = loadImage("map.png");
  nameTable = new Table("names.tsv");
  writer = createWriter("locations.tsv");
  cursor(CROSS);  // make easier to pinpoint a location
  println("Click the mouse to begin.");
}

void draw( ) {
  image(mapImage, 0, 0);
}

void mousePressed( ) {
  if (currentRow != -1) {
    String abbrev = nameTable.getRowName(currentRow);
    writer.println(abbrev + "\t" + mouseX + "\t" + mouseY);
  }

  currentRow++;
  if (currentRow == nameTable.getRowCount( )) {
    // Close the file and finish.
    writer.flush( );
    writer.close( );
    exit( );
  } else {
    // Ask for the next coordinate.
    String name = nameTable.getString(currentRow, 1);
    println("Choose location for " + name + ".");
  }
}
```

Next Steps

In this chapter, we learned the basics of reading, displaying, and interacting with a data set. The chapters that follow delve into far more sophisticated aspects of each, but all of them build on the basic skills you've picked up here.