Beginning

Google Maps Applications with Rails and Ajax

From Novice to Professional

Build awesome Rails-driven mapping applications using the powerful Google Maps API

Andre Lewis, Michael Purvis, Jeffrey Sambells, and Cameron Turner

Apress®

Beginning Google Maps Applications with Rails and Ajax

From Novice to Professional

Andre Lewis, Michael Purvis, Jeffrey Sambells, and Cameron Turner

Apress®

Beginning Google Maps Applications with Rails and Ajax: From Novice to Professional

ISBN-13 (pbk): 978-1-59059-787-3

ISBN-10 (pbk): 1-59059-787-7

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jason Gilmore
Technical Reviewer: Sam Aaron
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Matt Wade
Project Manager: Sofia Marchant
Copy Edit Manager: Nicole Flores
Copy Editor: Jennifer Whipple
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Cheu
Compositor: Kinetic Publishing Services, LLC
Proofreader: April Eddy
Indexer: Beth Palmer
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at the official web site, `http://googlemapsbook.com`.

Contents at a Glance

Contents

About the Authors

ANDRE LEWIS became interested in Google Maps when he set out to create a simple online list of local Wi-Fi cafes. That effort subsequently grew into an active community-driven site at `http://hotspotr.com`. Since then, he has developed numerous tools and techniques for map-based applications using Ruby on Rails.

While geographically oriented applications remain a favorite subject, Andre enjoys working with all kinds of technologies. He has architected systems to support millions of daily page views, but he also likes getting the JavaScript and CSS "just right" on a web page. He currently works freelance, consulting on Web 2.0 technologies and developing Ruby on Rails applications. He blogs at `http://earthcode.com` and speaks periodically at Bay Area technology groups.

Andre lives and works in San Francisco. When he's not working with clients or exploring the latest technologies, he likes to mountain bike, camp, and ride his motorcycle.

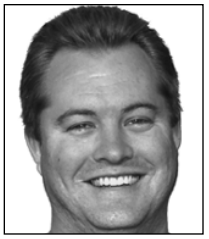
MICHAEL PURVIS is a mechatronics engineering student at the University of Waterloo, Ontario. Prior to discovering web scripting, he was busy with projects of other kinds, such as making a LEGO Mindstorms kit play the game Connect 4. After the PHP edition of this book was published, he was hired by Google for a four-month internship in the New York City office. He also continues to maintain an active community site for classmates, built from home-brewed extensions to PunBB and MediaWiki.

He has written about CSS for the Position Is Everything web site, and occasionally participates in the css-discuss mailing list. He loves simplicity, but cannot resist the charm of those few problems that do require negative margins and devious float tricks. Discussion of these and other nontechnical topics appears occasionally on his blog at `http://uwmike.com`.

Offline, he enjoys cooking, writing, cycling, and social dancing. He has worked with We-Create Inc. on a number of exciting PHP-based projects and has a strong interest in independent web standards.

JEFFREY SAMBELLS is a graphic designer and self-taught web applications developer best known for his unique ability to merge the visual world of graphics with the mental realm of code. With a bachelor of technology degree in graphic communications management and a minor in multimedia, Jeffrey was originally trained for the traditional paper-and-ink printing industry, but he soon realized the world of pixels and code was where his ideas would prosper. In 1999, he cofounded We-Create Inc., an Internet software company based in Waterloo, Ontario, which began many long nights of challenging and creative innovation. Currently, as director of research and development for We-Create, Jeffrey is responsible for investigating new and emerging Internet technologies and integrating them using web standards–compliant methods. In late 2005, he also became a Zend Certified Engineer.

When not playing at the office, Jeffrey enjoys a variety of hobbies, from photography to woodworking. When the opportunity arises, he also enjoys floating in a canoe on the lakes of Algonquin Provincial Park or going on an adventurous, map-free drive with his wife. Jeffrey also maintains a personal web site at `http://JeffreySambells.com`, where he shares thoughts, ideas, and opinions about web technologies, photography, design, and more. He lives in Ontario, Canada, eh, with his wife, Stephanie, daughter, Addison, and their little dog, Milo.



CAMERON TURNER has been programming computers since his first VIC 20 at age seven. He has been developing interactive web sites since 1994. In 1999, he cofounded We-Create Inc., which specializes in Internet software development. He is now the company's chief technology officer. Cam obtained his honors degree in computer science from the University of Waterloo with specialization in applied cryptography, database design, and computer security.

Since the PHP edition of this book was published, Cam started giving technology-related talks and lectures to companies and associations based in and around Waterloo, Ontario. Topics and interests range from Google Maps (of course) to search engine optimization as well as other topics relating to professional web software development.

Cam lives in Canada's technology capital of Waterloo with his wife, Tanya, son, Owen, and dog, Katie. His hobbies include geocaching, biking, hiking, water skiing, and painting. He maintains a low-volume personal blog at `http://CamTurner.com`, discussing nontechnical topics, thoughts, theories, and family life.

About the Technical Reviewer

SAM AARON is a Ph.D. student at the School of Computing Science at Newcastle University in the U.K. He is currently finishing off his thesis on the subject of interest management. He is both a Ruby and a Rails fanatic, and as such is actively involved in using and raising awareness of these wonderful technologies. He founded and organizes the local Ruby and Rails User Group—ncl.rb, which attracts more than 20 people every month. He is using Rails to build a web-based decision support tool for Newcastle University's transport department and is constantly looking for excuses to include as many of the exciting new Rails advances into his projects as possible.

Sam lives with his beautiful girlfriend, Susanna, on the quayside in Newcastle-upon-Tyne, England. He loves watching the birds fly over the river from his window. He spends his working days hacking away on his PowerBook listening to strange electronic music. When not working, he likes to get out of the city and relax. He loves camping, climbing mountains, and power kiting. When at home he's often found either playing the piano or table football. He doesn't own a car and can't even drive, preferring instead to cycle everywhere—especially long-distance expeditions with good friends.

When Sam finishes his Ph.D., he plans to start a web development, training, and consultancy company focusing on Ruby and Rails. He is currently in talks with Newcastle College with respect to it including Ruby and Rails content within its taught courses. If you're interested in finding out what Sam's up to today, just head along to his blog: `http://sam.aaron.name`.

PART 1

Your First Google Maps

CHAPTER 1

Google Maps and Rails

The last year or so has been an incredibly exciting time for web developers. New tools have come out that make web development easier, more productive, and more fun. A slew of new APIs are available that let you glue together data and services in interesting ways. As developers, we are more empowered with new and interesting technologies than ever before.

This book focuses on one API that has had a particularly profound impact: the Google Maps API. Since you have this book in hand, you're probably already convinced of Google Maps' importance. In case you need a reminder, however, visit Google Maps Mania (`http://googlemapsmania.blogspot.com`) for a view into the sprawling culture of innovation that Google Maps has fostered in the development community. The Google Maps API has spawned a whole class of web-based applications that would have been impossible to create without it.

You're going to use the Google Maps API on a platform that has inspired an equally fervent following: Ruby on Rails. The Rails framework facilitates radical improvements in productivity within its niche: database-backed web applications. Rails is intuitive, powerful, and free. Together, Rails and Google Maps enable you, the developer, to build impressive web-based applications that would have been difficult or impossible two years ago.

Over the course of the coming chapters, you're going to move from simple tasks involving markers and geocoding to more advanced topics, such as how to acquire data, present many data points, and provide a useful and attractive user interface.

There are many reasons why Ruby on Rails is an ideal platform to work with Google Maps. Rails makes it trivial to produce and consume XML, which Google Maps uses extensively. Rails also has built-in support for JSON (JavaScript Object Notation), a concise format for passing structured data from server to browser. Finally, Ruby has some excellent libraries for screen-scraping, which we will employ in later chapters.

We are assuming that you are coming to this book with a certain amount of Rails experience. You probably already have Ruby and Rails installed in a development environment and know how to get an application up and running. If not, don't fear: we list some resources in the sidebar "Just Getting Started with Ruby and Rails?" near the end of this chapter to help you get rolling on Rails. Whatever your current skill level vis-à-vis Rails, this book will get you in the mapping game and tell you everything you need to create killer maps applications. With the power of the Rails framework, the Ruby language, and the Google Maps API, you will command a development toolkit to be reckoned with.

We know you're eager to get started on a map project, but before we dig into the code, we want to show you two simple ways of creating ultraquickie maps: with KML (Keyhole Markup Language) files and through the Wayfaring map site.

Both these approaches are stepping stones; we will use them as easy introductions to the world of Google Maps. In Chapter 2, you will begin digging into code, which will of course lead to much greater flexibility and sophistication in what you can build.

KML: Your First Map

KML is one of the easiest methods to get your own markers and content displayed on a Google map. As you would expect, the ease is at the expense of flexibility, but it's still a great way to get started. In June 2006, Google announced that its official maps site would support the plotting of KML files. You can simply plug a URL into the search box, and Google Maps will show whatever locations are contained in the KML file specified by the URL. We aren't going to go in depth on this, but we've made a quick example to show you how powerful the KML method is, even if it is simple.

■ **Note** The name *Keyhole Markup Language* is a nod to both its XML structure and Google Earth's heritage as an application called Keyhole. Keyhole was acquired by Google in late 2004.

We created a file called `toronto.kml` and placed the contents of Listing 1-1 in it. The paragraph blurbs are borrowed from Wikipedia, and the coordinates were discovered by manually finding the locations on Google Maps.

Listing 1-1. *A Sample KML File*

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.google.com/earth/kml/2">
<Document>
  <name>toronto.kml</name>
  <Placemark>
    <name>CN Tower</name>
    <description> The CN Tower (Canada's National Tower, Canadian National Tower),
    at 553.33 metres (1,815 ft., 5 inches) is the tallest ➥
    freestanding structure on land.
    It is located in the city of Toronto, Ontario, Canada, and is considered the
    signature icon of the city. The CN Tower attracts close to two million visitors
    annually.

    http://en.wikipedia.org/wiki/CN_Tower</description>
    <Point>
      <coordinates>-79.386864,43.642426</coordinates>
    </Point>
  </Placemark>
</Document>
</kml>
```

In the actual file (located at `http://book.earthcode.com/kml/toronto.kml`), we included two more `Placemark` elements that point to other well-known buildings in Toronto. To view this on Google Maps, paste the previous URL into the Google Maps search field. Alternatively, you can just visit the following link: `http://maps.google.com/maps?f=q&hl=en&q=http://book.earthcode.com/kml/toronto.kml`.

Figure 1-1 shows what it looks like.

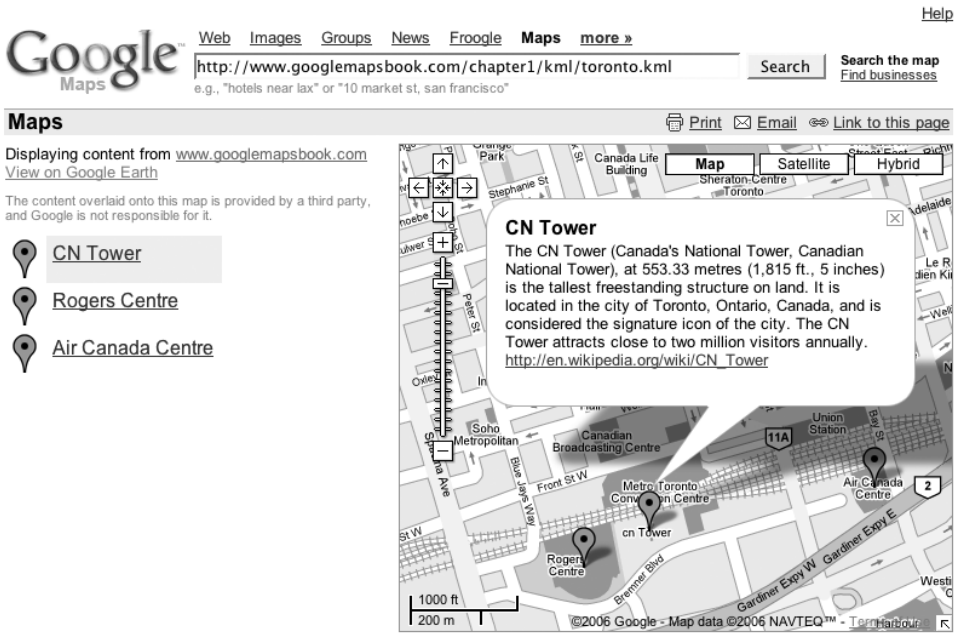


Figure 1-1. *A custom KML data file being displayed at maps.google.com*

Now, is that a quick result or what? Indeed, if all you need to do is show a bunch of locations, it's possible that a KML file will serve your purpose. If you're trying to link to your favorite fishing spots, you could make up a KML file, host it somewhere for free, and be finished.

But that wouldn't be any fun, would it? After all, as cool as the KML mapping is, it doesn't actually offer any interactivity to the user. In fact, most of the examples you'll work through in Chapter 2 are just replicating the functionality that Google provides here out of the box. But once you get to Chapter 3, you'll start to see things that you can do *only* when you harness the full power of the Google Maps API.

Before moving on, though, we'll take a look at one other way of getting a map online quickly.

Wayfaring: Your Second Map

A number of services out there let you publish free maps of quick plotted-by-hand data. One of these, which we'll demonstrate here, is Wayfaring, shown in Figure 1-2. Wayfaring has received attention and praise for its classy design and community features (such as commenting and shared locations). Wayfaring is also built using Ruby on Rails.

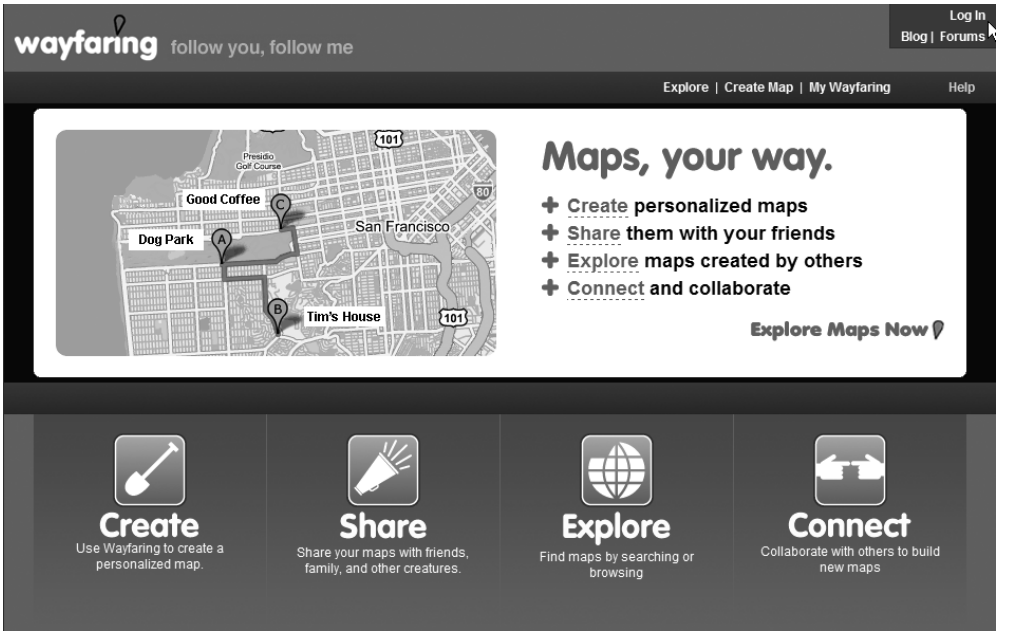


Figure 1-2. *Wayfaring home page*

Wayfaring is a mapping service that uses the Google Maps API and allows users to quickly create maps of anything they like. For example, some people make maps of their vacations; others have identified interesting aspects of their hometown or city. We'll walk you through making a quick map of an imaginary trip to the Googleplex in Mountain View, California.

Point your browser at `http://www.wayfaring.com` and follow the links to sign up for an account (clicking Log In will bring up the option to create a new account). Once you've created and activated your account, you can begin building your map by clicking the Create Map link in the upper right.

Adding the First Point

Let's start by adding the home airport for our imaginary journey. We're going to use Lester B. Pearson International Airport in Toronto, Ontario, Canada, but you could use the airport closest to you. Since Pearson is an international location (outside the United States), you need to drag and zoom the map view until you find it. If you're in the United States, you can use the nifty Jump To feature to search by text string. Figure 1-3 shows Pearson nicely centered and zoomed.

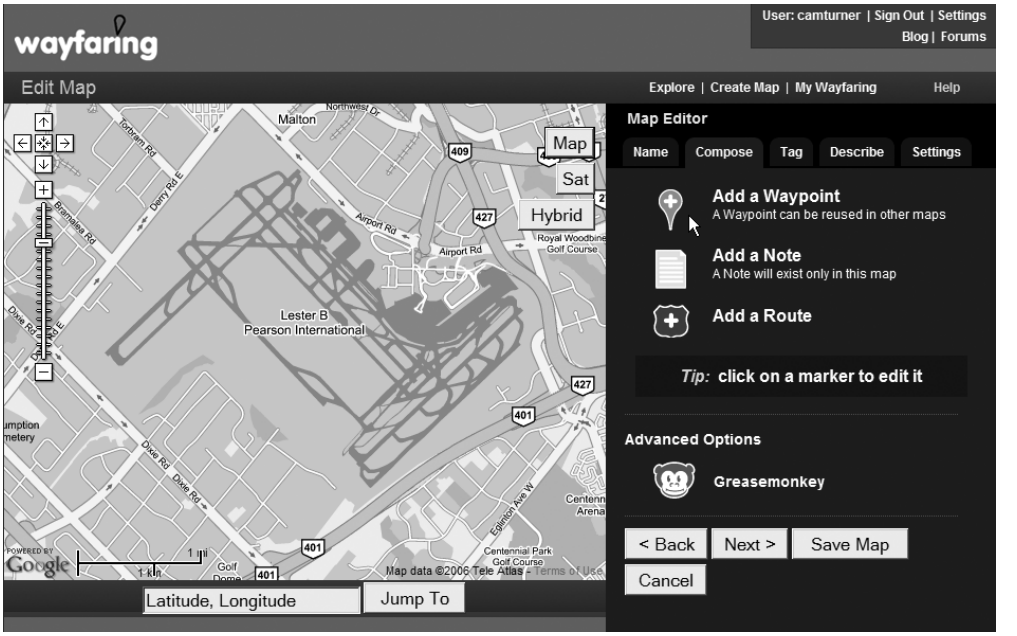


Figure 1-3. *Lester B. Pearson International Airport, Toronto, Ontario*

Once you've found your airport, you can click Next and name the map. Click Next again (after naming your map), and you should be back at the main Map Editor screen.

Select Add a waypoint from the list of options on the right. You'll be prompted to name the waypoint. We'll call ours *Lester B. Pearson International Airport*. However, as you type, you'll find that Wayfaring suggests this exact name. This means that someone else on some other map has already used this waypoint, and the system is giving you a choice of using their point or making one of your own. It's a safe bet that most of the airports you could fly from are already in Wayfaring, so feel free to use the suggested one if you would like. For the sake of the learning experience, let's quickly make our own. Click Next to continue.

The next two screens ask you to tag and describe this point in order to make your map more searchable for other members. We'll add the tags "airport Toronto Ontario Canada" and give it a simple description. Finally, click Done to commit the point to the map, which returns you to the Map Editor screen.

Adding the Flight Route

The next element you're going to add to your map is a *route*. A route is a line made up of as many points as you like. We'll use two routes in this example. The first will be a straight line between the two airports to get a rough idea of the distance the plane will have to travel to get us to Google's headquarters. The second will be used to plot the driving path we intend to take between the San Francisco airport and the Googleplex.

To begin, click Add a Route, name the route (something like *airplane trip*), and then click your airport. A small white dot appears on the place you click. This is the first point on your line. Now zoom out, scroll over to California, and zoom in on San Francisco. The airport is on the west side of the bay. Click the airport here, too. As you can see in Figure 1-4, a second white dot appears on the airport, and a blue line connects the two points. You can see the distance of the flight on the right side of the screen, underneath the route label. Wow, the flight seems to have been more than 2,000 miles! If you make a mistake and accidentally click on the map a few extra times (thereby creating extraneous midway points) in the process of getting to San Francisco, you can use the Undo Last option. Otherwise, click Save.

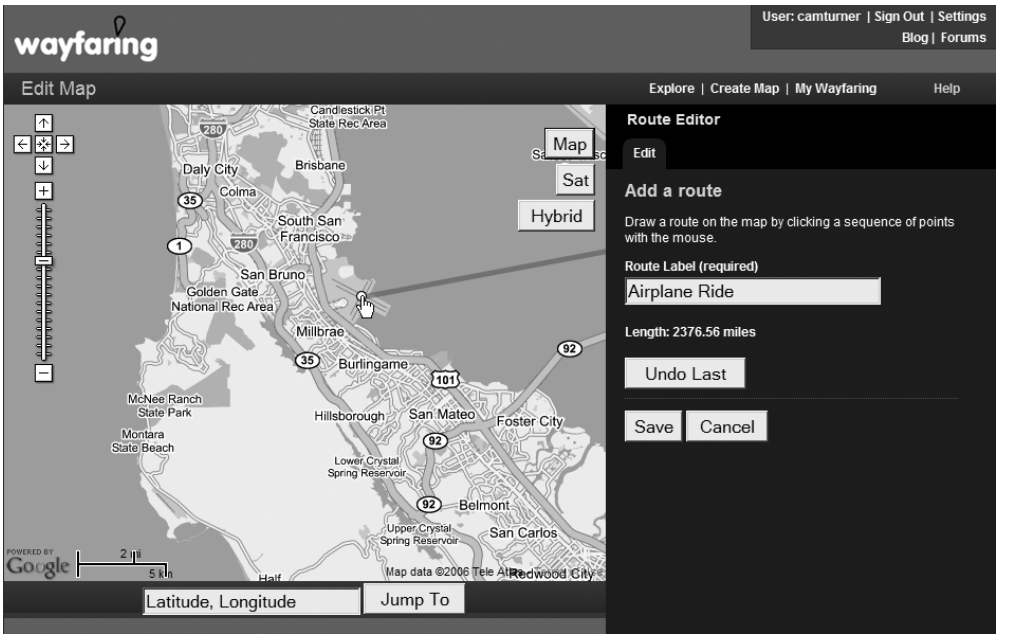


Figure 1-4. *Your flight landing at San Francisco International Airport*

Adding the Destination Point

Now that you're in San Francisco, let's figure out how to get to the Googleplex directly. Click Add a Waypoint. Your destination is Google, so the new point is called *The Googleplex*. Use the address box feature to jump directly to 1600 Amphitheatre Parkway, Mountain View, California, 94043. Wayfaring is able to determine latitude and longitude from an address via a process called *geocoding*, which you'll see a lot more of in Chapter 4.

To confirm you're in the right place, click the Sat button on the top-right corner of the map to switch it over to satellite mode. You should see something resembling Figure 1-5.

Figure 1-5. *The Googleplex*

Adding a Driving Route

Next, let's figure out how far the drive is. Routes don't really have a starting and ending point in Wayfaring from a visual point of view, so you can start your route from the Googleplex and work your way backward. Switch back into Map mode or Hybrid mode so you can see the roads more clearly. From the Map Editor screen, select Add a Route and click the point you just added. Use 10 to 20 dots to carefully trace the trip from Mountain View back up the Bayshore Freeway (U.S. Highway 101) to the airport. You'll end up with about 23 miles of driving, as shown in Figure 1-6.

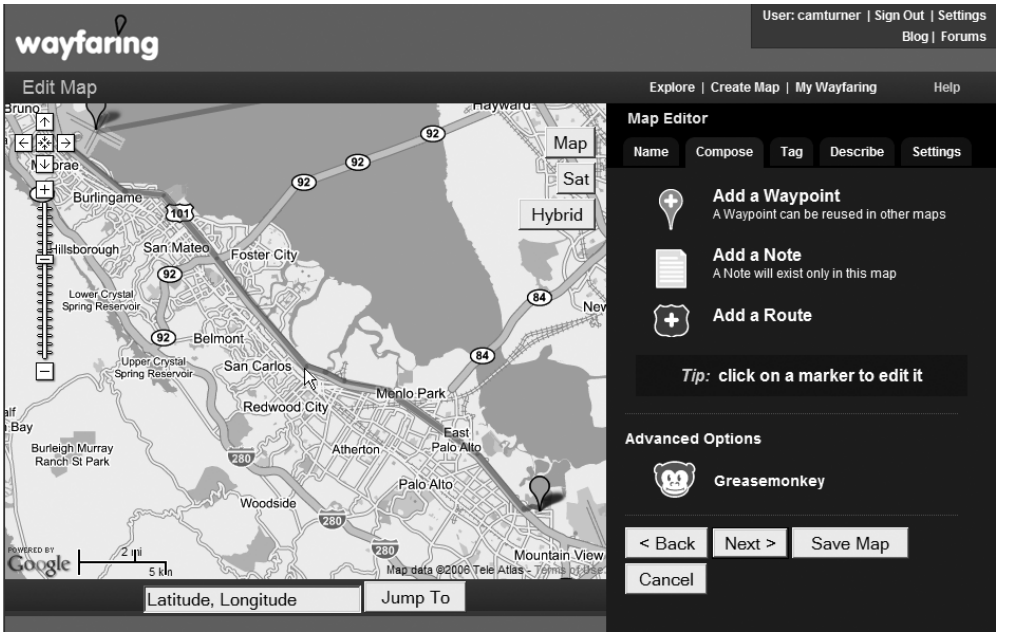


Figure 1-6. *The drive down the Bayshore Freeway to the Googleplex*

That's it. You can use the same principles to make an annotated map of your vacation or calculate how far you're going to travel; and best of all, it's a snap to share it. To see this map live, visit `http://www.wayfaring.com/maps/show/17131`.

Got Rails?

Of course, since this *is* a programming book, you're probably eager to dig into the code and make something really unique. Wayfaring may be nice, but it doesn't give you the flexibility of a programmatic approach. Looking forward to more programmatic interaction with the API, let's discuss Ruby on Rails, the server-side framework of choice.

As we mentioned at the beginning of this chapter, this book presumes you have some experience with Ruby and Rails already—at least enough to get a basic Rails application up and running. In particular, you should

- Have Ruby (1.8.4 or later) and Rails (1.1.5 or later) installed on your development machine.
- Know how to use the `rails` command to create an application skeleton and be comfortable with the model/view/controller layout of a Rails application.
- Know how to start a WEBrick, Mongrel, or other server to view your pages in a browser.
- Have a local database server. We use MySQL throughout this book, but you should be able to adapt to the DB engine of your choice.

- Have a development environment that you're comfortable with—for example, TextMate, RadRails, or Vim.

The following are things we *don't* expect you to know (or have) coming into this book:

- You don't need to know the details of the `prototype.js` JavaScript library. A lot of developers are confused by Prototype, probably due to the dearth of documentation. We've made a conscious decision in this book to not rely heavily on `prototype.js`, although we will be utilizing it from time to time.
- You don't need to know RJS. RJS is a Rails template type (like RHTML) and is a convenient way to build rich functionality on web pages with a minimum of JavaScript coding. Since the Google Maps API is implemented in JavaScript, we'll rely primarily on hand-coded JavaScript functions for this book.
- You don't need to be an expert in JavaScript programming. Yes, the Google Maps API is implemented in JavaScript, but you'll be ramping up on JavaScript techniques and principles as you go.
- You don't necessarily need to have a production web space for your Rails application. Of course, you'll want a production server so you can expose your killer app to the world, but you can learn everything in this book using your local development server.

JUST GETTING STARTED WITH RUBY AND RAILS?

If you're just getting started with Ruby and Rails and decided mapping applications are a fun way to learn, you'll probably want some additional resources to help you get up to speed. Ruby on Rails has a steeper learning curve than PHP or ColdFusion, so you will benefit from these resources to get you started out right:

- *Beginning Ruby: From Novice to Professional* by Peter Cooper (`http://www.apress.com/book/bookDisplay.html?bID=10244`). We recognize that many developers' first exposure to Ruby is through the Rails web framework. If this is the case for you, I highly recommend learning more about the wonderful Ruby language before diving headfirst into Rails.
- *Beginning Ruby on Rails: From Novice to Professional* by Cloves Carneiro Jr. and Jeffrey Allan Hardy (`http://www.apress.com/book/bookDisplay.html?bID=10124`).

What's Next?

We hope you're eager to learn how to build your own maps-based applications from the ground up using Rails. By the end of Part 1 of this book, you'll have the skills to do everything you've just done on Wayfaring (except the polylines and distances, which are covered in Chapter 10) using JavaScript and XHTML. By the book's conclusion, you'll have learned most of the concepts needed to build your own Wayfaring clone.

So what exactly is to come? This book is divided into three parts and two appendixes. Part 1 goes through Chapter 4 and deals with the basics that a hobbyist would need to get started.

You'll make a map, add some custom pins, and geocode a set of data using freely available services. Part 2 (Chapters 5 through 8) gets into more map development topics, such as building a usable interface, dealing with extremely large groups of points, and finding sources of raw information you may need to make your professional map ideas a reality. Part 3 (Chapters 9 through 11) dives into advanced topics: building custom map overlays, such as your own info window and tool tip; creating your own map tiles and projections; using the spherical equations necessary to calculate surface areas on the earth; and building your own geocoder from scratch. Finally, one appendix provides a reference guide to the Google Maps version 2 API, and another points to a few places where you can find neat data for extending the examples here and to inspire your own projects.

CHAPTER 2

Getting Started

In this chapter, you'll learn how to create your first Google map project, plot some markers, and add a bit of interactivity. Because JavaScript plays such a central role in controlling the maps, you'll also start to pick up a few essentials about that language along the way.

In this chapter, we will step through the following:

- Setting up your Rails application
- Getting off the ground with a basic map and a Google Maps API key
- Separating the map application's JavaScript functions and XHTML
- Unloading finished maps to help browsers free their memory
- Creating map markers and responding to clicks on them with an information pop-up

On JavaScript, Helpers, and Plug-ins

Rails' baked-in integration with Prototype and Scriptaculous via helper libraries is a godsend for many common JavaScript and Ajax use cases. Since the helpers cover so many common cases, many Rails developers never dig deep into JavaScript itself. If that's the case for you, get ready to learn some JavaScript. The Google Maps API is 100% JavaScript, and you need to get familiar with JavaScript to fully leverage all the API's mapping goodness.

Likewise, there are some Google Maps-related plug-ins listed on RubyForge, which will provide Google Maps-specific helpers. The two Google Maps-related plug-ins listed on RubyForge are Cartographer (`http://cartographer.rubyforge.org/`) and YM4R (`http://rubyforge.org/projects/ym4r/`). Cartographer has been around longer (according to RubyForge), but YM4R is more mature and full-featured. YM4R also seems to be more up-to-date and actively maintained.

YM4R provides an attractive way to begin working with Google Maps on Rails with minimal upfront JavaScript programming. If you choose to use it, you'll find excellent documentation at `http://thepochisuperstarmegashow.com/ProjectsDoc/ym4r_gm-doc/`.

The approach taken in this book is to interact with the Google Maps with JavaScript, the API's native tongue. Doing so has several advantages:

- You get access to the full range of capabilities of the API. You are not relying on any intermediary between your code and Google Maps.
- There are fewer layers to debug when something goes wrong with your code.
- It's easier to evolve your application to take advantage of updates to the Google Maps API. Google has demonstrated that the Maps API will undergo frequent updates. If you're working against the raw API, you are not dependent upon a third-party to upgrade a plug-in in order to take advantage of new capabilities.

Ultimately, we believe that you will be well served by learning the Google Maps API in its native JavaScript. If you later decide to utilize a plug-in like YM4R, you will possess a fundamental understanding of the JavaScript code that the plug-in produces. You will also be able to extend the plug-in or intermingle its output with raw JavaScript if necessary.

Creating Your Rails Application

You need a Rails application to hold your Google Maps code. Let's create an application now from the command line by changing into the directory in which your Rails application will live, and executing the `rails` command, passing it the name of our application. We'll call our application *maps*, so type `rails maps` from the command line. Now create a `chap_two` controller with the command `ruby script/generate controller chap_two`. You'll use this controller to get started. Note that in this chapter, you are only working with HTML and JavaScript; you don't technically need any Ruby code yet. In fact, you could run all the examples in this chapter as plain HTML files under Apache or IIS (Internet Information Services), if you are so inclined. But this is a Rails book; so why not use the Rails environment from the very start?

Before going on, make sure to note the port that your application will be running on. For example, if you're developing with RadRails, the WEBrick server generator assigns a port to each newly created server. If you already have applications running on ports 3000 and 3001, your new application will be on port 3002. You'll need to know the port to get your Google Maps key, which is the next step. From now on, we will use `http://localhost:3000` as our development URL; if yours is different, substitute your actual URL as you go.

The First Map

In this section, you'll obtain a Google Maps API key and then begin experimenting with it by retrieving Google's starter map code.

Keying Up

Before you start a Google Maps web application, you need to sign up for a Google Maps API key. To obtain your key, you must accept the Google Maps API Terms of Use. At the time of this writing, a few of the important terms included not stealing Google's imagery, obscuring the Google logo, or holding Google responsible for its software. Additionally, you're prevented from creating maps that invade privacy or facilitate illegal activities. Terms of use can change, so make sure you check them over before you plan your Google Maps–based application.

Google will issue as many keys as you need, but separate domains (or different ports on the same domain) *must* apply for a separate key. Your first key will be used in your local development environment; so enter **http://localhost:3000** as your domain. Visit `http://www.google.com/apis/maps/signup.html` (Figure 2-1) and submit the form to get your key. Throughout this book, nearly all of the examples will require you to include this key in the JavaScript `<script>` element for the Google Maps API, as you're about to see in Listing 2-1.

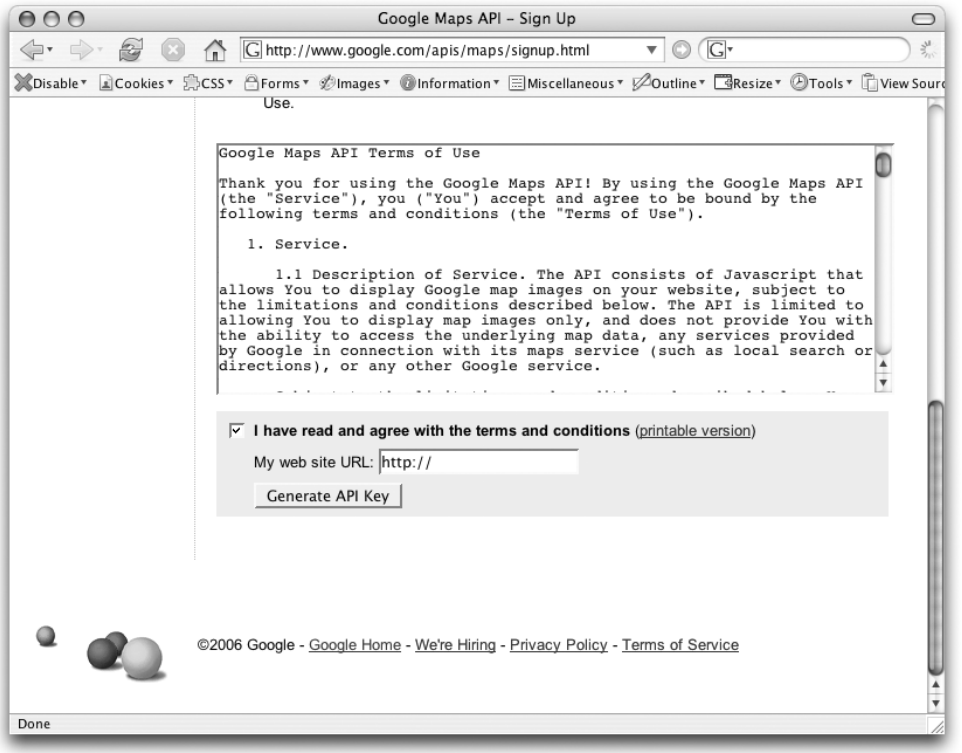


Figure 2-1. *To sign up for an API key, check the box and enter the URL of your webspace.*

■ **Note** Why a key? It's common practice for API providers to require a key, so they can monitor usage, identify the most popular projects, and note potential terms of service violations. Google is not the only one that makes you authenticate to use an API. Del.icio.us, Amazon, and others all provide services with APIs that require you to first obtain a key.

When you sign up to receive your key, Google will provide you with some very basic starter code to help you get a map up and running as quickly as possible. We'll begin by dissecting and working with this starter code so you can gain a basic understanding of what's happening.

If you start off using Google's sample, your key is already embedded in the JavaScript. Alternatively, you can—as with all listings—grab the source code from this book's web site at `http://googlemapsbook.com` and insert your own key by hand.

Either way, save the code to a file called `map.rhtml` in the directory `views/chap_two/`. Your key is that long string of characters following `key=` (Our key is `ABQIAAAA33EjxkLYsh9SEveh_MphphQP1y➥R2bHJW2Brl_bW_lOKXsyt8cxTKO5Zz-UKoJ6IepTlZRxN8nfTRgw`). Listing 2-1 shows the contents of your `views/chap_two/map.rhtml` file.

Listing 2-1. *The Google Maps API Starter Code in views/chap_two/map.rhtml*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <meta http-equiv="content-type" content="text/html; charset=utf-8"/>
        <title>Google Maps JavaScript API Example</title>
        <script src="http://maps.google.com/maps?file=api&v=2&key=ABQIAAAA➥
33EjxkLYsh9SEveh_MphphQP1yR2bHJW2Brl_bW_lOKXsyt8cxTKO5Zz-UKoJ6Ie➥
pTlZRxN8nfTRgw" type="text/javascript"></script>
        <script type="text/javascript">

        //<![CDATA[

        function load() {
            if (GBrowserIsCompatible()) {
                var map = new GMap2(document.getElementById("map"));
                map.setCenter(new GLatLng(37.4419, -122.1419), 13);
            }
        }

        //]]>
        </script>
    </head>

    <body onload="load()" onunload="GUnload()">
        <div id="map" style="width: 500px; height: 300px"></div>
    </body>
</html>
```

DOCUMENT.GETELEMENTBYID — IT'S BACK!

If you have been doing any JavaScript development with Rails, the use of `document.getElementByid` in Listing 2-1 may have caught your eye. After all, if you're using Prototype, why not use the more powerful and concise `$()` function?

The answer is that you want to make your code library-agnostic, rather than tying it closely to Prototype. We recognize the utility of other JavaScript libraries, such as jQuery (`http://jquery.com`) and MochiKit (`http://mochikit.com`), and realize that some Rails developers will opt to use alternative libraries.

We won't entirely eschew Prototype, however. In fact, some of the work in Chapter 3 is implemented in both Prototype and "raw' JavaScript, so you can compare both implementations side by side.

Since there is a complete XHTML document in your view, you don't need to wrap your view in a layout. To keep Rails from applying the layout, just remove the `application.rhtml` file from the `views/layouts` directory.

Examining the Sample Map

Make sure that your development Rails server for your maps application is started. You should now be able to hit your development URL at `http://localhost:3000/chap_two/map` and see your first Google map. You should see a map centered on Palo Alto, California.

In Listing 2-1, the container holding the map is a standard XHTML web page. A lot of the listing here is just boilerplate—standard initialization instructions for the browser. However, there are three important elements to consider.

First, the head of the document contains a critical `script` element. Its `src` attribute points to the location of the API on Google's server, and your key is passed as a parameter:

```
<script src="http://maps.google.com/maps?file=api&v=2&key=YOUR_KEY_HERE"➥
type="text/javascript"></script>
```

Second, the body section of the document contains a `div` called `map`:

```
<div id="map" style="width: 500px; height: 300px"></div>
```

Although it appears empty, this is the element in which the map will sit. Currently, a `style` attribute gives it a fixed size; however, it could just as easily be set to a dynamic size, such as `width: 50%`.

Finally, back in the head, there's a `script` element containing a short JavaScript, which is triggered by the document body's `onload` event. It's this code that communicates with Google's API and actually sets up the map:

```
function load() {
    if (GBrowserIsCompatible()) {
        var map = new GMap2(document.getElementById("map"));
        map.setCenter(new GLatLng(37.4419, -122.1419), 13);
    }
}
```

The first line is an `if` statement, which checks that the user's browser is supported by Google Maps. Following that is a statement that creates a `GMap2` object, which is one of several important objects provided by the API. The `GMap2` object is told to hook on to the map `div`, and then it gets assigned to a variable called `map`.

■ **Note** Keen readers will note that we've already encountered another of Google's special API objects: `GLatLng`. `GLatLng` is a representation of a latitude and longitude bound together for easy reference. The code `new GLatLng(37.4419, -122.1419)` creates a new `GLatLng` object with a latitude of 37.4419 and a longitude of -122.1419. `GLatLng` is a pretty important class that you're going to see a lot more of.

After you have your `GMap2` object in a `map` variable, you can use it to call any of the `GMap2` methods. The very next line, for example, calls the `setCenter()` method to center and zoom the map on Palo Alto, California. Throughout the book, we introduce various methods of each of the API objects; but if you need a quick reference while developing your web applications, you can use Appendix B of this book or view the Google Maps API reference directly online at `http://www.google.com/apis/maps/documentation/` (note that you must include the trailing slash in this URL).

Specifying a New Location

A map centered on Palo Alto is interesting, but it's not exactly groundbreaking. As a first attempt to customize this map, you're going to specify a new location for it to center on.

For this example, we'll use the Golden Gate Bridge in San Francisco (Figure 2-2). It's a large landmark and is visible in the satellite imagery provided on Google Maps. You can choose any starting point you like, but if you search for "Golden Gate Bridge" in Google Maps, move the view slightly, and then click Link to This Page, you'll get a URL in your location bar that looks something like this:

```
http://maps.google.com/maps?f=q&ll=37.818361,-122.478032&spn=0.029969,0.05579
```

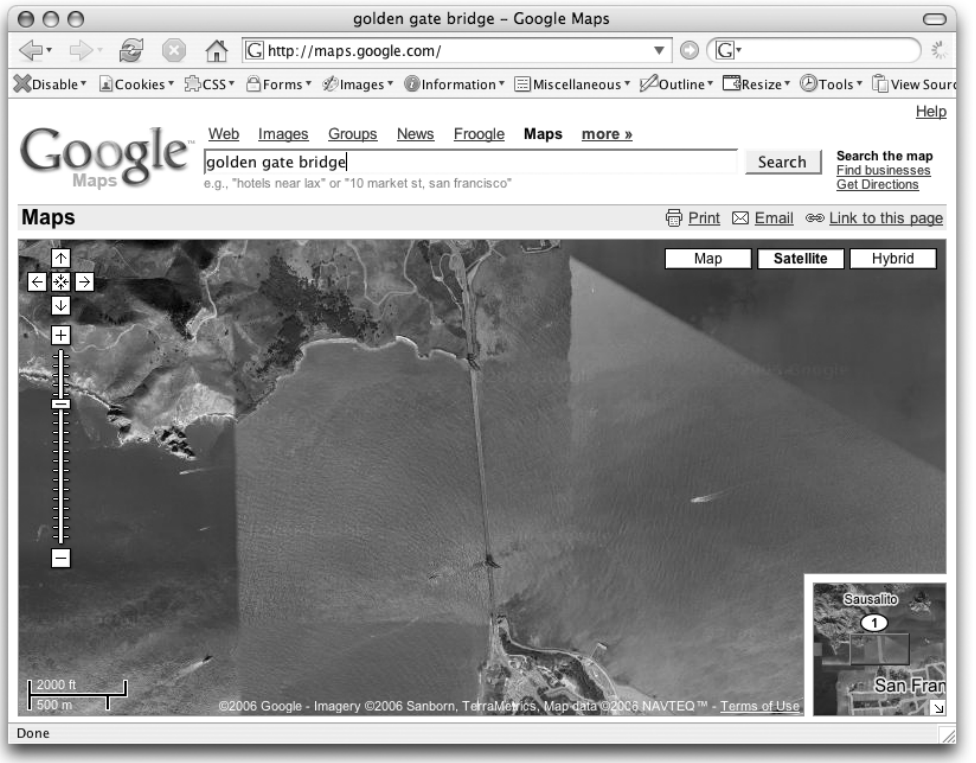


Figure 2-2. *The Golden Gate Bridge satellite imagery from Google Maps*

■ **Caution** If you use Google Maps to search for landmarks, the Link to This Page URL won't immediately contain the latitude and longitude variable but will instead have a parameter containing the search terms. To also include the latitude and longitude, you need to adjust the zoom level, or move the map so that the link is no longer to the default search position.

It's clear that the URL contains three parameters separated by ampersands:

```
f = q
ll = 37.818361, -122.478032
spn = 0.029969, 0.05579
```

The `ll` parameter is the important one you'll use to center your map. Its value contains the latitude and longitude of the center of the map in question. For the Golden Gate Bridge, the coordinates are 37.82N and 122.48W.

■ **Note** *Latitude* is the number of degrees north or south of the equator and ranges from –90 (South Pole) to 90 (North Pole). *Longitude* is the number of degrees east or west of the prime meridian at Greenwich, in England, and ranges from –180 (westward) to 180 (eastward). There are several different ways you can record latitude and longitude information. Google uses decimal notation, where a positive or negative number indicates the compass direction. The process of turning a street address into a latitude and longitude is called *geocoding*, and is covered in more detail in Chapter 4.

You can now take the latitude and longitude values from the URL and use them to recenter your own map to the new location. Fortunately, it's a simple matter of plugging the values directly into the `GLatLng` constructor.

Separating Code from Content

To further improve the cleanliness and readability of your code, you may want to consider separating the JavaScript into a different file. Just as Cascading Style Sheets (CSS) should not be mixed in with HTML, it's best practice to also keep JavaScript separated.

The advantages of this approach become clear as your project increases in size. With large and complicated Google Maps web applications, you could end up with hundreds of lines of JavaScript mixed in with your XHTML. Separating these not only increases loading speeds, as the browser can cache the JavaScript independently of the XHTML, but removal also helps prevent the messy and unreadable code that results from mixing XHTML with other programming languages. As an added benefit, moving the JavaScript to a separate file also conforms better to Rails convention, which places the JavaScript code in `public/javascripts/application.js`.

So, we will keep the XHTML in `map.rhtml`, and move the behavioral JavaScript code to `public/javascripts/application.js`. We'll also change the raw `<script>` tag to the more concise rails equivalent, `<%=javascript_include_tag%>`. At this point, your Rails application should have the following files:

```
controllers/chap_two_controller.rb
views/chap_two/map.rhtml
public/javascripts/application.js
```

Listing 2-2 shows the revised version of the `map.rhtml` file.

Listing 2-2. *Extrapolated map.rhtml File*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"➥
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <script src="http://maps.google.com/maps?file=api&amp;v=2&amp;key=➥
ABQIAAAAfAb2RNhzPafOW1mtifapBRI9caN7296ZHDcvjSpGbL7PxwkwBS➥
ZidcfOwy4q2EZpjEJx3rc4Lt5Kg" type="text/javascript"></script>
    <%=javascript_include_tag 'application'%>
</head>
```

```
<body>
     <div id="map" style="width: 500px; height: 300px"></div>
</body>
</html>
```

Listing 2-2 is the same basic HTML document as before, except that now there is an extra `script` element inside the head. Also note that we have taken the `onload` and `onunload` calls (shown in Listing 2-1) out of the `body` tag in the XHTML. We've attached the `init` function to the `onload` event in the `application.js` file; we'll deal with `onunload` later. Listing 2-3 shows the revised `public/javascripts/application.js` file.

Listing 2-3. *Extrapolated application.js File*

```
var centerLatitude = 37.818361;
var centerLongitude = -122.478032;
var startZoom = 13;

var map;

function init()
{
    if (GBrowserIsCompatible()) {
        map = new GMap2(document.getElementById("map"));
        var location = new GLatLng(centerLatitude, centerLongitude);
        map.setCenter(location, startZoom);
    }
}

window.onload = init;
```

Although the behavior is almost identical, the JavaScript code in Listing 2-3 has two important changes:

- The starting center point for latitude, longitude, and start zoom level of the map are stored in `var` variables at the top of the script, so it will be more straightforward to change the initial center point the next time. You won't need to hunt down a `setCenter()` call that's buried somewhere within the code.
- The initialization JavaScript has been moved out of the body of the XHTML and into the `application.js` file. Rather than embedding the JavaScript in the body of the XHTML, you can attach a function to the `window.onload` event. Once the page has loaded, this function will be called and the map will be initialized.

For the rest of the examples in this chapter, the `map.rhtml` file will remain exactly as it is in Listing 2-2, and you will need to add code only to the `application.js` file to introduce new features to your map.

Caution It's important to see the difference between `init` and `init()`. When you add the parentheses after the function name, it means "execute it." Without the parentheses, it means "give me a reference to it." When you assign a function to an event handler such as `document.onload`, you want to be very careful that you don't include the parentheses. Otherwise, all you've assigned to the handler is the function's return value, probably a `null`.

Cleaning Up

One more important thing to do with your map is to be sure to correctly unload it. The extremely dynamic nature of JavaScript's variables means that correctly reclaiming memory (called *garbage collection*) can be a tricky process. As a result, some browsers do it better than others.

Firefox and Safari both seem to struggle with this, but the worst culprit is Internet Explorer. Even up to version 6, simply *closing* a web page is not enough to free all the memory associated with its JavaScript objects. An extended period of surfing JavaScript-heavy sites such as Google Maps could slowly consume all system memory until Internet Explorer is manually closed and restarted.

Fortunately, JavaScript objects can be manually destroyed by setting them equal to `null`. The Google Maps API now has a special function that will destroy most of the API's objects, which helps keep browsers happy. The function is `GUnload()`, and to take advantage of it is a simple matter of hooking it on to the `body.onunload` event, as in Listing 2-4.

Listing 2-4. *Calling GUnload() in application.js*

```
var centerLatitude = 37.818361;
var centerLongitude = -122.478032;
var startZoom = 13;

var map;

function init() {
    if (GBrowserIsCompatible()) {
        map = new GMap2(document.getElementById("map"));
        var location = new GLatLng(centerLatitude, centerLongitude);
        map.setCenter(location, startZoom);
    }
}

window.onload = init;
window.onunload = GUnload;
```

There's no obvious reward for doing this, but it's an excellent practice to follow. As your projects become more and more complex, they will eat up available memory at an increasing rate. On the day that browsers are perfect, this approach will become a hack of yesterday. But for now, it's a quiet way to improve the experience for all your visitors.

Now that you have finished refactoring the code, try viewing the page again (at `http://localhost:3000/chap_two/map`) to ensure that the map still displays as expected.

Basic User Interaction

Centering the map is all well and good, but what else can you do to make this map more exciting? You can add some user interaction.

Using Map Control Widgets

The Google Maps API provides five standard controls that you can easily add to any map:

- `GLargeMapControl`: The large pan and zoom control, which is used on `http://maps.google.com/`
- `GSmallMapControl`: The mini pan and zoom control, which is appropriate for smaller maps
- `GScaleControl`: The control that shows the metric and imperial scale of the map's current center
- `GSmallZoomControl`: The two-button zoom control used in driving-direction pop-ups
- `GMapTypeControl`: The button that lets the visitor toggle between Map, Satellite, and Hybrid types

---

■ **Tip** If you're interested in making your own custom controls, you can do so by extending the `GControl` class and implementing its various functions. We don't want to divert your attention from the basics of map creation at this point. However, we may discuss custom controls on the `http://googlemapsbook.com` blog, so be sure to check there for more information.

In all cases, it's a matter of instantiating the control object and then adding it to the map with the `GMap2` object's `addControl()` method. For example, here's how to add the small map control (part of Listing 2-5):

```
map.addControl(new GSmallMapControl());
```

You use an identical process to add the other controls: simply pass in a new instance of the control's class.

■ **Note** Since you already have some familiarity with Ruby on Rails, we're assuming you're comfortable with object-oriented terminology such as *instantiating*. You may or may not know (depending on how much you've dug into native JavaScript) that JavaScript can support object-oriented programming techniques. The JavaScript code already presented (starting with Listing 2-1) illustrates this by instantiating a `GMap2` object. Note the difference in syntax between Ruby and JavaScript: in Ruby, you instantiate with `o=MyClass.new`; in JavaScript, `o=new MyClass()`. Same principle, different syntax. If you've gone under the covers of Rail's JavaScript helpers and used the Prototype library "in the raw," you are already familiar with object-oriented JavaScript techniques.

Creating Markers

The Google Maps API makes an important distinction between creating a marker, or *pin*, and adding the marker to a map. In fact, the `map` object has a general `addOverlay()` method, used for both markers and the white information bubbles.

In order to plot a marker (Figure 2-3), you need the following series of objects:

- A `GLatLng` object stores the latitude and longitude of the location of the marker.
- An optional `GIcon` object stores the image that visually represents the marker on the map.
- A `GMarker` object is the marker itself.
- A `GMap2` object has the marker plotted on it, using the `addOverlay()` method.

Figure 2-3. *Marker plotted in the middle of the Golden Gate Bridge map*

Listing 2-5 shows an updated `application.js` that incorporates these additions. The new lines are marked in bold.

Listing 2-5. *Plotting a Marker*

```
var centerLatitude = 37.818361;
var centerLongitude = -122.478032;
var startZoom = 13;

var map;
```

```
function init()
{
    if (GBrowserIsCompatible()) {
        map = new GMap2(document.getElementById("map"));
        map.addControl(new GSmallMapControl());
        var location = new GLatLng(centerLatitude, centerLongitude);
        map.setCenter(location, startZoom);

        var marker = new GMarker(location);
        map.addOverlay(marker);
    }
}

window.onload = init;
window.onunload = GUnload;
```

Caution If you try to add overlays to a map before setting the center, it will cause the API to give unpredictable results. Be careful to `setCenter()` your `GMap2` object before adding any overlays to it, even if it's just to a hard-coded dummy location that you intend to change again right away.

See what happened? We assigned the new `GLatLng` object to a variable, and then we were able to use it twice: first to center the map, and then a second time to create the marker.

The exciting part isn't creating one marker; it's creating many markers. But before we come to that, you must quickly look at the Google Maps facility for showing information bubbles.

WHITHER THOU, GICON?

You can see that we didn't actually use a `GIcon` object anywhere in Listing 2-5. If we had one defined, it would be possible to make the marker take on a different appearance, like so:

```
var marker = new GMarker(my_GLatLng, my_GIcon);
```

However, when the icon isn't specified, the API assumes the red inverted teardrop as a default. There is a more detailed discussion of how to use the `GIcon` object in Chapter 3.

Detecting Marker Clicks

It's time to make your map respond to the user. For instance, clicking a marker could reveal additional information about its location (Figure 2-4). The API provides an excellent method for achieving this result: the *info window*. The info window is the cartoonlike bubble that often appears when you click map markers; we go into info windows in more detail in Chapter 3. To know when to open the info window, you'll need to listen for a `click` event on the marker you plotted.

Figure 2-4. *An info window open over the Golden Gate Bridge*

JavaScript is primarily an event-driven language. The `init()` function that you've been using since Listing 2-3 is hooked on to the `window.onload` event. Although the browser provides many events such as these, the API gives you a convenient way of hooking up code to various events related to user interaction with the map. The code you use to listen to events is called an event listener.

For example, if you have a `GMarker` object on the map called `marker`, you could detect marker clicks like so:

```
function handleMarkerClick() {
    alert("You clicked the marker!");
}

GEvent.addListener(marker, 'click', handleMarkerClick);
```

It's workable, but it will be a major problem once you have a lot of markers. Fortunately, the dynamic nature of JavaScript yields a terrific shortcut here. You can actually just pass the function itself directly to `addListener()` as a parameter:

```
GEvent.addListener(marker, 'click',
    function() {
        alert("You clicked the marker!");
    }
);
```

Opening the Info Window

The method we'll demonstrate here is `openInfoWindowHtml()`. Although you can open info windows over arbitrary locations on the map, here you'll open them above markers only, so the code can take advantage of a shortcut method built into the `GMarker` object:

```
marker.openInfoWindowHtml(description);
```

Of course, the whole point is to open the info window only when the marker is clicked, so you'll need to combine this code with the `addListener()` function:

```
GEvent.addListener(marker, 'click',
    function() {
        marker.openInfoWindowHtml(description);
    }
);
```

Finally, you'll wrap up all the code for generating a marker, an event, and an info window into a single function called `addMarker()`, shown in Listing 2-6.

Listing 2-6. *Creating a Marker with an Info Window*

```
var centerLatitude = 37.818361;
var centerLongitude = -122.478032;
var description = 'Golden Gate Bridge';

var startZoom = 13;
var map;

function addMarker(latitude, longitude, description) {
    var marker = new GMarker(new GLatLng(latitude, longitude));

    GEvent.addListener(marker, 'click',
        function() {
            marker.openInfoWindowHtml(description);
        }
    );

    map.addOverlay(marker);
}
```

```
function init() {
    if (GBrowserIsCompatible()) {
        map = new GMap2(document.getElementById("map"));
        map.addControl(new GSmallMapControl());
        map.setCenter(new GLatLng(centerLatitude, centerLongitude), startZoom);

        addMarker(centerLatitude, centerLongitude, description);
    }
}

window.onload = init;
window.onunload = GUnload;
```

This is a nice, clean function that does everything you need for plotting a pin with a clickable information bubble. Now you're perfectly set up for plotting a whole bunch of markers on your map. Before moving on, be sure to fire up your browser and make sure the marker displays the pop-up bubble when clicked.

A List of Points

In Listing 2-3, we introduced the variables `centerLongitude` and `centerLatitude`. Global variables such as these are fine for a single centering point, but what you *probably* want to do is store a whole series of values and map a bunch of markers all at once. Specifically, you want a list of latitude and longitude pairs representing the points of the markers you'll plot.

Using Arrays and Objects

To store the list of points, you can combine the power of JavaScript's array and object constructs. An *array* stores a list of numbered entities. An *object* stores a list of keyed entities. The two constructs are analogous to Ruby's array and hash classes. Compare these two JavaScript lines to the Ruby equivalents:

```
// JavaScript Array and Object
var myArray = ['John', 'Sue', 'James', 'Edward'];
var myObject = {'John': 19, 'Sue': 21, 'James': 24, 'Edward': 18};
```

```
# Ruby Array and Hash
my_array=['John', 'Sue', 'James', 'Edward']
my_hash={'John' => 19, 'Sue' => 21, 'James' => 24, 'Edward' => 18}
```

As you can see, both the concept and syntax are very similar in both languages. And in both languages, the means of accessing elements in the data structures is similar as well. If you have used arrays and hashes in Ruby, then you will have no problem using arrays and objects in JavaScript.

To access elements of the array in JavaScript, you must use their numeric indices. So, `myArray[0]` is equal to `'John'`, and `myArray[3]` is equal to `'Edward'`.

The JavaScript object is like the Ruby hash. In the object, the names themselves are the indices, and the numbers are the values. To look up how old Sue is, all you do is check the value of `myObject['Sue']`.

■ **Note** For accessing members of an object, JavaScript allows both `myObject['Sue']` and the alternative notation `myObject.Sue`. The second is usually more convenient, but the first is important if the value of the index you want to access is stored in *another* variable, for example, `myObject[someName]`.

For each marker you plot, you want an object that looks like this:

```
var myMarker = {
        'latitude': 37.818361,
        'longitude': -122.478032,
        'name': 'Golden Gate Bridge'
};
```

Having the data organized this way is useful because the related information is grouped as "children" of a common parent object. The variables are no longer just `latitude` and `longitude`—now they are `myMarker.latitude` and `myMarker.longitude`.

Most likely, for your application you'll want more than one marker on the map. To proceed from one to many, it's just a matter of having an array of these objects:

```
var myMarkers = [Marker1, Marker2, Marker3, Marker4];
```

Then you can cycle through the array, accessing the members of each object and plotting a marker for each entity.

When the nesting is combined into one step (Figure 2-5), it becomes a surprisingly elegant data structure, as in Listing 2-7.

Listing 2-7. *A JavaScript Data Structure for a List of Locations*

```
var markers = [
    {
        'latitude': 37.818361,
        'longitude': -122.478032,
        'name': 'Golden Gate Bridge'
    },
    {
        'latitude': 40.6897,
        'longitude': -74.0446,
        'name': 'Statue of Liberty'
    },
    {
        'latitude': 38.889166,
        'longitude': -77.035307,
        'name': 'Washington Monument'
    }
];
```

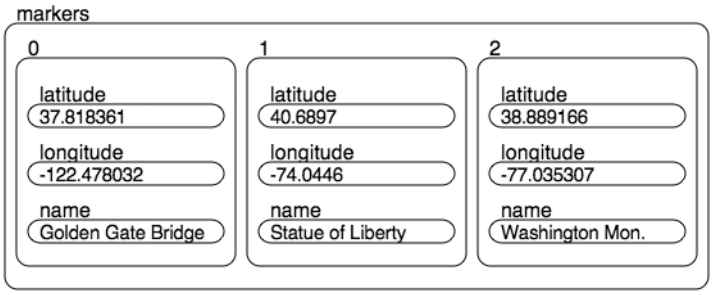


Figure 2-5. *A series of objects stored inside an array*

Iterating

JavaScript, like many languages, provides a `for` loop—a way of repeating a block of code *for* so many iterations, using a counter. One way of cycling through your list of points is a loop such as this:

```
for (id = 0; id < markers.length; id++) {
        // create a marker at markers[id].latitude, markers[id].longitude
}
```

You may have come across another way of iterating through an array called a `for in` loop. The `for in` loop looks like this:

```
for (id in markers) {
        // create a marker at markers[id].latitude, markers[id].longitude
}
```

It looks cleaner, and it is. Unfortunately, Rails' own Prototype library completely breaks the `for in` construct by adding additional methods to the array class. These methods show up alongside your data in `for in` iterations, yielding unexpected results and often causing confusing, hard-to-find bugs. If you expect to use Prototype in your project, don't use `for in`. Likewise, if you are incorporating JavaScript code that was not designed to run with Prototype, take a close look at the code for any `for in` loops. Any `for in` loops you find may be the root of subtle, hard-to-find problems when the code is used together with Prototype.

■ **Note** For more discussion on Prototype's approach and its consequences, see James Mc Parlane's blog post "Why I Don't Use the prototype.js JavaScript Library" at `http://blog.metawrap.com/blog/WhyIDontUseThePrototypejsJavaScriptLibrary.aspx`.

Now it's time to incorporate the data structure from Listing 2-7 into your code. For now, you'll simply put this structure in your `map.rhtml` file between `<script>` tags. Listing 2-8 shows `map.rhtml` with this addition in bold.

Listing 2-8. *Extrapolated map.rhtml File*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"➥
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <script src="http://maps.google.com/maps?file=api&amp;v=2&amp;key=➥
ABQIAAAAfAb2RNhzPafOW1mtifapBRI9caN7296ZHDcvjSpGbL7PxwkwBS➥
ZidcfOwy4q2EZpjEJx3rc4Lt5Kg" type="text/javascript"></script>
    <%=javascript_include_tag 'application'%>
    <script type="text/javascript">
      var markers = [
       {
         'latitude': 37.818361,
         'longitude': -122.478032,
         'name': 'Golden Gate Bridge'
       },
       {
         'latitude': 40.6897,
         'longitude': -74.0446,
         'name': 'Statue of Liberty'
       },
       {
         'latitude': 38.889166,
         'longitude': -77.035307,
         'name': 'Washington Monument'
       }
      ];
    </script>
</head>
<body>
     <div id="map" style="width: 500px; height: 300px"></div>
</body>
</html>
```

To get that structure plotted, it's just a matter of wrapping the marker-creation code in a `for` loop, as shown in Listing 2-9. Note that we've also changed the map center and zoom in this code (relative to the previous listings in this chapter), so all three markers are visible on the map.

Listing 2-9. *application.js Modified to Use the Markers from map_data.rhtml*

```
var map;
var centerLatitude = 40.6897;
var centerLongitude = -95.0446;
var startZoom = 3;
```

```
function addMarker(latitude, longitude, description) {
    var marker = new GMarker(new GLatLng(latitude, longitude));

    GEvent.addListener(marker, 'click',
        function() {
           marker.openInfoWindowHtml(description);
        }
    );

    map.addOverlay(marker);
}

function init() {
    if (GBrowserIsCompatible()) {
        map = new GMap2(document.getElementById("map"));
        map.addControl(new GSmallMapControl());
        map.setCenter(new GLatLng(centerLatitude, centerLongitude), startZoom);

        for(i=0; i<markers.size; i++) {
           addMarker(markers[i].latitude, markers[i].longitude, markers[i].name);
        }
    }
}

window.onload = init;
window.onunload = GUnload;
```

Nothing here should be much of a surprise. You can see that the `addMarker()` function is called for each of the markers, so you have three markers and three different info windows.

Summary

With this chapter behind you, you've made an incredible amount of progress. You've looked at several good programming practices, seen how to plot multiple markers, and popped up the info window. And all of this is in a tidy, reusable package.

So what will you do with it? Plot your favorite restaurants? Mark where you parked the car? Show the locations of your business? Maybe mark your band's upcoming gigs?

The possibilities are endless, but it's really just the beginning. In the next chapter, you'll expand on what you learned here by creating your map data dynamically and learning the key to building a real community: accepting user-submitted information. After that, the weird and wonderful science of geocoding—turning street addresses into latitudes and longitudes—will follow, along with a variety of tips and tricks you can use to add flavor to your web applications.

CHAPTER 3

Interacting with the User and the Server

Now that you've created your first map (in Chapter 2) and had a chance to perform some initial experiments using the Google Maps API, it's time to make your map a little more useful and dynamic.

In this chapter, you'll explore a few examples of how to provide interactivity in your map using the Google Maps API, and you'll see how you can use the API to save and retrieve information from your server. While building a small web application, you'll learn how to do the following:

- Trigger events on your map and markers to add either new markers or info windows
- Modify the content of info windows attached to a map or to individual markers
- Use Google's `GXmlHttp` object to communicate with your server
- Improve your web application by changing the appearance of the markers

Adding Interactivity

Most good Google Maps applications rely on interaction with the user in order to customize the information displayed on the map. As you've already learned, it's relatively easy to create a map and display a fixed set of points using static HTML and a bit of JavaScript. Anyone with a few minutes of spare time and some programming knowledge could create a simple map that would, for example, display the markers of all the places he visited on his vacation last year. A static map such as this is nice to look at, but once you've seen it, what would make you return to the page to look at it again? To keep people coming back and to hold their attention for longer than a few seconds, you need a map with added interactivity and a bit of flair.

You can add interactivity to your map applications in a number of ways. For instance, you might offer some additional detail for each marker using the info window bubbles or something more elaborate such as filtering the markers based on search criteria. Google Maps is a good reference point for simple and effective map interaction. It provides the required interactivity by allowing you to search for specific businesses and lists other relevant businesses nearby but then goes even further to offer driving directions to the marked locations. Allowing you to see the location of a business you're looking for is great, but telling you how to get there in your car, now that's

interactivity. Without the directions, the map would be an image with a bunch of pretty dots, and you would be left trying to figure out how to get to each dot. Regardless of how it's done, the point is that interacting with the map is always important, but don't go overboard and overwhelm your users with too many options.

Going on a Treasure Hunt

To help you learn about some of the interactive features of the Google Maps API, you're going to go on a treasure hunt and create a map of all the treasures you find. The treasures in this case are *geocaches*, little plastic boxes of goodies that are hidden all over the earth.

For those of you who are not familiar with geocaches (not to be confused with *geocoding*, which we discuss in the next chapter), or *geocaching* as the activity is commonly referred to, it is a global hide-and-seek game that can be played by anyone with a Global Positioning System (GPS) device (Figure 3-1) and some treasure to hide and seek. People worldwide place small caches of trinkets in plastic containers and then distribute the GPS locations using the Internet. Other people then follow the latitude and longitude coordinates and attempt to locate the hidden treasures within the cache. Upon finding a cache, they exchange the item in the cache for something of their own.

Figure 3-1. *A common handheld GPS device used by geocachers to locate hidden geocaches*

Note For more information about geocaching, check out the official Geocaching web site (`http://www.geocaching.com`) or pick up *Geocaching: Hike and Seek with Your GPS* by Erik Sherman (`http://www.apress.com/book/bookDisplay.html?bID=194`).

As you create your interactive geocache treasure map, you'll learn how to do the following:

- Create a map and add a JavaScript event trigger using the `GEvent.addListener()` method to react to clicks by the users, so that people who visit the map can mark their finds on the map.
- Ask users for additional information about their finds using an info window and an embedded HTML form.
- Create a MySQL table to store information about the points on the map.
- Save the latitude, longitude, and additional information in an HTML form to your server using the `GXmlHttp` Asynchronous JavaScript and XML (Ajax) object on the client side and a Rails action on the server. In the "Ajax with Prototype" section, you'll see code to perform similar functionality using the `prototype.js` JavaScript library.
- Retrieve the existing markers and their additional information from the server using Ajax.
- Recreate the map upon loading by inserting new markers from a server-side list, with each marker having an info window to display its information.

In this chapter, we're not going to discuss any CSS styling of the map and its contents; we'll leave the CSS styling up to you for now.

Reviewing Application Structure

In Chapter 2, you did the following:

- Created the Rails application named *maps*
- Created the `chap_two` controller
- Created the `map.rhtml` file to hold the markup, including the map container `div`
- Added JavaScript to `application.js`

In Chapter 3, you'll reuse the application and create a new controller, view, and JavaScript file. You'll also create a migration to set up your database schema, and a model to interact with the database. Finally, you'll create new actions to list existing markers and to create new ones. You'll interact with both these actions through Ajax.

Building on Your Application

In Chapter 2, we called our controller `chap_two`. In this chapter, we'll call our controller `chap_three`, and we'll continue this convention (naming the controller after the chapter) throughout this book.

Creating a New Controller

Run the `ruby script/generate controller chap_three` command, which will generate the `chap_three_controller.rb` file in the `controllers` directory. You should now have both `chap_two_controller.rb` and `chap_three_controller.rb` files in your `controllers` directory.

Creating a Marker Model and Migration

Let's create a model representing marker objects. Run the `ruby script/generate model Marker` command, which will generate the `models/marker.rb` file. This should be the first file you have in your `models` directory. The `ruby script/generate model` command also generates a migration for the new model; look for the new migration in `db/migrations/001_create_marker.rb`. Place the following code in the new migration file:

```
class CreateMarkers < ActiveRecord::Migration
  def self.up
    create_table :markers do |t|
      t.column :lat, :decimal
      t.column :lng, :decimal
      t.column :found, :string, :limit=>100
      t.column :left, :string, :limit=>100
    end
    execute("ALTER TABLE markers MODIFY lat numeric(15,10);")
    execute("ALTER TABLE markers MODIFY lng numeric(15,10);")
  end

  def self.down
    drop_table :markers
  end
end
```

This is a standard Rails migration, with the exception of the two `ALTER TABLE` lines. These are here to set the precision and scale of the numeric `lat` and `lng` columns, which you'll use to store the latitude and longitude of the markers. The `ALTER TABLE` commands are necessary because at the time of this writing, the Rails migrations to specify the precision and scale of numeric columns were not working correctly (all numeric columns were getting the default of scale=10, precision=0). We expect this to be fixed in Rails 1.2; in the meantime, this workaround is an acceptable way to get a numeric column type with the desired scale and precision.

Creating the Database, Connecting via Rails, and Running the Migration

As with any Rails application, you need to have a database for your application to connect to. Since you're already working with Rails, you probably already know how to set up a database. Regardless, we will provide a brief overview here in case your memory needs jogging. In keeping with standard Rails naming conventions, we will call our database `maps_development`. This example assumes you are using MySQL, but you should be able to adapt the examples if you are using a different database.

First, create the database itself. From the command line, type `mysql -uroot` to get the `mysql>` prompt, then type `create database maps_development;`. You can also use your favorite MySQL administrative GUI; we prefer MySQL Query Browser (`http://www.mysql.com/products/tools/query-browser`).

Next, back in your Rails environment, open up your `config/database.yml` file and configure it with the right database name. The development section of `database.yml` should look like this:

```
development:
  adapter: mysql
  database: maps_development
  username: root
  password:
  host: localhost
```

Finally, you need to run your migration to create the `markers` table. At your command line, run `rake db:migrate`. You should see the following:

```
== CreateMarkers: migrating ================================
-- create_table(:markers)
   -> 0.4010s
-- execute("ALTER TABLE markers MODIFY lat numeric(15,10);")
   -> 0.1800s
-- execute("ALTER TABLE markers MODIFY lng numeric(15,10);")
   -> 0.2000s
== CreateMarkers: migrated (0.7810s) =======================
```

If the migration fails, double-check your `database.yml` file for database name, user, and password. We assume that your local development environment is set up with a blank password for your MySQL root login; if this is not the case in your environment, adjust the `database.yml` settings accordingly.

You might want to try creating a `Marker` object from the command line to ensure that everything is set up properly. To do so, start up an interactive Rails console with the `ruby script/console` command. The following console session demonstrates creating and then deleting a `Marker` model interactively with ActiveRecord. The bold lines are your input:

```
Loading development environment.
>> Marker.find :all
=> []
>> Marker.create({:lat => 37.0, :lng=>-122.0})
=> #<Marker:0x3924170 @errors=#<ActiveRecord::Errors:0x391ffe8 @errors={},➥
@base=#<Marker:0x3924170 ...>>, @attributes={"left"=>nil, "found"=>nil, "id"=>1,➥
"lng"=>-122.0, "lat"=>37.0}, @new_record=false>
>> m=Marker. find :first
=> #<Marker:0x3911408 @attributes={"left"=>nil, "found"=>nil,➥
"lng"=>"-122.0000000000", "id"=>"1", "lat"=>"37.0000000000"}>
>> m.destroy
=> #<Marker:0x3911408 @attributes={"left"=>nil, "found"=>nil,➥
"lng"=>"-122.0000000000", "id"=>"1", "lat"=>"37.0000000000"}>
>> Marker.find :all
=> []
>> exit
```

Creating the Map View

Since you are using a new controller for this chapter, you need a new view. Fortunately you can reuse the view code from Chapter 2; just copy `views/chap_two/map.rhtml` to `views/chap_three/map.rhtml`. Your `views/chap_three/map.rhtml` file will be referencing your existing `public/javascripts/application.js` file. That's fine, or you can archive your existing `application.js` file as `application_chap_two.js` (or something similar) and copy its contents to a fresh `public/javascripts/application.js` for this chapter.

At this point, you should be able to hit `http://localhost:3000/chap_three/map` and see your map, just as you left it in Chapter 2. Now, let's get back to the JavaScript and Google Maps code by creating the map and marking points.

Creating the Map and Marking Points

You may have noticed in Chapter 2 that you declared the `map` variable outside the `init()` function in Listing 2-2. Declaring `map` outside the `init()` function allows you to reference `map` at any time and from any auxiliary functions you add to the `application.js` file. It will also ensure you're targeting the same `map` object. Also, you may want to add some of the control objects mentioned in Chapter 2, such as `GMapTypeControl`. Listing 3-1 highlights the `map` variable and additional controls. You should replace the contents of your existing `application.js` file with the code in this listing.

Listing 3-1. *The New public/javascripts/application.js File*

```
var centerLatitude = 37.4419;
var centerLongitude = -122.1419;
var startZoom = 12;

var map;
```

```
function init() {
    if (GBrowserIsCompatible()) {
        map = new GMap2(document.getElementById("map"));
        map.addControl(new GSmallMapControl());
        map.addControl(new GMapTypeControl());
        map.setCenter(new GLatLng(centerLatitude, centerLongitude), startZoom);
    }
}

window.onload = init;
window.onunload = GUnload;
```

Now you have a solid starting point for your web application. When viewed in your web browser, the page will have a simple map with controls centered on Palo Alto, California, (Figure 3-2), the location represented by the `centerLatitude` and `centerLongitude` variables in Listing 3-1. For this example, the latitude and longitude is not important, so feel free to change it to some other location if you wish.

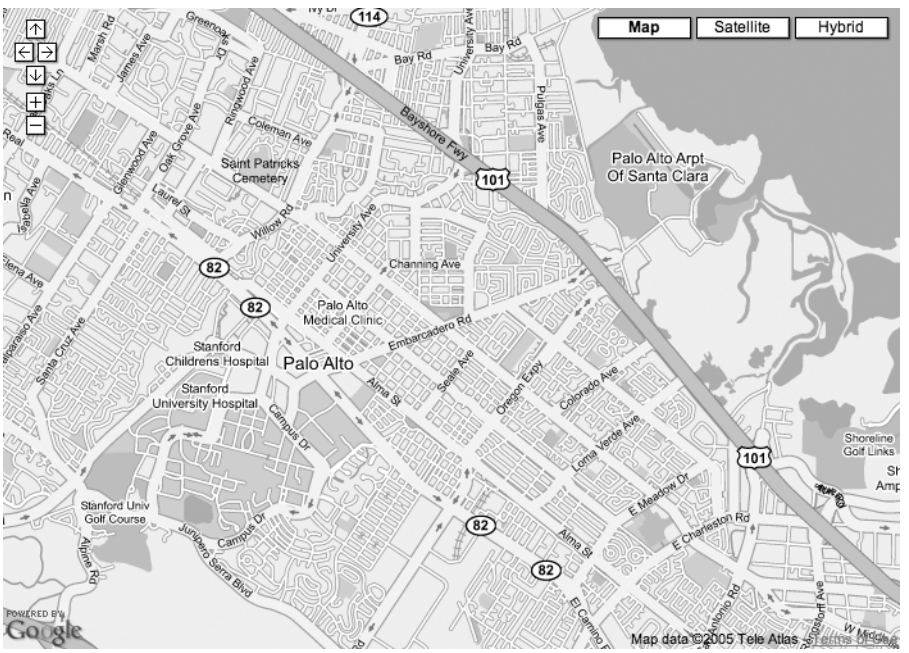


Figure 3-2. *Starting map with controls centered on Palo Alto, California*

Listening to User Events

The purpose of your map is to allow visitors to add markers wherever they click. To capture the clicks on the map, you'll need to trigger a JavaScript function to execute whenever the map area is clicked. As you saw in Chapter 2, Google's API allows you to attach these triggers, called *event listeners*, to your map objects through the use of the `GEvent.addListener()` method. You

can add event listeners for a variety of events, including `move` and `click`, but in this case, you are interested only in users clicking on the map, not moving it or dragging it around.

■ **Tip** If you refer to the Google Maps API documentation in Appendix B of this book, you'll notice a wide variety of events for both the `GMap2` and the `GMarker` objects, as well as a few others. Each of these different events can be used to add varying amounts of interactivity to your map. For example, you could use the `moveend` event for the `GMap2` to trigger an Ajax call and retrieve points for the new area of the map. For the geocaching map example, you could also use the `GMarker`'s `infowindowclose` event to check to see if the information in the form has been saved and, if not, ask the user what to do. You can also attach events to Document Object Model (DOM) elements using `GEvent.addDomListener()` and trigger an event using JavaScript with the `GEvent.trigger()` method.

The `GEvent.addListener()` method handles all the necessary code required to watch for and trigger each of the events. All you need to do is tell it which object to watch, which event to listen for, and which function to execute when it's triggered. Given the source `map` and the event `click`, this example will trigger the function to run any code you wish to implement:

```
GEvent.addListener(map, "click", function(overlay, latlng) {
     //your code
});
```

Let's add an event listener to the `init()` function in your `application.js` file. Listing 3-2 shows the complete `init()` function, with the new lines you should add in bold.

Listing 3-2. *Using the addListener() Method to Create a Marker at the Click Location*

```
function init() {
     if (GBrowserIsCompatible()) {
          map = new GMap2(document.getElementById("map"));
          map.addControl(new GSmallMapControl());
          map.addControl(new GMapTypeControl());
          map.setCenter(new GLatLng(centerLatitude, centerLongitude), startZoom);

          //allow the user to click the map to create a marker
          GEvent.addListener(map, "click", function(overlay, latlng) {
               var marker = new GMarker(latlng)
               map.addOverlay(marker);
          });
     }
}
```

Compare this to the marker creation you did in Chapter 2. In Chapter 2, you used `new GLatLng()` to create the latitude and longitude location for the markers based on data embedded in the page. Here, instead of creating a new `GLatLng`, you use the `latlng` variable passed

into the event listener's handler function. The `latlng` variable is a `GLatLng` representation of the latitude and longitude where you clicked on the map. The other argument to the handler function—`overlay`—represents the marker or overlay you clicked, if applicable. The handler function in this example ignores the `overlay` variable altogether.

With this listener, anyone visiting your map page could add as many markers as they want (Figure 3-3). However, all the markers will disappear as soon as the user leaves the page, never to be seen again. To keep the markers around, you need to collect some information and send it back to the server for storage using the `GXmlHttp` object or the `GDownloadUrl` object discussed in the "Using Google's Ajax Object" section later in this chapter.

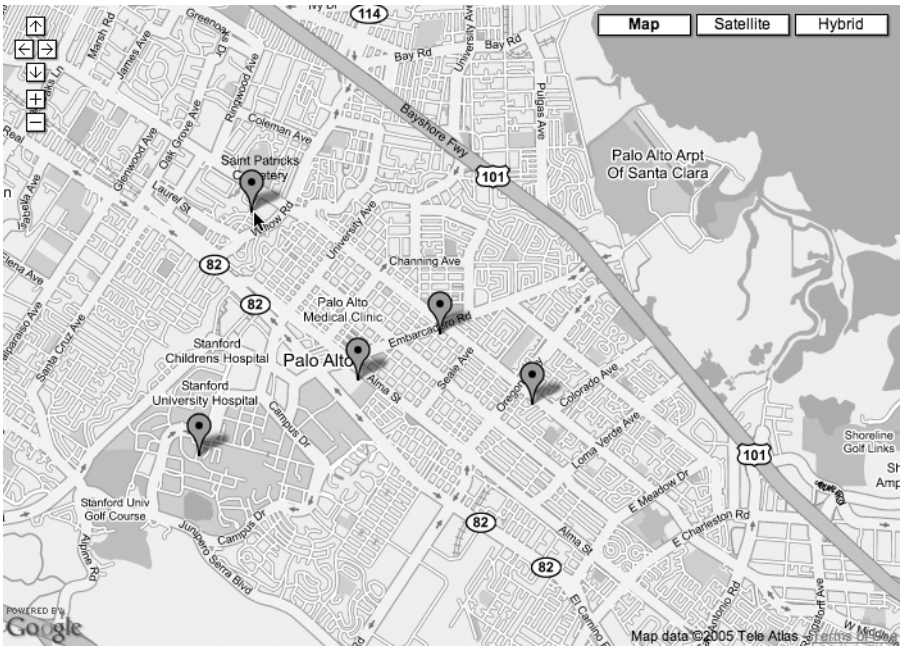


Figure 3-3. *New markers created by clicking on the map*

RETRIEVING THE LATITUDE AND LONGITUDE FROM A MAP CLICK

When you click on a Google map, the `latlng` variable passed into the event listener's handler function is a `GLatLng` object with `lat()` and `lng()` methods. Using the `lat()` and `lng()` methods makes it relatively easy for you to retrieve the latitude and longitude of any point on earth simply by zooming in and clicking on the map. This is particularly useful when you are trying to find the latitude and longitude of places that do not have readily accessible latitude/longitude information for addresses.

In countries where there is excellent latitude and longitude information, such as the United States, Canada, and more recently, France, Italy, Spain, and Germany, you can often use an address lookup service to retrieve the latitude and longitude of a street address. But in other locations, such as the United Kingdom, the data is limited or inaccurate. In the case where data can't be readily retrieved by computer, manual human

entry of points may be required. For more information about geocoding and using addresses to find latitude and longitude, see Chapter 4.

Additionally, if you want to retrieve the X and Y coordinates of a position on the map in pixels on the screen, you can use the `fromLatLngToDivPixel()` method of the `GMap2` object. By passing in a `GLatLng` object, `GMap2.fromLatLngToDivPixel(latlng)` will return a `GPoint` representation of the X and Y offset relative to the DOM element containing the map.

Asking for More Information with an Info Window

You could simply collect the latitude and longitude of each marker on your map, but just the location of the markers would provide only limited information to the people browsing your map. Remember, interactivity is key, so you want to provide a little more than just a marker. For the geocaching map, visitors really want to know what is found at each location. To provide this extra information, let's create a little HTML form. When asking for input of any type in a web browser, you need to use HTML form elements. In this case, let's put the form in an info window indicating where the visitor clicked.

As introduced in Chapter 2, the info window is the cartoonlike bubble that often appears when you click map markers (Figure 3-4). It is used by Google Maps to allow you to enter information in the To Here and From Here fields to get driving directions, or to show you a zoomed view of the map at each point in the directions. Info windows do not need to be linked to markers on the map. They can also be created on the map itself to indicate locations where no marker is present.

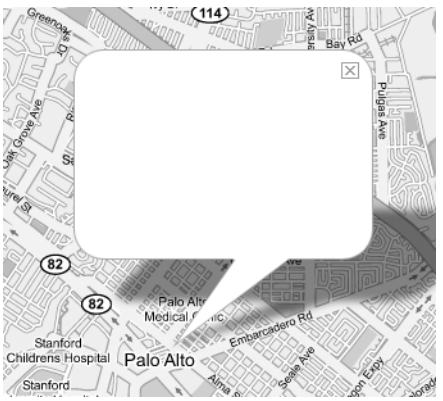


Figure 3-4. *An empty info window*

You're going to use the info window for two purposes:

- It will display the information about each existing marker when the marker is clicked.
- It will hold a little HTML form so that your geocachers can tell you what they've found.

Note We explain how to save the content of the info window to your server in the "Implementing Ajax" section later in this chapter.

Creating an Info Window on the Map

In Listing 3-2, you used the event listener to create a marker on your map where it was clicked. Rather than creating markers when you click the map, you'll modify your existing code to create an info window. To create an info window directly on the map object, call the `openInfoWindow()` method of the map:

```
GMap2.openInfoWindow(GLatLng, htmlDomElem, GInfoWindowOptions);
```

`openInfoWindow()` takes a `GLatLng` as the first parameter and an HTML DOM document element as the second parameter. The last parameter, `GInfoWindowOptions`, is optional unless you want to modify the default settings of the window.

For a quick demonstration, modify your `application.js` file's `init()` function to use the following event listener, which opens an info window when the map is clicked, rather than creating a new marker:

```
GEvent.addListener(map, "click", function(overlay, latlng) {
    map.openInfoWindow (latlng,document.createTextNode("You clicked here!"));
});
```

Now when you click the map, you'll see an info window pop up with its base pointing at the position you just clicked with the content "You clicked here!" (Figure 3-5).

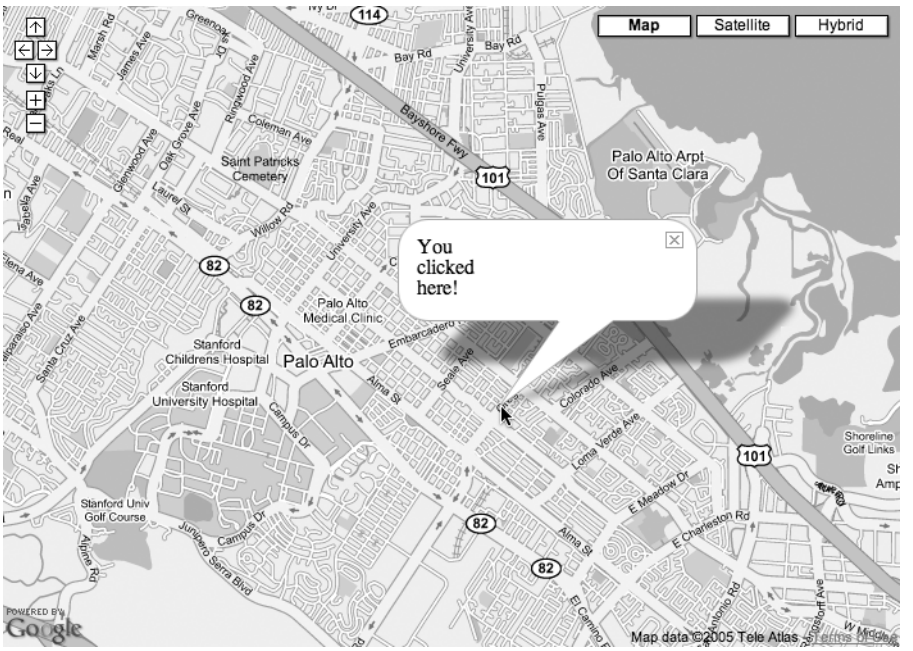


Figure 3-5. *An info window created when clicking the map*

Embedding a Form into the Info Window

When geocachers want to create a new marker, you'll first prompt them to enter some information about their treasure. You'll want to know the geocache's location (this will be determined using the point where they clicked the map), what they found at the location, and what they left behind. To accomplish this in your form, you'll need the following:

- A text field for entering information about what they found
- A text field for entering information about what they left behind
- A hidden field for the longitude
- A hidden field for the latitude
- A Submit button

The HTML form used for the example is shown in Listing 3-3, but as you can see in Listing 3-4, you are going to use the JavaScript DOM object and methods to create the `form` element. You need to use DOM because the `GMarker.openInfoWindow()` method expects an HTML DOM element as the second parameter, not simply a string of HTML.

■ **Tip** If you want to make the form a little more presentable, you could easily add IDs and/or classes to the `form` elements and use CSS styles to format them accordingly.

Listing 3-3. *HTML Version of the Form for the Info Window*

```
<form action="" onsubmit="createMarker(); return false;">
     <fieldset style="width:150px;">
          <legend>New Marker</legend>
          <label for="found">Found</label>
          <input type="text" id="found" name="m[found]" style="width:100%;"/>
          <label for="left">Left</label>
          <input type="text" id="left" name="m[left]" style="width:100%;"/>
          <input type="submit" value="Save"/>
          <input type="hidden" id="longitude" name="m[lng]" />
          <input type="hidden" id="latitude" name="m[lat]" />
     </fieldset>
</form>
```

■ **Note** You may notice the form in Listing 3-3 has an `onsubmit` event attribute that calls a `createMarker()` JavaScript function. The `createMarker()` function does not yet exist in your script, and if you try to click the Save button, you'll get a JavaScript error. Ignore this for now, as you'll create the `createMarker()` function in the "Saving Data with GXmlHttp" section later in the chapter, when you save the form contents to the server.

Listing 3-4. *Adding the DOM HTML Form to the Info Window*

```
GEvent.addListener(map, "click", function(overlay, latlng) {

     //create an HTML DOM form element
     var inputForm = document.createElement("form");
     inputForm.setAttribute("action","");
     inputForm.onsubmit = function() {createMarker(); return false;};

     //retrieve the longitude and lattitude of the click point
     var lng = latlng.lng();
     var lat = latlng.lat();

      inputForm.innerHTML = '<fieldset style="width:150px;">'
          + '<legend>New Marker</legend>'
          + '<label for="found">Found</label>'
          + '<input type="text" id="found" name="m[found]" style="width:100%;"/>'
          + '<label for="left">Left</label>'
          + '<input type="text" id="left" name="m[left]" style="width:100%;"/>'
          + '<input type="submit" value="Save"/>'
          + '<input type="hidden" id="longitude" name="m[lng]" value="'+ lng +'"/>'
          + '<input type="hidden" id="latitude" name="m[lat]" value="' + lat + '"/>'
          + '</fieldset>';
     map.openInfoWindow (latlng,inputForm);
});
```

Caution When creating the DOM `form` element, you need to use the `setAttribute()` method to define attributes such as `name`, `action`, `target`, and `method`, but once you venture beyond these basic four, you may begin to notice inconsistencies. For example, using `setAttribute()` to define `onsubmit` works fine in Mozilla-based browsers but not in Microsoft Internet Explorer browsers. For cross-browser compatibility, you need to define `onsubmit` using a function, as you did in Listing 3-4. For more detailed information regarding DOM and how to use it, check out the DOM section of the W3Schools web site at `http://www.w3schools.com/dom/`.

After you've changed the `GEvent.addListener()` call in Listing 3-2 to the one in Listing 3-4, when you click your map, you'll see an info window containing your form, as shown in Figure 3-6.

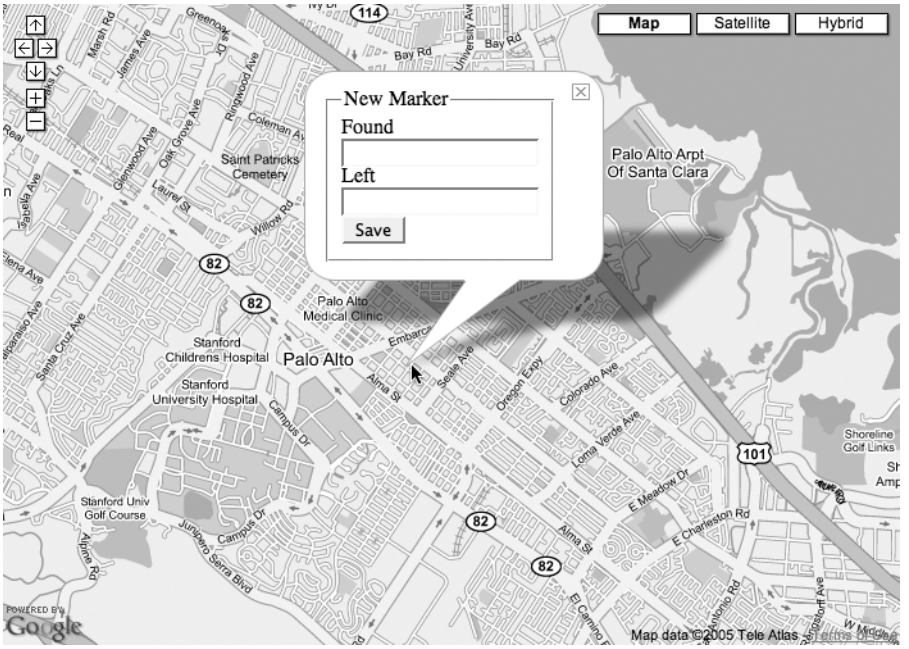


Figure 3-6. *The info window with an embedded form*

In Listing 3-4, the latitude and longitude elements of the form are prepopulated with the `latlng.lat()` and `latlng.lng()` values from the `GLatLng` object passed in to the event listener. This allows you to later save the latitude and longitude coordinates and recreate the marker in the exact position when you retrieve the data from the server. Also, once the information is saved for the new location, you can use this latitude and longitude to instantly create a marker at the new location, bypassing the need to refresh the web browser to show the newly saved point.

If you click again elsewhere on the map, you'll notice your info window disappears and reappears at the location of the new click. As a restriction of the Google Maps API, you can have only one instance of the info window open at any time. When you click elsewhere on the map, the original info window is destroyed and a brand-new one is created. Be aware that it is not simply moved from place to place.

You can demonstrate the destructive effect of creating a new info window yourself by filling in the form, as shown in Figure 3-7, and then clicking elsewhere on the map without clicking the Save button. You'll notice that the information you entered in the form disappears (Figure 3-8) because the original info window is destroyed and a new one is created.

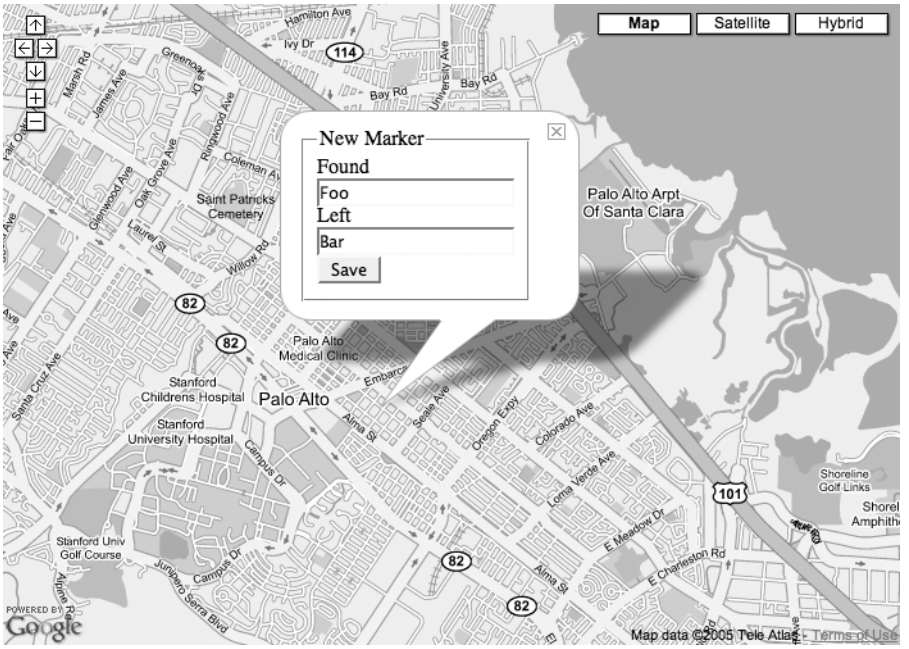


Figure 3-7. *Info window with populated form information*

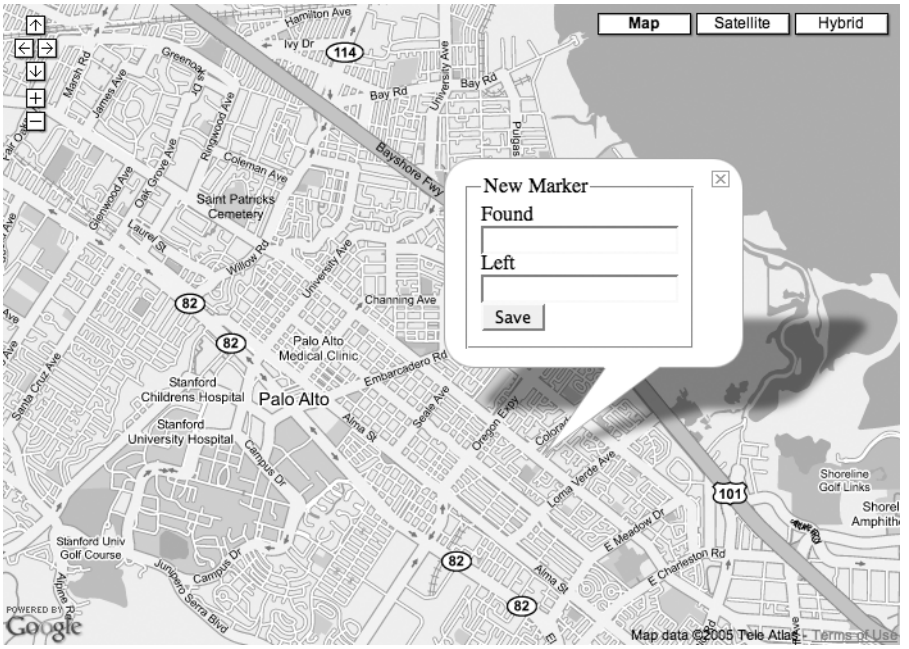


Figure 3-8. *New info window that has lost the previously supplied information*

Earlier, when you created the info window containing "You clicked here!" the same thing happened. Each marker had the same content ("You clicked here!"), so it just *appeared* as though the info window was simply moving around.

Tip If you've taken some time to review the Google Maps API in Appendix B, you might be wondering why you couldn't use the `GMap2.openInfoWindowHtml()` method to add the form to the info window. After all, it lets you use an HTML string rather than an HTML DOM element. The short answer is you can. Our example created a DOM element, set its `innerHTML` property with the HTML of our form, and used the `openInfoWindow()` to display the window. You could just as easily take the HTML itself and display it with `openInfoWindowHtml()`.

Avoiding an Ambiguous State

When creating your web applications, be sure not to create the marker until after you've verified the information and saved it to the server. If you create the marker first and the user then closes the info window using the window's close button (Figure 3-9), there would be a marker on the map that wasn't recorded on the server (Figure 3-10).

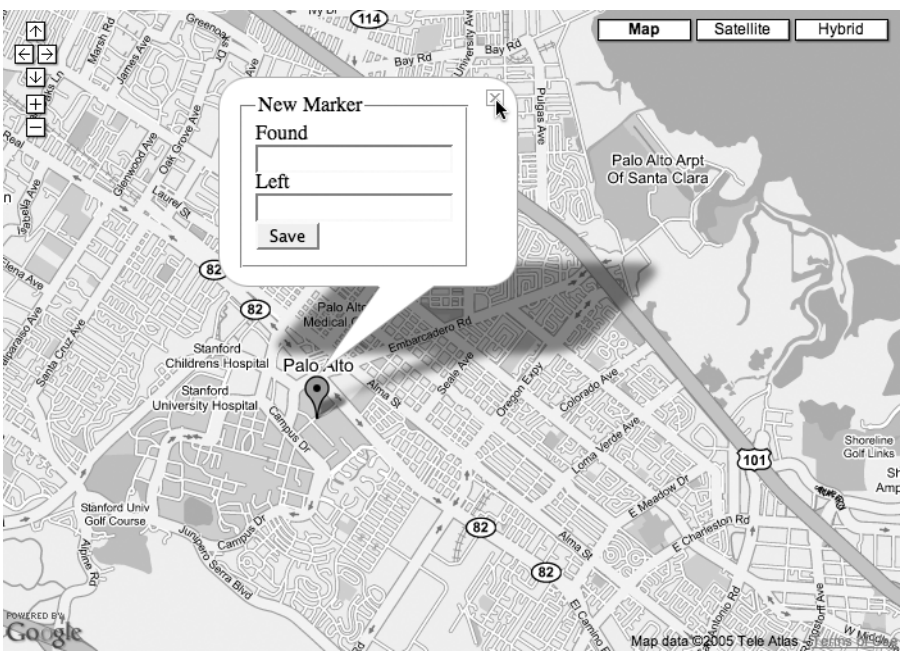


Figure 3-9. *Using the close (X) button to close the info window*

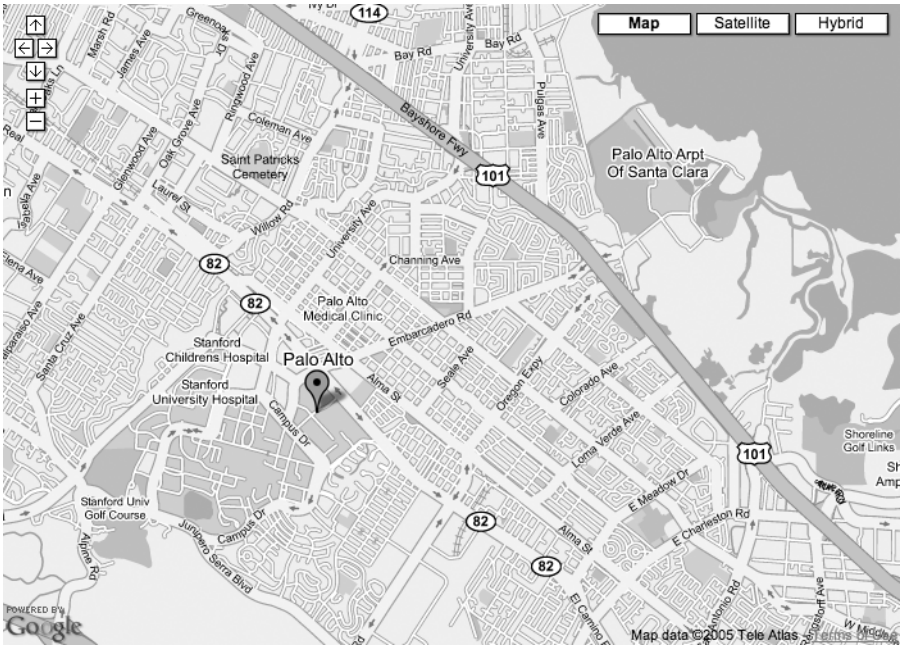


Figure 3-10. *Marker left behind by closing the window*

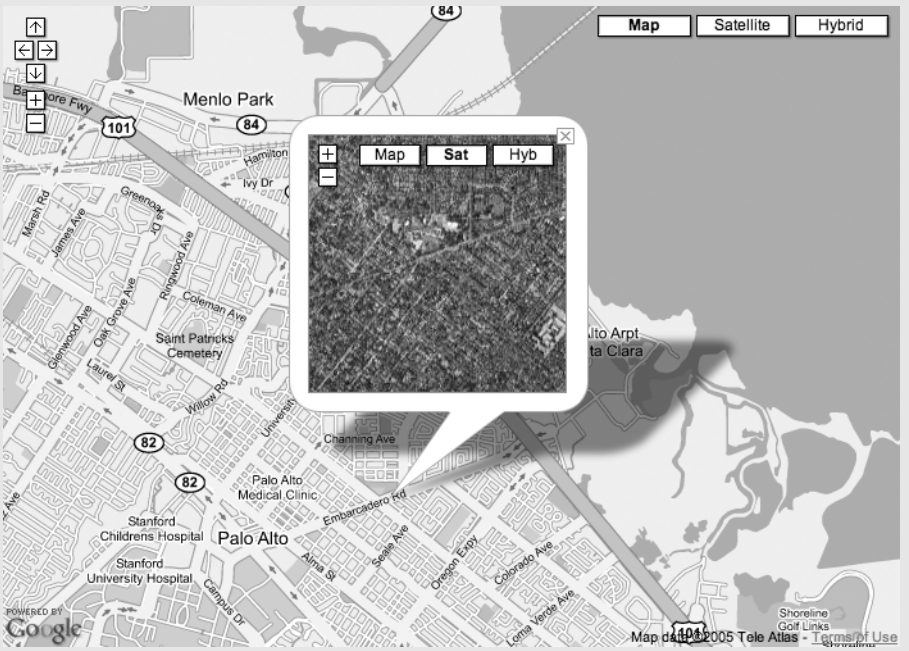
By creating the marker only after the data has been saved, you ensure the integrity of the map and keep the visible markers in sync with the stored markers on the server.

If you want, you can save the marker information in two steps: first send just the latitude and longitude to save the marker's location, and then send a second request to save the additional information, if any. Storing the latitude and longitude immediately may seem like a logical idea, until you realize that users may click the map in the wrong location and inadvertently add a bunch of points that don't really mean anything. For the geocaching map, you want to be sure there is information associated with each point, so you need to save all the information in one request.

■ **Caution** Don't confuse the `GMap2.openInfoWindow()` method with the `GMarker.openInfoWindow()` method. The map and marker objects have similar properties and methods; however, their parameters differ. You need to use the `GMap2` methods when creating info windows attached to the map itself, but if you have an existing marker, you could then use the `GMarker` methods to attach an info window to the marker. The `GMarker` methods can't be used to create an info window without a marker.

INFO WINDOWS THAT ZOOM

In addition to the `GMap2.openInfoWindow()` method with the `GMarker.openInfoWindow()` method, `GMap2.showMapBlowup()` and `GMarker.showMapBlowup()` are two other methods in the Google Maps API that will let you create info windows. Info windows created by these methods are special and contain a zoomed-in view of the map. For example, `map.showMapBlowup(new GLatLng(37.4419, -122.1419), {zoomLevel:15, mapType:G_SATELLITE_TYPE});` will display a small satellite map at zoom level 14 centered on Palo Alto, California. If you create the map blowup in an event listener, you can zoom in on any point you click on your map. The figure illustrates what the blowup info window looks like.



Controlling the Info Window Size

When you add content to the info window, it will automatically expand to encompass the content you've placed in it. The info window will expand in the same way a `<div>` tag expands to its internal content. To provide a bit of control over how it expands, you can add CSS styles to the content of the info window in the same way you would in a regular HTML page.

In Listings 3-3 and 3-4, the `<fieldset>` element was assigned a width of `150px`, forcing the info window's content container to 150 pixels wide (Figure 3-11). Also, the text `<input>` elements were set to a width of `100%` to display a simple, clean form. (For more tips and tricks regarding info windows, see Chapter 9.)

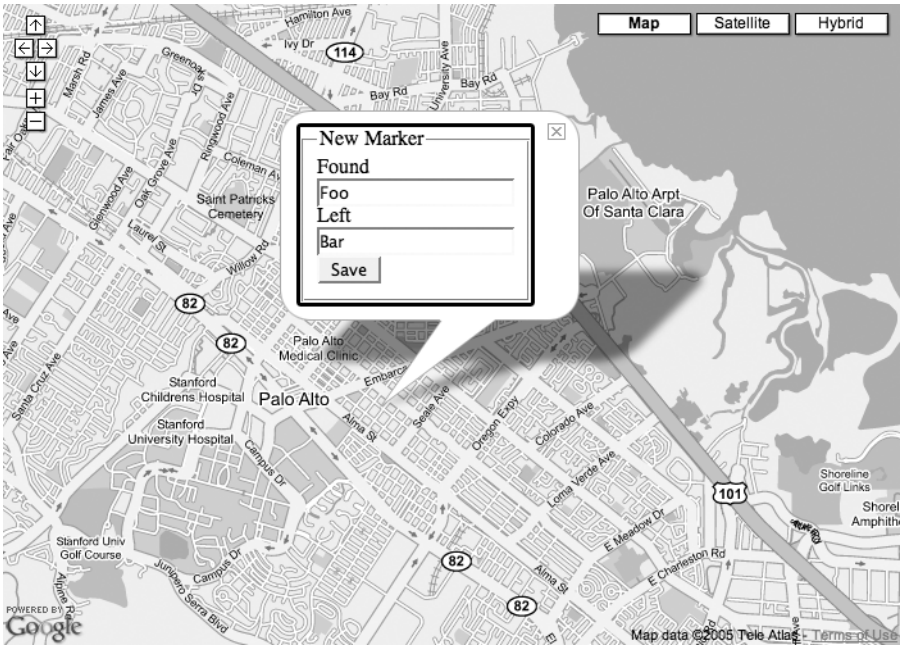


Figure 3-11. *Info window with an inner width of 150 pixels*

HOW TO CHANGE THE STYLE OF THE INFO WINDOW

Getting tired of the cartoon bubble and want to create something fancier with the info window API? Sorry, you're out of luck—well, sort of.

Currently, the Google Maps API doesn't allow you to change the style of the info window, but you could create your own using the `GOverlay` class. If you're interested, check out Chapter 9, where you'll learn how to create your own info window, as shown in the figure.

Implementing Ajax

To save the markers entered by your geocaching visitors, you're going to use Ajax to communicate with your server. Ajax relies completely on JavaScript running in your visitor's web browser; without JavaScript running, Ajax simply won't work. You can argue that using a strictly JavaScript-based Ajax interface might not be a good idea. You read everywhere that in order to be good coders and offer compliant services, you should always have an alternative solution to JavaScript-based user interfaces, and that's completely true, but the Google Maps API itself doesn't offer an alternative for JavaScript-disabled browsers. Therefore, if geocachers are visiting your page without the ability to use JavaScript, they're not going to see the map. Feel free to build alternative solutions for all your other web tools, and we strongly suggest that you do, but when dealing strictly with the Google Maps API, there isn't really much point in a non-JavaScript solution, since without JavaScript, the map itself is useless.

Google's GXmlHttp vs. Prototype's Ajax.Request

To communicate with your server, Google has provided you access to its integrated Ajax object called `GXmlHttp`. Obviously, as a Rails developer, you also have the Prototype library at your fingertips, which (among much else) provides an `Ajax.Request` object. The question is, which to use? There are pros and cons to both approaches.

Reasons to Use GXmlHttp

The primary reason to use `GXmlHttp` is that it's already included in the Google Maps JavaScript library. There's nothing more to include or download. If your project is not JavaScript-centric, you may not need the functionality that Prototype provides. In this case, you can include only the Google Maps JavaScript and utilize `GXmlHttp` as your Ajax abstraction.

Another reason to use `GXmlHttp` is if you are exploring other JavaScript libraries (such as moo.fx, jQuery, or MochiKit), and haven't yet committed to one. In this case, you can use `GXmlHttp`, and be assured that whatever supplementary JavaScript library you adopt, your Google Maps applications will continue to work without modification.

In short, you may decide to use `GXmlHttp` if either 1) you want to include the minimal JavaScript necessary to build functional maps applications, or 2) you want your maps application to remain JavaScript library-agnostic.

Reasons to Use Prototype's Ajax Functions

There are many good reasons to use the Prototype library: 1) It's included by default with your Rails environment; 2) Prototype offers richer Ajax functionality, including periodical updates and global callback functions; and 3) Using Prototype means you can also leverage the Scriptaculous effects library (which requires Prototype). However, the best reason may simply be that you already know some Prototype and prefer to leverage that experience rather than learning the Google Maps Ajax library.

You Be the Judge

We're going to give you the best of both worlds. First we'll take you through the details of building the Ajax functionality with Google's own `GXmlHttp` object. Near the end of this chapter, we will reimplement a key function with Prototype's `Ajax.Request` object. By seeing the same functionality built with both libraries, you can make an informed decision for your own

project on which approach is best for you. However, this is a Google Maps book, so we definitely spend more time with Google than with Prototype.

The next step is to look in depth at Google's Ajax object.

Using Google's Ajax Object

To implement the `GXmlHttp` object, a few things need to happen when users click the Save button:

- The information in your form needs to be sent to the server and verified for integrity.
- The information needs to be stored in the database.
- Your server-side action needs to respond back to the client-side JavaScript to let the client know that everything was successful and send back any necessary information.
- The client-side JavaScript needs to indicate to the user that there was either an error or a successful response.

To accomplish this, let's send the information back to the server and store it in our MySQL table. Then, when responding that everything is OK, let's create a new marker on the map with the new information to confirm to the user that the data is successfully saved.

■ **Caution** The Google `GXmlHttp` object, and any other Ajax script based on the `XmlHttpRequest` object, allows you to query only within the domain where the map is served. For example, if your map were at `http://example.com/webapp/`, the `GXmlHttp.request()` method can retrieve data only from scripts located in the `http://example.com` domain. You can't retrieve data from another domain such as `http://jeffreysambells.com`, as the request would break the web browser's same origin security policy (for more information see `http://www.mozilla.org/projects/security/components/same-origin.html`). Using a little JavaScript trickery to dynamically add `<script>` tags to the page does allow you to get around this policy but requires you to do special things on the server side as well. For an example of how to do this, check out the `XssHttpRequest` object at `http://jeffreysambells.com/posts/2006/03/06/centralized_ajax_services/`.

Saving Data with GXmlHttp

To send information to the server using the `GXmlHttp` object, first you need to retrieve the information from the form in the info window you created. Referring back to Listings 3-3 and 3-4, you'll notice that each of the form elements has a unique ID associated with it. Since you're using the Ajax method to send data, the form will not actually submit to the server using the traditional `POST` method. To submit the data, you retrieve the values of the form by using the JavaScript `document.getElementById()` method and concatenate each of the values onto the `GET` string of the `GXmlHttp` request object. Then, using the `onreadystatechange()` method of the `GXmlHttp` object, you can process the request when it is complete.

Listing 3-5 shows the `createMarker()` and `addMarkerToMap()` functions to add to your `application.js` file. As we indicated earlier, you'll need an action back at the server for the `createMarker()` function to call via Ajax. We'll create this action next.

Tip For more information about the `XmlHttpRequest` object and using it to send data via the `POST` method, see the W3Schools page at `http://www.w3schools.com/xml/xml_http.asp`.

Listing 3-5. *Sending Data to the Server Using GXmlHttp*

```
function createMarker(){
    var lng = document.getElementById("longitude").value;
    var lat = document.getElementById("latitude").value;

    var getVars =  "?m[found]=" + document.getElementById("found").value
        + "&m[left]=" + document.getElementById("left").value
        + "&m[lng]=" + lng
        + "&m[lat]=" + lat ;

    var request = GXmlHttp.create();

    //call the create action back on the server
    request.open('GET', 'create' + getVars, true);
    request.onreadystatechange = function() {
        if (request.readyState == 4) {
            //the request is complete

            var success=false;
            var content='Error contacting web service';
            try {
              //parse the result to JSON (simply by eval-ing it)
              res=eval( "(" + request.responseText + ")" );
              content=res.content;
              success=res.success;
            }catch (e){
              success=false;
            }

            //check to see if it was an error or success
            if(!success) {
                alert(content);
            } else {
                //create a new marker and add its info window
                var latlng = new GLatLng(parseFloat(lat),parseFloat(lng));
                var marker = addMarkerToMap(latlng, content);
                map.addOverlay(marker);
                map.closeInfoWindow();
            }
        }
    }
```

```
    request.send(null);
    return false;
}

function addMarkerToMap(latlng, html) {
   var marker = new GMarker(latlng);
   GEvent.addListener(marker, 'click', function() {
   var markerHTML = html;
   marker.openInfoWindowHtml(markerHTML);
   });
   return marker;
}
```

This `createMarker()` function is responsible for sending the marker information to the server through Ajax. It retrieves the information from the form and sends it to the `create` action on the `ChapThreeController` controller. The supporting `addMarkerToMap()` function is used to create the `GMarker` object and populate its info window. By creating the `GMarker` in another function, you can reuse the same function later when retrieving markers from the server (which you'll see in the "Retrieving Markers from the Server" section later in this chapter).

Now let's add the `create` action to `ChapThreeController`. The code in Listing 3-6 goes in the `controllers/chap_three_controller.rb` file.

Listing 3-6. *The create Rails Action Used to Save the Marker Information in the Database*

```
class ChapThreeController < ApplicationController
  def create
    marker = Marker.new(params[:m])
    if marker.save
      res={:success=>true,:content=>"<div><strong>found </strong>#{marker.found}➥
</div><div><strong>left </strong>#{marker.left}</div>"}
    else
      res={:success=>false,:content=>"Could not save the marker"}
    end
    render :text=>res.to_json
  end
end
```

As you can see from the code, you instantiate a new `Marker` object using the parameters passed into the action. In this case, you are taking the hash stored under the `:m` symbol in the parameters and using it to create the new marker. You'll recall that in the HTML form we displayed on the map, the fields were named in an `m[fieldname]` bracketed format—for example, `m[lat]`. When a form is submitted to a Rails action using the bracketed format, ActiveView converts all the bracketed strings to a hash, in this case under the key `:m`.

After the action creates the marker, it saves it. Finally, it renders a JSON structure with two keys, `:success` and `:content`, which indicate whether the action is successful, and the HTML to be displayed, respectively.

Checking When the Request Is Complete

When you click the Save button on the info window, the information in the form is sent back to the server using the `GXmlHttp` object in Listing 3-5 and awaits a response in JSON format from the Rails action in Listing 3-6. During the request, the `readyState` property of the request object will contain one of five possible incrementing values:

`0`: uninitialized

`1`: loading

`2`: loaded

`3`: interactive

`4`: completed

The changes to the `readyState` property are monitored using the `GXmlHttp.onreadystatechange()` event listener (Figure 3-12). At each increment, the function you've defined for the `onreadystatechange()` method will be triggered to allow you to execute any additional JavaScript code you would like. For the example, you need to deal with only the completed state of the request, so your function checks to see when `readyState` is equal to `4`, and then parses the XML document as necessary.

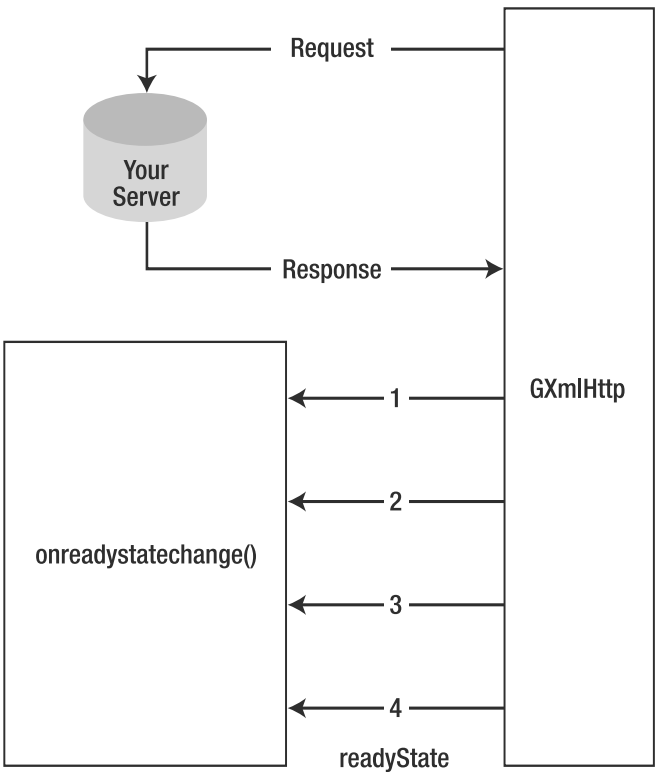


Figure 3-12. *GXmlHttp request response diagram*

Tip This Ajax implementation is not actually checking to see if the request completed in a valid state. For example, if the page you request is not found, the `readyState` will still be `4` at the end of the request, but no XML will have been returned. To check the state of the `GXmlHttp` request, you need to check the `GXmlHttp.status` property. If `status` is equal to 200, the page is successfully loaded. A status of 404 indicates the page is not found. `GXmlHttp.status` could contain any valid HTTP request codes returned by the server.

Testing the Asynchronous State

Don't forget that the `GXmlHttp` request is asynchronous, meaning that the JavaScript continues to run while your `GXmlHttp` request is awaiting a response. While the Ruby server-side action is busy saving the marker in the database, any code you've added after `request.send(null);` may execute before the response is returned from the server.

You can observe the asynchronous state of the request and response by adding a JavaScript `alert()` call right after you send the request:

```
request.send(null);
alert('Continue');
return false;
```

and another alert in the `onreadystatechange()` method of the request:

```
request.onreadystatechange = function() {
     if (request.readyState == 4) {
          alert('Process Response');
     }
}
```

If you run the script over and over, sometimes the alert boxes will appear in the order of `Process Response` then `Continue`, but more likely, you'll get `Continue` then `Process Request`. Just remember that if you want something to occur after the response, you must use the `onreadystatechange()` method when the `readyState` is `4`.

Tip To further enhance your web application, you could use the various states of the request object to add loading, completed, and error states. For example, when initiating a request, you could show an animated loading image to indicate to the user that the information is loading. Providing feedback at each stage of the request makes it easier for the user to know what's happening and how to react. If no loading state is displayed, users may assume they have not actually clicked the button or will become frustrated and think nothing is happening. By the way, triggering UI modifications on various states of the `XmlHttp` call is something that Prototype makes very easy, as you'll see in the "Ajax with Prototype" section later in this chapter.

Using GDownloadUrl for Simple Ajax Requests

If your web application doesn't require a high level of control over the Ajax request, you can use an alternative object called `GDownloadUrl`. You can use `GDownloadUrl` to send and retrieve content the same way you do with `GXmlHttp`; however, the API is much simpler. Rather than checking response states and all that other stuff, you just supply a URL with any appropriate `GET` variables and a function to execute when the response is returned. This simplifies the request to the following:

```
GDownloadUrl('create' + getVars,  function(data,responseCode)) {
    //Do something with the data
});
```

But note that this approach doesn't give you as much control over the different states of the request.

Parsing the JSON Structure

When the `readyState` reaches `4` and your `onreadystatechange()` function is triggered, you need to parse the response from the server to determine whether the action replied with an execution error or a successful save. Referring back to Listing 3-6 (the `create` action on `ChapThreeController`), you can see that in the event of a successful save, the `success` attribute of the JSON structure is `true`. The JSON result from a successful call to the `create` action looks like this:

```
{ "success":true, "content": "<div><b>Found</b> foo</div><div>➥
<b>Left</b>bar</div>"}
```

In the event of an error, such as the script not having permission to write to the file, the value of `success` is `false`. The JSON result of a failed call to the `create` action looks like this:

```
{ "success":false, "content": "Could not save the marker"}
```

When the web browser receives the JSON structure from your `request` object, it is contained in the `responseText` property. Evaluating the contents of `responseText` gives you a JavaScript object, the attributes of which you can access just like any other JavaScript object. For example, you can inspect `res.success` to see if the save is successful.

In the event of an error in Listing 3-5, you simply need to call a JavaScript `alert(res.content)` function to alert the user.

With a successful execution, you create a new marker at the latitude and longitude of the click and attach an event listener to the marker itself in order to create a new info window with the content of the response. The new marker now indicates the newly created location on the map, and when clicked, displays the information about the marker, as shown in Figure 3-13.

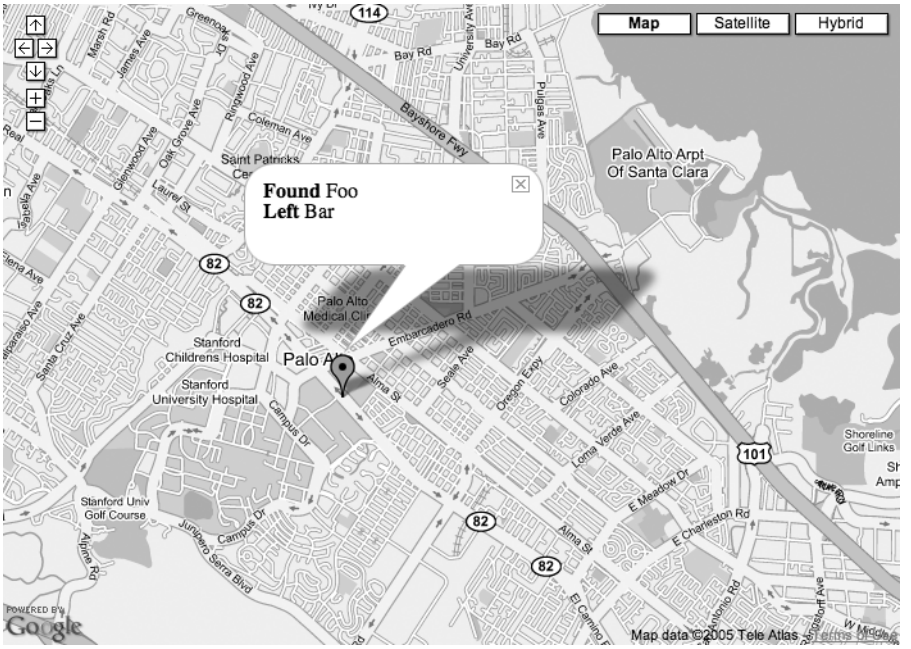


Figure 3-13. *A successful request and response*

You've probably noticed that in Listing 3-5, you used the `marker.openInfoWindowHTML()` method rather than the `map.openInfoWindow()` method. Since you now have your marker on the map, you can apply the info window directly to it and pass in an HTML string rather than an HTML DOM element.

■ **Caution** When accepting input from the users of your web site, it is always good practice to assume the data is evil and the user is trying to take advantage of your system. Always filter input to ensure it's in the format you are expecting. Numbers should be numbers, strings should be strings, and unless desired, nothing should contain HTML or JavaScript code. Listing 3-6 could easily be compromised through cross-site scripting (XSS) if you don't filter out JavaScript in the user-submitted data. For more information see `http://owasp.org`.

Retrieving Markers from the Server

Your geocaching map is almost finished. So far, you've used event listeners to add marks on the map, displayed info windows to ask for more input, and saved the input back to the server using Ajax. You now want to take all the markers you've been collecting and show them on the page when users first visit.

The information to display the markers resides on the server in the `markers` table, populated by the `create` action in Listing 3-5. In Chapter 2, you put the map data in line with the `map` view. You could easily do the same thing here, but instead, you're going to mix things up a bit and try a more controlled way. To gain more interactive control, you'll retrieve the data from the server using the `GXmlHttp` object. This will allow you to retrieve points at any time, not just when the page loads. For example, you could use this approach to track the movements of the map and retrieve the points based on the map's viewable area, as you'll see in Chapter 7.

To do this, you need a new action on `chapThreeController` to render the markers in JSON format. Since all the action does is list the markers stored in the database, we'll name the action `list`. Add the code in Listing 3-7 to `controllers/chap_three_controller.rb`.

Listing 3-7. *A New Action to be Placed in controllers/chap_three_controller.rb*

```
def list
  render :text=>(Marker.find :all).to_json
end
```

Isn't Rails great? A single line of code retrieves all the markers in the database and renders them in JSON format.

The `list` action on `ChapThreeController` is only half of the solution. You also need the JavaScript code to call the `list` action and use the JSON data it returns to create markers on the map. The `listMarkers()` function in Listing 3-8 takes care of this. Add the `listMarkers()` function to your `public/javascripts/application.js` file (you should add to, not replace, the contents of `application.js`). Notice that markers are added by the same `addMarkerToMap()` function you used in Listing 3-5. This allows you to maintain the proper scope of the data passed into the info window. If you create each marker in the `listMarkers()` function, each marker's info window will have the `html` value of the last marker created in the loop. The value of `html` will be identical for each marker because the info window is not actually created until you click the marker and `html` is retrieved from the scope of the JavaScript at that time. By moving the creation into another function, you've given each instance of the function its own namespace.

Listing 3-8. *Ajax listMarkers() Function*

```
function listMarkers() {
    var request = GXmlHttp.create();

    //tell the request where to retrieve data from.
    request.open('GET', 'list', true);

    //tell the request what to do when the state changes.
    request.onreadystatechange = function() {
      if (request.readyState == 4) {
        //parse the result to JSON,by eval-ing it.
        //The response is an array of markers
```

```
        markers=eval( "(" + request.responseText + ")" );

        for (var i = 0 ; i < markers.length ; i++) {
          var marker=markers[i].attributes
          var lat=marker.lat;
          var lng=marker.lng;

          //check for lat and lng so MSIE does not error
          //on parseFloat of a null value
          if (lat && lng) {
            var latlng = new GLatLng(parseFloat(lat),parseFloat(lng));

            var html = '<div><b>Found</b> '
              + marker.found
              + '</div><div><b>Left</b> '
              + marker.left
              + '</div>';

            var marker = addMarkerToMap(latlng, html);
            map.addOverlay(marker);
          } // end of if lat and lng
        } // end of for loop
      } //if
    } //function

    request.send(null);
}
```

Once you've created the `list` action and appended the `listMarkers()` function to the `application.js` file, you can load the markers into your map by calling the `listMarkers()` function. For example, to load the markers when the page loads, you'll need to call the `listMarkers()` function from the `init()` function after you create the map:

```
function init() {
     ... cut ...
     map = new GMap2(document.getElementById("map"));
     listMarkers();
     ... cut ...
}
```

When the `listMarkers()` function is executed, the server-side Rails `list` action (Listing 3-7), will return a JSON data structure containing the latitude and longitude for each marker you previously saved. The JSON data structure will look like this:

```
[{"attributes":
  { "lng": "-122.1676254272",
    "lat": "37.4203446339"}},
 {"attributes":
    {"lng": "-122.1834182739",
     "lat": "37.4623235089"}}
]
... etc ...
```

The JSON data structure also contains the additional information you requested for each marker so that you'll be able to include it in the info window.

You don't necessarily have to return JSON to the `GXmlHttp` object. You can also return HTML, text, or XML. JSON has emerged as a favorite format for many Rails programmers because it is easily and consistently parsed by browsers, and its fundamental structure (arrays and hashes) map well to data structures typically produced by Rails actions.

Adding Some Flair

You now have a fun little interactive web application. If you have a production environment to deploy to, anyone with Internet access can use it. Geocachers can come and see what kinds of things have been found and let others know what they've found. You could be finished with the map, but let's use the Google Maps API to add just a bit more flair.

All the red markers on the map don't really mean anything when you look at them as a whole. Without clicking on each marker to reveal the info window there's no way to tell anything about what's there other than the location.

One of the keys to a successful web application is to provide your users the information they want, both easily and quickly. For instance, if you come back to the map frequently, you would prefer to quickly pick out the points you haven't seen before, rather than hunt for and examine each marker to see the information. To give the map more visual information, let's let the geocachers add a custom icon for their finds. This will make the map more visually interesting and provide quick and easy information to the viewers.

By default, Google uses an inverted teardrop pin for marking points on a map, and up until now, this is what you've been using as well. Now, using Google's `GIcon` object, you can rid your map of the little red dots and customize them to use whatever image you like. Rather than looking at a red marker, you can add a small icon, as shown in Figure 3-14.

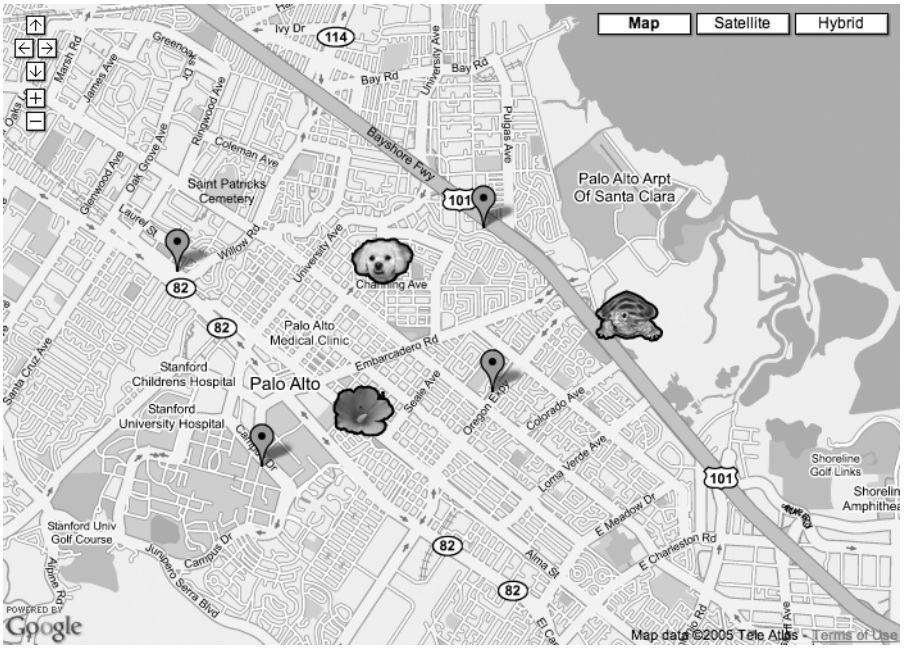


Figure 3-14. *Different marker icons on a map*

To use the `GIcon` object, you are required to set a minimum of three properties:

`GIcon.image`: URL of the image

`GIcon.iconSize`: Size of the image in pixels

`GIcon.iconAnchor`: Location of the anchor point

Also, because you're currently making use of the info window for each of your markers, you must specify the `infoWindowAnchor` property of the icon.

To get the URL for the `GIcon.image` property, you'll need to ask the geocaching users where their icon is by adding another element to the info window's form and then passing it through the `GET` parameters of your `GXmlHttp.request`. First, in the `click` event for the map in your `application.js` file, add the two highlighted lines from this code listing:

```
inputForm.innerHTML = '<fieldset style="width:150px;">'
          + '<legend>New Marker</legend>'
          + '<label for="found">Found</label>'
          + '<input type="text" id="found" name="m[found]" style="width:100%;"/>'
          + '<label for="left">Left</label>'
          + '<input type="text" id="left" name="m[left]" style="width:100%;"/>'
          + '<label for="left">Icon URL</label>'
          + '<input type="text" id="icon" name="m[icon]" style="width:100%"/>'
          + '<input type="submit" value="Save"/>'
          + '<input type="hidden" id="longitude" name="m[lng]" value="'+ lng +'"/>'
          + '<input type="hidden" id="latitude" name="m[lat]" value="' + lat + '"/>'
               + '</fieldset>':
```

Note For a complete working example of the code we've written so far in this chapter, look online at `http://book.earthcode.com/chap_three/final`.

Second, in the `createMarker()` function from Listing 3-5, add the following parameter shown in bold to the request:

```
var getVars =  "?m[found]=" + document.getElementById("found").value
        + "&m[left]=" + document.getElementById("left").value
        + "&m[icon]=" + document.getElementById("icon").value
        + "&m[lng]=" + lng
        + "&m[lat]=" + lat ;
```

Now the icon's URL can be entered and passed to the server. Of course, you need someplace to store the marker, so let's create a second migration to add the icon column to your `markers` table. From a command line, type `ruby script/generate migration AddIcon`, which will generate the file `/db/migrations/002_add_icon.rb`. Copy the following code into the body of the file:

```
class AddIcon < ActiveRecord::Migration
  def self.up
    add_column :markers, :icon, :string, :limit=>100, :default=>""
  end

  def self.down
    remove_column :markers, :icon
  end
end
```

Next, run the migration as usual to add the column to your `markers` table. There is also one small change you need to make to the `create` action in `ChapThreeController`, which is to return the icon in the JSON representing a newly created marker. The following line of code replaces the existing `res={:success=>true . . . }` in `ChapThreeController`:

```
res={:success=>true,:content=>"<div><strong>found </strong>#{marker.found}➥
</div><div><strong>left </strong>#{marker.left}</div>", :icon=>marker.icon}
```

Note that you don't have to modify the `list` action, because it is automatically rendering all the properties of the `marker` class in JSON.

Back to the `addMarkerToMap()` function. The `GIcon` objects are created as independent objects and passed in as the second parameter when creating a new `GMarker` object. The `GIcon` objects are reusable, so you do not need to create a new `GIcon` object for each new `GMarker` object, unless you are using a different icon for each marker, as you are doing in this example. To use the icons while retrieving the saved pins in Listing 3-8, add the icon URL as a third parameter to the `addMarkerToMap()` call:

```
var marker = addMarkerToMap(latlng, html, marker.icon);
```

Then create your `GIcon` object in the `addMarkerToMap()` function and assign it to the marker with the following changes shown in bold:

```
function addMarkerToMap(latlng, html, iconImage) {
     if(iconImage!='') {
         var icon = new GIcon();
         icon.image = iconImage;
         icon.iconSize = new GSize(25, 25);
         icon.iconAnchor = new GPoint(14, 25);
         icon.infoWindowAnchor = new GPoint(14, 14);
         var marker = new GMarker(latlng,icon);
     } else {
         var marker = new GMarker(latlng);
     }
     GEvent.addListener(marker, 'click', function() {
         var markerHTML = html;
         marker.openInfoWindowHtml(markerHTML);
     });
     return marker;
}
```

Additionally, when you create the new `GIcon` object in the `createMarker()` and `listMarkers()` functions, you'll need to pass the icon image into the `addMarkerToMap` call. In `createMarker()`, add the following:

```
var marker = addMarkerToMap(latlng, content, res.icon);
```

In `listMarkers()`, add this:

```
var iconImage =  marker.icon;
var marker = addMarkerToMap(latlng, html, iconImage);
```

Now when you regenerate the map and create new points, the icon from the URL will be used rather than the default red marker. The size you pick for your `GIcon` objects is based on a width and height in pixels. The preceding changes use an arbitrary `GSize` of 25 by 25 pixels for all of the icons. If the image in the URL is larger than 25 by 25 pixels, it will be squished down to fit.

Ajax with Prototype

As we indicated earlier, we are going to take a look at an alternative implementation of the Ajax call using Prototype. In the process, you've changed some code to take advantage of several other Prototype features. Listing 3-9 shows the `createMarker()` function reimplemented with Prototype.

Listing 3-9. *The createMarker() Function Reimplemented with Prototype*

```
function createMarker(){
  var lng = $("longitude").value;
  var lat = $("latitude").value;
  var formValues=Form.serialize('geocache-input');
```

```
  new Ajax.Request( 'create',
      { method: 'post',
        parameters: formValues,
        onComplete: function(request){
          //parse the result to JSON (simply by eval-ing it)
          res=eval( "(" + request.responseText + ")" );

          //check to see if it was an error or success
          if(!res.success) {
            alert(res.content);
          } else {
            //create a new marker and add its info window
            var latlng = new GLatLng(parseFloat(lat),parseFloat(lng));
            var marker = addMarkerToMap(latlng, res.content, res.icon);
            map.addOverlay(marker);
            map.closeInfoWindow();
          } // end of the res.success check
        }
  }); // end of the new Ajax.Request() call
}
```

Before you fire this up in your browser, there are two additional changes you must make. First, you have to include `prototype.js` in your `map.rhtml` file. You already have a `javascript_include_tag` in this file, so just add an additional argument to it:

```
<%=javascript_include_tag 'prototype', 'application' %>
```

Next, you have to give the form an ID so Prototype's `Form.serialize()` function (more on that later in this section) can identiffy the form. To do this, add a line to the `init()` function in `application.js` (only the bold line is new):

```
. . .
GEvent.addListener(map, "click", function(overlay, latlng) {
  //create an HTML DOM form element
  var inputForm = document.createElement("form");
  inputForm.id='geocache-input';
  . . .
```

Now that you've made the changes, check out the page in your browser. Is everything still working? Good. Let's look at some of the differences between the Prototype and the non-Prototype code:

1. `$()` *function replaces* `document.getElementById()`: Prototype's `$()` serves as a shorthand for `document.getElementById`. It can also do more (for example, it can get multiple IDs and return an array of DOM elements), but in this case you are just using it to make the code shorter.
2. `Form.serialize()` *replaces* `getVars`: Recall that in the previous implementation, you manually retrieved the inputs from your form and concatenated the values together to make a query string. Prototype recognizes that this is a common operation and provides the `Form.serialize()` method to do the work for you. Simply pass in the ID of the form, and the function returns a string with the values assembled, escaped, and formatted

A key advantage to using Prototype's `Form.serialize()` is that you don't need to change anything if you add new fields to the form. The function inspects the form and automatically serializes all the fields. Contrast this with the process of adding the icon field to the existing form and amending the `getVars` code to accommodate the new field.

3. *Different Ajax syntax*: Some developers prefer Prototype's `Ajax.Request` object due to the way it handles events. Recall that with Google's `GXmlHttp` object, you had to listen for the state change, then check to see if the new state equaled `4`, the "request is complete" state. Prototype encapsulates this process inside a more intuitive `onComplete` callback. Prototype's `Ajax.Request` object provides additional callbacks for `onSuccess`, `onFailure`, and so on.

   The Prototype Ajax object has a lot more tricks up its sleeve, including automatic parsing of JSON structures in the header (so you could move the success/failure indicator out of the body of the response and into the header), periodic calls, and global responders. If you want to learn more, the best documentation currently available for Prototype's Ajax object is at `http://www.sergiopereira.com/articles/prototype.js.html#Ajax.Request`.

Summary

Congratulations! You have your first interactive Google Maps web application. The ideas and techniques covered in this chapter can be applied to many different web applications, and the same basic interface can be used to mark any geographical information on a map. Want to chart the world's volcanoes? Just click away on the map and mark them down.

You may also want to build on the example in this chapter and incorporate some of the other features of the Google Maps API. For example, try retrieving only a specified list of markers, or maybe markers within a certain distance of a selected point. You could also improve the interface by adding listener events to trigger when you open and close an info window, or improve the server-side script by downloading and automatically resizing the desired icons. Throughout this book, you'll see a variety of other ways to improve your maps.

In the next chapter, we show you how you can use publicly available services to automatically plot markers on your map based not just on clicks, but also on postal and street addresses.

CHAPTER 4

Geocoding Addresses

As you've probably already guessed, the heart of most mapping applications is correlating your information with latitudes and longitudes for plotting on your map. Fortunately, geocoding services are available to help you convert postal addresses to precise latitude and longitude coordinates. For locations in the United States and Canada, these services make geocoding addresses relatively easy and quite accurate most of the time. In other parts of the world, the job can become much harder.

In this chapter, while building a store locator map, you'll do the following:

- Request information from geocoding web services and process their responses
- Learn the pros and cons of Google's new JavaScript-based geocoder, as well as suggestions on when to use it
- Learn about the output formats of three major U.S. geocoding services and one Canadian service

In the process of learning about geocoding web services, we'll utilize the following techniques:

- Populate a database table with YAML fixtures
- Execute chunks of Ruby code with full access to our Rails environment through our own custom Rake tasks
- Leverage two powerful Ruby standard library modules: OpenURI to issue HTTP requests to web services, and REXML to parse the web services' results

Preparing the Address Data

In this chapter, you're going to create a simple store location map using the postal address of each location in the store chain to map the markers. The important aspect about this kind of data is that it changes slowly over time. A few points are added every now and then as the chain of stores expands, but rarely are points removed. In general, it makes sense to precompute and persist information such as latitude and longitude for this type of data, as you'll see in the "Persisting Lookups" section later in this chapter.

For this example, we'll use the chain of stores and attractions known as Ron Jon Surf Shop, since its story appeals to our own entrepreneurial style:

> *It was 1959, and on the New Jersey shore a bright young man named Ron DiMenna was just discovering the sport of surfing with fiberglass surfboards. The pastime soon became a passion, and homemade surfboards would no longer do. When his father heard that Ron wanted his own custom surfboard from California, he suggested, "Buy three, sell two at a profit, then yours will be free." His dad was right and Ron Jon Surf Shop was born.*
>
> `http://www.ronjons.com`

With permission, we've taken the addresses of all of the Ron Jon properties from its web site and converted them into the sample YAML data file for this chapter. We'll use the Ron Jon data for the rest of the chapter.

We will continue to use the same Rails application we have been using in previous chapters. There is no need to create a new controller right now, as most of our work in this chapter will be from the command line. At the end of this chapter we will create a new controller, but for now we need a new model to work with the Ron Jon data.

Creating the Model

Let's put the Ron Jon information into the database where we can access it easily. The first step is to create a `Store` model. Enter this command from the command line: `ruby script/generate model Store`. You should see the following:

```
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/store.rb
      create  test/unit/store_test.rb
      create  test/fixtures/stores.yml
      exists  db/migrate
      create  db/migrate/003_create_stores.rb
```

Next, open up the `db/migrate/003_create_stores.rb` file, and replace its contents with the following:

```
class CreateStores < ActiveRecord::Migration
def self.up
  create_table :stores do |t|
    t.column :name, :string, :limit=>50
    t.column :address, :string, :limit=>100
    t.column :address2, :string, :limit=>100
    t.column :city, :string, :limit=>50
    t.column :state, :string, :limit=>2
    t.column :zip, :string, :limit=>9
```

```
      t.column :phone, :string, :limit=>15
      t.column :lat, :string, :limit=>20
      t.column :lng, :string, :limit=>20
    end
  end

  def self.down
    drop_table :stores
  end
end
```

Run the migration (`rake db:migrate` from the command line) to create the table.

Adding a full_address Method

As you can see from the migration, the address information for the stores is broken into parts—separate columns for city, state, ZIP code, and so on. However, most of the geocoders take full addresses as a single string. For convenient access, let's create a method on the `Store` model to get the full address. Open up the `app/models/store.rb` file and add the `full_address` method:

```
  def full_address
    "#{address}, #{city}, #{state}, #{zip}"
  end
```

Recall that in the absence of an explicit `return` keyword, Ruby methods return the value of the last evaluation in the method. Therefore, this method returns a string.

Populating the Table

We'll populate the newly created `stores` table by loading a YAML fixture into it. Typically, fixtures are used for testing. For this example, fixtures are an expedient way to load our shops into the table. Our Ron Jon store data is in the `stores.yml` file shown in Listing 4-1. Notice that while the `stores` table (as defined previously in the "Creating the Model" section) has columns for latitude and longitude (`lat` and `lng`), the YAML file doesn't contain that data. Populating these columns is the point of the geocoding we will do in this chapter.

Listing 4-1. *Ron Jon Properties (from* `http://www.ronjons.com/` *as of July 2006)*

```
store_1:
  name: The Original Ron Jon Surf Shop
  address: 901 Central Avenue
  city: Long Beach Island
  state: NJ
  zip: 08008
  phone: (609) 494-8844
store_2:
  name: One of a Kind Ron Jon Surf Shop
  address: 4151 North Atlantic Avenue
```

```
  city: Cocoa Beach
  state: FL
  zip: 32931
  phone: (321) 799-8888
store_3:
  name: Ron Jon Surf Shop - Sunrise
  address: 2610 Sawgrass Mills Circle
  address2: Suite 1415
  city: Sunrise
  state: FL
  zip: 33323
  phone: (954) 846-1880
store_4:
  name: Ron Jon Surf Shop - Orlando
  address: 5160 International Drive
  city: Orlando
  state: FL
  zip: 32819
  phone: (407) 481-2555
store_5:
  name: Ron Jon Surf Shop - Key West
  address: 503 Front Street`
  city: Key West
  state: FL
  zip: 33040
  phone: (305) 293-8880
store_6:
  name: Ron Jon Surf Shop - California
  address: 20 City Blvd.
  address2: West Building C Suite 1
  city: Orange
  state: CA
  zip: 92868
  phone: (714) 939-9822
store_7:
  name: Ron Jon Cape Caribe Resort
  address: 1000 Shorewood Drive
  city: Cape Canaveral
  state: FL
  zip: 32920
  phone: (321) 328-2830
```

To load this data, place the file in `/test/fixtures/stores.yml`. Note that this should replace the previous contents of `stores.yml`. Then invoke this Rake task on the command line: `rake db:fixtures:load FIXTURES=stores`. Now we have address data conveniently placed in a database table so we can access it through ActiveRecord. If you want to verify that the data is loaded correctly, fire up your command-line console (bold lines are your input):

```
>ruby script/console
Loading development environment.
>> Store.count
=> 7
>> exit
```

Next, you'll learn about geocoding web services.

■ **Tip** If you leave off the `FIXTURES=` argument from the command-line Rake invocation, Rake loads all your fixtures. You can also specify fixtures for multiple classes for Rake to load by passing a comma-delimited list. In this case, we are loading the fixtures for only one class, `Store`.

Using Geocoding Web Services

Converting postal addresses to precise latitude and longitude coordinates is made simple by a few good geocoding services. In this section, we're going to cover some of the most popular geocoding services we've found to date. (For an updated list of the geocoders we know about, check out our web site at `http://book.earthcode.com/geocoders`.)

However, before you dive into the available web services, there are a few server-side requirements you'll need to consider.

■ **Note** There are also sources of raw information that you can use to make your own geocoding solutions. So if you can't find a service that fits your needs, and you have a place to get some raw street data, see Chapter 11 for the basics of creating your own geocoding service.

Requirements for Consuming Geocoding Services

To consume the services, your environment needs to be connected to the Internet. For the examples in this chapter, you'll be using the OpenURI module to retrieve the XML information from the available services, and REXML to parse the XML you retrieve.

OpenURI

Many of these services require you to send a carefully crafted URL request to retrieve your information. For this purpose, you'll use Ruby's OpenURI module. OpenURI is part of Ruby's standard library, which means that it comes bundled with your Ruby installation. You don't need to install any additional gems to access it. The standard library includes a lot more tools for tasks such as network programming, accessing operating system services, and working with threads.

You'll be using a very small subset of OpenURI's functionality here, although we encourage you to look deeper into the module at `http://www.ruby-doc.org/stdlib/libdoc/open-uri/rdoc/`. You can find out more about the standard library at `http://www.ruby-doc.org/stdlib/`.

REXML

Most of the geocoding solutions you're about to investigate return an XML document as their result. To process these responses, you'll use REXML, another standard library module. REXML has been included in the standard Ruby distribution since Ruby 1.8. You can learn more about REXML at `http://www.germane-software.com/software/rexml/`. If you haven't worked with REXML before, the examples in this chapter will serve as a good foundation for the most common XML tasks: finding elements by tag name, iterating through sets of child elements, and extracting the value of specific text nodes.

The Google Maps API Geocoder

You'll begin your investigation of geocoding solutions with the Google Maps API geocoder (`http://www.google.com/apis/maps/documentation/#Geocoding_Examples`). Google claims that this solution should give street-level accuracy for the United States, Canada, France, Italy, Germany, and Spain. The Google developers expect to roll out support for more countries in the near future, so before you rule the Google geocoder out for a particular country, you might want to check either our web site (`http://googlemapsbook.com`) or the official API documentation.

Before June 2006, there was no official geocoder from Google. Many hacks used the Google Maps site's built-in geocoder and screen scraped the result. This was an explicitly unauthorized use of the service, and while we never heard of a crackdown on people doing this, Google did frown upon it. As a result, a number of alternative services popped up to fill the void, which we'll cover later in this section. Despite being late to the game, Google's geocoder has a number of really interesting features that none of the others have yet, and we'll highlight them throughout the discussion.

First, we'll look at the most basic method for accessing the geocoder: the HTTP-based lookup methods. You can also access the geocoder within JavaScript, as discussed later in this section and in Chapter 10's polyline example.

Like most of the other services we'll investigate, the Google method uses Representational State Transfer (REST) requests for accessing the service. REST is basically a simple HTTP request that passes `GET` parameters by appending key-value pairs such as `key=value&key2=value2` to the end of the request URL. Generally, a REST service returns some form of text-based data structure such as XML. Google's geocoder is very versatile; it can return Keyhole Markup Language (KML) (for use in Google Earth), JSON, and comma-separated values (CSV), in addition to XML.

THE ORIGIN OF REST

Representational State Transfer (REST) is a concept used to connect services in distributed systems such as the World Wide Web. The term originated in a 2000 doctoral dissertation about the Web written by Roy Fielding, one of the principal authors of the HTTP specification, and has quickly passed into widespread use in the networking community.

Fielding's vision of REST describes a strict abstraction of architectural principles. However, people now often loosely use the term to describe any simple web-based interface that uses XML and HTTP without the extra abstraction layers of approaches such as the SOAP protocol. As a result, these two different uses of REST cause some confusion in technical discussions. Throughout this book, we refer to it in the looser, more common meaning of REST.

Rails 1.2 has embraced REST wholeheartedly and makes creating your own REST services very easy.

Google has outdone many of the other geocoders on the market in that its geocoder returns an excellent answer when given fairly poor input. It does not require you to separate the street number, street name, direction (e.g., N, S, E, W), city, state, or even ZIP code. It simply takes what you give it, uses Google's extensive experience with understanding your search terms, and returns a best guess. Moreover, the service formats the input you give it into a nice, clean, consistent representation when it gives you the latitude and longitude answer. The geocoder even goes so far as to look past poor punctuation and strange abbreviations, which is great if you're taking the input from a visitor to your site.

Like most of the geocoders available on the market, Google limits the number of geocoding requests that you can make before it cuts you off. The Google limit is a generous 50,000 lookups per API key per day, provided you space them out at a rate of one every 1.75 seconds (as of the time of this publishing). To maximize this limit and your bandwidth, we suggest you use the server-side caching approach discussed in the "Persisting Lookups" section later in this chapter.

Google Geocoder Responses

Let's look at the Google geocoder's response for a sample query adapted from the official documentation:

```
http://maps.google.com/maps/geo?q=1600+AmPhItHEaTRe+PKway+Mtn+View+CA➥
&output=xml&key=your_api_key
```

This query returns the XML shown in Listing 4-2.

Listing 4-2. *Sample Response from Google's REST Geocoder*

```
<?xml version="1.0" encoding="UTF-8"?>
<kml>
 <Response>
  <name>1600 AmPhItHEaTRe PKway Mtn View CA</name>
  <Status>
   <code>200</code>
   <request>geocode</request>
  </Status>
  <Placemark>
   <address>
    1600 Amphitheatre Pkwy, Mountain View, CA 94043, USA
   </address>
   <AddressDetails>
    <Country>
     <CountryNameCode>US</CountryNameCode>
      <AdministrativeArea>
       <AdministrativeAreaName>CA</AdministrativeAreaName>
       <SubAdministrativeArea>
        <SubAdministrativeAreaName>Santa Clara</SubAdministrativeAreaName>
        <Locality>
         <LocalityName>Mountain View</LocalityName>
```

```
        <Thoroughfare>
         <ThoroughfareName>1600 Amphitheatre Pkwy</ThoroughfareName>
        </Thoroughfare>
        <PostalCode>
         <PostalCodeNumber>94043</PostalCodeNumber>
        </PostalCode>
       </Locality>
      </SubAdministrativeArea>
     </AdministrativeArea>
    </Country>
   </AddressDetails>
  <Point>
   <coordinates>-122.083739,37.423021,0</coordinates>
  </Point>
 </Placemark>
<Response>
</kml>
```

■ **Note** Want a format other than XML? The `output=` argument in the REST URL tells Google the format you want. For example, to get JSON, try `http://maps.google.com/maps/geo?q=1600+AmPhItHEaTRe+PKway+Mtn+View+CA&`**`output=json`**`&key=your_api_key`.

The response has three major components, which correspond to tags in the XML result shown in Listing 4-2:

- `name`: The name is exactly what you feed into the geocoder, so you know whether it interprets your URL encoding properly.
- `Status`: This is the response code, which indicates whether the lookup is successful or whether it failed. Table 4-1 lists the possible response codes and their meanings.
- `Placemark`: This is available only if the geocoding is successful and contains the information you're seeking. The placemark itself contains three important components:
  - `address`: The address is the full, nicely formatted string that Google actually uses after it cleans up the input you give it. This is useful for a number of reasons, including storing something clean in your database and debugging when the answers seem to come back incorrectly.
  - `Point`: The point is a coordinate in 3D space and represents longitude, latitude, and elevation. Elevation data may or may not be available for a given answer, so take a `0` with a grain of salt, as it is the default and is also returned if no data is available.
  - `AddressDetails`: This is a block of more complicated XML that uses a standard format called eXtensible Address Language (xAL). Unless you're interested in extracting the individual pieces of the address for storage in your database or formatting on your screen, you could safely ignore this chunk of XML and get away with using only the status, address, and point information.

■ **Note** Upon launch of its geocoder, Google developers stated that all elevations would return 0 and that they were unsure when they would be able to supply elevation data. Before you use any of the elevation data, check the official API documentation online or the official Google Maps API blog (`http://googlemapsapi.blogspot.com/`) to see which regions now have elevation data available.

Table 4-1. *Google Geocoder Response Codes*

Code	Constant Name	Description
200	G_GEO_SUCCESS	No errors occurred; the address was successfully parsed and its geocode has been returned.
500	G_GEO_SERVER_ERROR	A geocoding request could not be successfully processed, yet the exact reason for the failure is not known.
601	G_GEO_MISSING_ADDRESS	The HTTP q parameter is either missing or has no value.
602	G_GEO_UNKNOWN_ADDRESS	No corresponding geographic location can be found for the specified address. This may be due to the fact that the address is relatively new, or it may be incorrect.
603	G_UNAVAILABLE_ADDRESS	The geocode for the given address cannot be returned due to legal or contractual reasons.
610	G_GEO_BAD_KEY	The given key is either invalid or does not match the domain for which it was given.
620	G_TOO_MANY_QUERIES	You have accessed the service too frequently and are either temporarily or permanently blocked from further use.

XAL

Defining a uniform way to describe addresses across 200 countries is no easy task. Some countries use street names; others don't. Some place higher importance on the postal code; others insist that the street number is most important. Some divide their "administrative" zones into a two-tier system of province/city; others use more tiers, such as state/county/city/locality. Whatever format is chosen, it must take all of these situations into account. The Organization for the Advancement of Structured Information Standards (OASIS) has defined a format called xAL, which stands for eXtensible Address Language. Google has adopted it as a component of the XML response that its geocoder returns.

xAL uses a hierarchical data model (XML) since it seems like such a natural fit for addresses. For example, a country has states, a state has counties, a county has cities, a city has streets, and a street has individual plots of land. Some countries omit one or more of these levels, of course, but in general, it's not a problem.

However, you should realize that the xAL specification is designed to describe the address elements, not to be specific about the formatting and presentation of the address. There is no guarantee that the use of whitespace in the different elements will be consistent or even predictable, only that each type of data will be separated in a defined way. Using an XML-based format ensures that the data can be compared, sorted, and understood using simple programmatic methods.

For more information on xAL, visit the official site at `http://www.oasis-open.org/committees/ciq/ciq.html#6` or Google for the term **xAL address**.

Google Geocoder Requests

Now let's look at code that uses OpenURI to query the HTTP-based geocoding API and REXML to parse the answer. Listing 4-3 shows this code in the format of a Rake task. Place this code into a new file, `/lib/tasks/geocoding_tasks.rake`. Simply by putting the file there, you can run the code on the command line with `rake google_geocode`. Because Rake automatically includes your environment, you have access to everything your Rails application has, including ActiveRecord queries, which this code utilizes to loop through all the stores in the database.

Listing 4-3. *Using the Google Maps API Geocoder to Locate Stores*

```
require 'open-uri'
require 'rexml/document'

# Retrieve geocode information for all records in the Stores table
task :google_geocode => :environment do
  api_key="YOUR_KEY"

  (Store.find :all).each do |store|

    puts "\nStore: #{store.name}"
    puts "Source Address: #{store.full_address}"
      xml=open("http://maps.google.com/maps/geo?q=#{CGI.escape➥
(store.full_address)}&output=xml&key=#{api_key}").read
    doc=REXML::Document.new(xml)

    puts "Status: "+doc.elements['//kml/Response/Status/code'].text

    if doc.elements['//kml/Response/Status/code'].text != '200'
      puts "Unable to parse Google response for #{store.name}"
    else
      doc.root.each_element('//Response') do |response|
        response.each_element('//Placemark') do |place|
          lng,lat=place.elements['//coordinates'].text.split(',')

          puts "Result Address: " << place.elements['//address'].text
          puts "  Latitude: #{lat}"
          puts "  Longitude: #{lng}"
        end # end each place
      end # end each response
    end # end if result == 200
  end # end each store
end # end rake task
```

Insert your own Google Maps key in the `api_key="YOUR_KEY"` line near the top of the file. Use the same API key you got in Chapter 2 by signing up at `http://www.google.com/apis/maps/`. Be sure that there is no whitespace at the end of your API key; OpenURI complains if there is whitespace at the end of the URL you give it.

Let's step into the code. First we tell Ruby to require the OpenURI and REXML/Document modules from the standard library. Then we ask ActiveRecord for all the stores in the database, and loop through each. Within the loop, the `xml=open(...).read` line invokes the web service and places the results into the `xml` variable.

Next, you use `REXML::Document.new` to parse the XML result. There are either one or two inner loops, depending on Google's interpretation of the address information; there may be one or more `Response` elements, and within each of those, one or more `Placemark` elements. Finally, inside each of the `Placemark` elements, you get what you're looking for: the coordinates represented as comma-separated longitude and latitude values.

■ **Caution** Remember that REXML and XML in general are case-sensitive. The Google result XML uses `Placemark` (in title case). Querying for **placemark** or **PlaceMark** wouldn't work.

Go ahead and run your newly created Rake task from the command line: `rake google_geocode`. You should see the results shown in Listing 4-4 printed on your console. If you have trouble, double-check your API key as described earlier in this section.

Listing 4-4. *Output from the Google Geocoding Script*

```
Store: The Original Ron Jon Surf Shop
Source Address: 901 Central Avenue,  Long Beach Island , NJ, 08008
Status: 200
Result Address: 901 Central Ave, Barnegat Light, NJ 08008, USA
  Latitude: 39.748586
  Longitude: -74.111764
Result Address: 901 Central Ave, Surf City, NJ 08008, USA
  Latitude: 39.661016
  Longitude: -74.168010
Result Address: 901 Central Ave, Ship Bottom, NJ 08008, USA
  Latitude: 39.649667
  Longitude: -74.177253

Store: One of a Kind Ron Jon Surf Shop
Source Address: 4151 North Atlantic Avenue, Cocoa Beach, FL, 32931
Status: 200
Result Address: 4151 N Atlantic Ave, Cocoa Beach, FL 32931, USA
  Latitude: 28.356453
  Longitude: -80.608170

Store: Ron Jon Surf Shop - Sunrise
Source Address: 2610 Sawgrass Mills Circle, Sunrise, FL, 33323
Status: 200
Result Address: 2610 Sawgrass Mills Cir, Sunrise, FL 33323, USA
  Latitude: 26.150899
  Longitude: -80.316233
```

```
Store: Ron Jon Surf Shop - Orlando
Source Address: 5160 International Drive, Orlando, FL, 32819
Status: 200
Result Address: 5160 International Dr, Orlando, FL 32819, USA
  Latitude: 28.469873
  Longitude: -81.450311

Store: Ron Jon Surf Shop - Key West
Source Address: 503 Front Street, Key West, FL, 33040
Status: 200
Result Address: 503 Front St, Key West, FL 33040, USA
  Latitude: 24.560287
  Longitude: -81.805817

Store: Ron Jon Surf Shop - California
Source Address: 20 City Blvd., Orange, CA, 92868
Status: 200
Result Address: 100 City Blvd E, Orange, CA 92868, USA
  Latitude: 33.782107
  Longitude: -117.889878
Result Address: 2 City Blvd W, Orange, CA 92868, USA
  Latitude: 33.779838
  Longitude: -117.893568

Store: Ron Jon Cape Caribe Resort
Source Address: 1000 Shorewood Drive, Cape Canaveral, FL, 32920
Status: 200
Result Address: 699 Shorewood Dr, Cape Canaveral, FL 32920, USA
  Latitude: 28.402944
  Longitude: -80.604093
```

There are several interesting things to discuss in this result:

The Original Ron Jon Surf Shop: The original store lists Long Beach Island as the city. Google doesn't recognize this as a valid city and has instead used the ZIP code to determine which cities might be more appropriate. More important, each of the answers differs by at least a few tenths of a degree, and this is a significant difference (about ten kilometers). It's up to you to decide how to handle this situation. A few suggestions might be to always use the first answer and assume that this is the one Google thinks is best. Another option would be to average the answers. Lastly, you could treat multiple `Placemark` nodes as a geocoding failure and ignore all of the data.

Ron Jon Surf Shop – California: For the store in California, the web site lists the address as 20 City Blvd. but fails to give a direction. Google's two closest matches are 100 City Blvd. E. and 2 City Blvd. W. Both closest matches are returned in a separate `Placemark` node, and this is where the xAL data becomes very useful. Since each `Placemark` node is broken down in

a consistent way, you can determine in which component the answer differs from your input. Doing so will allow you to write code that will make educated decisions about what to do with the answers. Chapters 10 and 11 will provide some math to help you code an estimated location in situations like this.

Ron Jon Cape Caribe Resort: The Cape Caribe Resort doesn't geocode perfectly. This is probably because the resort is extremely new and the address hasn't yet been officially marked in the data that Google received. What you do in this case is again your decision, but our suggestion would be to assume that when you receive a single answer, it's the best you're going to get.

The Google JavaScript Geocoder

Google also provides a means to geocode user input without the intervention of your server. This is a first in the realm of geocoders and enables a few things that can be cumbersome with server-side geocoding, such as jumping a map directly to a user-provided address. The JavaScript geocoder is built directly into the Google Maps JavaScript API itself and makes Ajax calls directly to Google's servers from your visitor's computer.

The benefit is that it's quick and convenient because the API abstracts out all of the Ajax stuff, leaving you with a simple client-side JavaScript call. In addition to this, the latitude and longitude data can come back in such a way that it is trivial to place a point on your map using the API.

However, you need to keep in mind that while you don't have to contact *your own* server, you are talking to *a server*—Google's. So you still need to carefully design your application to minimize the wait times your visitor sees while using your application.

Good and Bad Reasons to Use the JavaScript Geocoder

Here are two cases where it might be appropriate to use the JavaScript geocoder:

- When the visitor is inputting an address that you then plot on a map but would never otherwise *store* for future use or display to another visitor. For example, this might be the case for a store locator that suggests locations based on proximity to a particular address.
- When you are getting a point from the user that is used *solely* for math calculations. We walk through an example of using the JavaScript geocoder in Chapter 10, where we show you how to add a corner to a polygon by either clicking on the map or entering an address into a text field.
- Once Google exposes its route calculation capabilities, it may become useful for computing one endpoint of the path on the fly, but this is pure speculation.

It is *not* appropriate to geocode a list of points (such as the Ron Jon stores) on the client-side simply because it's easy. Overall, this would be a waste of bandwidth. This in turn means a longer download time for your visitor and a less responsive map. Also, you definitely don't want to use this approach if the user is likely to be looking up the same thing over and over again.

Basically, while useful for quick-and-dirty mapping, the JavaScript geocoder isn't really useful for many professional map applications, since you'll almost always have a server-side

component. Accessing the REST-based geocoder from your server-side code will allow you to integrate and consolidate the geocoding calls with the rest of your application (e.g., combining geocoding with looking up store hours). Another benefit of using your own server is ensuring consistency by guaranteeing that your points are saved back to the server before showing them on a map, as discussed in Chapter 3. The same principle applies here. If you need to record any information back to your own server, you might as well use the REST-based geocoder to do the lookups and save yourself one Ajax call.

Client-Side Caching

Google has made a significant effort to limit the impact of lazy mappers (not you!) who will use the JavaScript geocoder just because it's easy. Aside from pleading with developers to "please cache your lookups" when it announced the geocoder, Google has integrated a client-side geocoding cache into the API. It is on by default and merely uses your visitors' RAM to store things they've previously looked up in case they look the same thing up again. You don't need to do anything special to use this cache, but there is something special you can do with it: you can seed it with information you already have. This means that you could precompute all of the addresses for your stores server-side, and then seed the client-side cache with the data. In certain applications, this could provide a huge speed boost for your map.

As of the time this book was written, the jury was still out on the best way to use some of these shiny new features. The official Google Maps API newsgroup is gushing with discussion about the best ways to do things and when to use the client-side cache and JavaScript geocoder to the best effect. We suggest that you check our web site (`http://googlemapsbook.com`) and the official documentation (`http://www.google.com/apis/maps/documentation/`) to see what the current best practices are.

The Yahoo Geocoding API

Currently, the Yahoo Geocoding API (`http://developer.yahoo.com/maps/rest/V1/geocode.html`) is really useful only for geocoding addresses in the United States, though with competition from Google, we're sure this will change. Before Google's geocoder came along, this was the geocoder of choice for many people doing U.S.-centric applications using both the Google Maps API and the Yahoo Maps API. The only real limitation is that you can make only 5,000 lookup requests per day (per IP address).

■ **Caution** The rate limit for Yahoo is based on a 24-hour window, not a calendar day. This window begins when you first send a request to the service and is reset 24 hours later. Also, the window does not "slide" (as it does with other services), meaning Yahoo does not a count the requests made in the *last* 24-hours, but rather during a fixed time frame. For a more thorough explanation of rate limiting in the Yahoo Geocoding API, visit `http://developer.yahoo.net/search/rate.html`.

To use the API, you must register for a Yahoo application ID (such as the Google API key from Chapter 2). Yahoo asks that you register a separate application ID for each of your applications. To obtain your application ID, visit `http://api.search.yahoo.com/webservices/`

register_application after logging in with your Yahoo account. If you do not have a Yahoo account, you'll need to create one before proceeding. Once you have your application ID, you'll need to include it in the requests to the service. Note that unlike the Google API key, Yahoo's application ID is not tied to a specific host name, so if you change the host name for an application, there's no need to get a new Yahoo application ID.

Like the Google geocoder, the Yahoo service is REST-based and requires you to append URL-encoded parameters on to the end of the request URL, as listed in Table 4-2.

Table 4-2. *Request Parameters to the Yahoo Geocoding API*

Parameter	Value	Description
appid	String (required)	The application ID you obtained from Yahoo.
street	String	The name and number of the street address. The number is optional but can improve accuracy.
city	String	The name of the city or town.
state	String	The name of the state, either spelled out or in its two-letter abbreviation, which is more accurate.
zip	Integer	The five-digit ZIP code. This could also be a string of five digits, a dash, and the four-digit extension.
location	String	A free-form string representing an address.*
output	String	The format for the output. Possible values are xml (the default) or php. If php is requested, the results will be returned in serialized PHP format.

* *The* location *parameter overrides the* street, city, state, *and* zip *parameters, and allows you to enter many different common formats for addresses. Thus, you are relying on Yahoo to parse the string accurately and as you intended, much like the Google service. Yahoo's geocoder is quite good at doing this parsing (for the same reasons as Google's geocoder), so unless you already have the data broken out into components, your best bet might be to use the single* location *parameter instead of the individual parameters.*

Yahoo Geocoder Responses

The following is an example of a request for geocoding the Apress headquarters:

```
http://api.local.yahoo.com/MapsService/V1/geocode?appid=YOUR_APPLICATION_ID➥
&street=2560+Ninth+Street&city=Berkeley&state=CA&zip=94710
```

This returns the XML shown in Listing 4-5.

Listing 4-5. *Sample Response from the Yahoo Geocoding API*

```
<?xml version="1.0" encoding="UTF-8"?>
<ResultSet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns="urn:yahoo:maps" xsi:schemaLocation="urn:yahoo:maps
           http://api.local.yahoo.com/MapsService/V1/GeocodeResponse.xsd">
   <Result precision="address" warning="The exact location could not be found,➥
here is the closest match: 2560 9th St, Berkeley, CA 94710">
```

```
      <Latitude>37.859569</Latitude>
      <Longitude>-122.291673</Longitude>
      <Address>2560 9TH ST</Address>
      <City>BERKELEY</City>
      <State>CA</State>
      <Zip>94710-2500</Zip>
      <Country>US</Country>
   </Result>
</ResultSet>
```

For the purposes of this discussion, we will ignore the `xmlns:` and `xsi:` namespaces. What we care about is the `Result` node and the elements inside it.

■ **Caution** As with the Google service, it is possible to get a `ResultSet` with multiple `Result` values. If you would like to see this, try geocoding the White House (1600 Pennsylvania Avenue, Washington, D.C.) while leaving out the ZIP code.

The `Result` node has two attributes in this case:

`precision`: This is a string indicating how accurate Yahoo thinks the answer is. This can be one of eight values at the moment: `address`, `street`, `zip+4`, `zip+2`, `zip`, `city`, `state`, or `country`. Changes to this list and additional information can be found in Yahoo's API developer documentation (`http://developer.yahoo.net/maps/rest/V1/geocode.html`).

`warning`: In our experience, nearly all Yahoo geocoding requests come back with the warning "exact location could not be found." This seems to occur for valid addresses whenever the capitalization of the street name, abbreviation of the street type, or spelling in the address don't exactly match the form in the database. For example, it can happen if a street name is given as a word (*Ninth* spelled out in full), when the Yahoo database has it listed as *9th*. Using the `warning` node to determine whether Yahoo's answer is a good match can be tricky; so for now, let's assume that the first answer in the result set is always the *best* answer (but not necessarily the *right* answer).

Next, we have the actual result fields corresponding to latitude, longitude, address, city, state, ZIP code, and country. Most of this data probably corresponds to the information you used to make the request; however, getting back all of this information is useful in picking the "right" answer in the event of Yahoo returning multiple matches. For now, the latitude and longitude fields are the ones you're most interested in, as those will be used to plot the Ron Jon store locations on your map.

Yahoo Geocoder Requests

So now that you have a handle on what you should be expecting out of the Yahoo API, let's automate this process with another Rake task. Listing 4-6 shows the script.

Listing 4-6. *Using the Yahoo Geocoding API to Locate the Stores*

```
# geocode all the stores in our database with Yahoo's geocoder
task :yahoo_geocode => :environment do
  (Store.find :all).each do |store|
    puts "\nStore: #{store.name}"
    url="http://api.local.yahoo.com/MapsService/V1/geocode?appid=YOUR_APPLICATION_KEY➥
&location=#{CGI.escape(store.full_address)}"
      xml=open(url).read
      doc=REXML::Document.new(xml)

      doc.root.each_element('//Result') do |result|
      puts "Result Precision: " << result.attributes['precision']
      if result.attributes['precision'] != 'address'
        puts "Warning: " << result.attributes['warning']
        puts "Address: " << result.elements['//Address'].text.to_s
      end
      puts "Latitude: " << result.elements['//Latitude'].text
      puts "Longitude: " << result.elements['//Longitude'].text

    end # end each result
  end  # end each store
end # end task
```

The code in Listing 4-6 represents another Rake task, so you can just add it to the `/lib/tasks/geocoding_tasks.rake` file. To run the code, type **rake yahoo_geocode** on the command line. The code is similar to that for the Google geocoder (Listing 4-3). In fact, this is a template we will use a few more times in this chapter, and one that will serve you well for most REST-based services that return XML.

Listing 4-7 gives the resulting output from Listing 4-6.

Listing 4-7. *Output from the Yahoo Geocoding Script*

```
Store: The Original Ron Jon Surf Shop
Result Precision: zip
Warning: The exact location could not be found. Here is the center of the ZIP code.
Address:
Latitude: 39.635
Longitude: -74.1884

Store: One of a Kind Ron Jon Surf Shop
Result Precision: address
Latitude: 28.356577
Longitude: -80.608069
```

```
Store: Ron Jon Surf Shop - Sunrise
Result Precision: address
Latitude: 26.156292
Longitude: -80.316945

Store: Ron Jon Surf Shop - Orlando
Result Precision: address
Latitude: 28.469972
Longitude: -81.450143

Store: Ron Jon Surf Shop - Key West
Result Precision: address
Latitude: 24.560448
Longitude: -81.805998

Store: Ron Jon Surf Shop - California
Result Precision: address
Latitude: 33.783329
Longitude: -117.890562

Store: Ron Jon Cape Caribe Resort
Result Precision: street
Warning:
Address: [600-699] SHOREWOOD DR
Latitude: 28.40232
Longitude: -80.59554

Result Precision: street
Warning:
Address: SHOREWOOD DR
Latitude: 28.40168
Longitude: -80.59774
```

Yahoo has two problems with this batch of addresses. The first store fails to geocode with accuracy beyond its ZIP code. The last entry, Cape Caribe Resort, failed to geocode any more accurately than the general location of the street. This seems to corroborate Google's answer quite nicely (remember that it gave us 699 Shorewood instead of 1000 Shorewood). For now, simply remember that you'll always need to do some sort of error checking on the results, or you might end up sending your customers to the wrong place. This entry also shows an example of multiple results being returned, as discussed earlier in the "Yahoo Geocoder Responses" section.

A possible solution to the ambiguous answer problem is to cross-reference (and average) the answers you get from one service (Google) with another (Yahoo). This is an onerous task if done for all of the data, but might be an excellent solution for your particular application if applied only to data that gives you grief.

Geocoder.us

Let's adapt our code for another U.S.-centric geocoding service. Geocoder.us is a very popular service and was introduced well before Yahoo's and Google's services hit the market. For a long while it was the measuring stick against which all other services were compared. The service was developed by two enterprising programmers who took the freely available 2004 U.S. Census Bureau's data and converted it into a web service.

■ **Note** The developers of Geocoder.us have made the Perl code that they wrote for their service available under an open source license and a module called `Geo::Coder::US`. If this interests you, Chapter 11 should also interest you. In Chapter 11, we dig deep into the U.S. Census data to build our own geocoder from scratch using Ruby instead of Perl.

Just as with the Google and Yahoo services, there are limitations to the Geocoder.us service. This free service cannot be used for commercial purposes and is rate-limited to prevent abuse, though the developers haven't published exactly what the limit is. You can purchase a high-volume or commercial account that will get you four lookups per penny (20,000 lookups for $50) with no rate limiting whatsoever.

Geocoder.us offers four different ways to access its web services: an XML-RPC interface, a SOAP interface, a REST interface that returns an RDF/XML document, and a REST interface that returns a plain-text CSV result. The accuracy, methods, and return values are equivalent across all of these interfaces. It's merely a matter of taste as to which one you'll use. For our example, we use the REST-based service and the CSV result (for some variety).

The following is an example of a Geocoder.us request for geocoding the Apress headquarters:

```
http://geocoder.us/service/csv/geocode?address=2560+Ninth+Street,+Berkeley+CA+94710
```

This returns the CSV string `37.859524,-122.291713,2560 9th St,Albany,CA,94710`. You can see that it has mistaken Berkeley for Albany, despite the fact that the ZIP codes match. The latitude and longitude are nearly identical to the results Yahoo gave.

Let's again reuse the code from Listing 4-3 and adapt it to suit this new service. As with the Google geocoder, only one parameter is passed into this REST service, and it is called `address`. At minimum, either a city and a state or a ZIP code must be contained in the `address` parameter. Listing 4-8 shows the adapted code. To run the code, enter **rake geocoder_us** on the command line.

■ **Caution** The code in Listing 4-8 takes a while to run. We'll discuss why in a moment, but for now be patient.

Listing 4-8. *Using the Geocoder.us Service to Locate the Stores*

```
task :geocoder_us => :environment do
  (Store.find :all).each do |store|
    puts "\nStore: #{store.name}"
```

```
    url="http://geocoder.us/service/csv/geocode?address=➥
#{CGI.escape(store.full_address)}"
    res=open(url).read.chomp
    lat,lng,address,city,state,zip=res.split(',')

    puts "Latitude: #{lat}"
    puts "Longitude: #{lng}"

    end  # end each store
end # end task
```

The only real difference here is with the CSV-style response. We've parsed it by using Ruby's array assignments to named variables. Listing 4-9 shows the output of the code in Listing 4-8.

Listing 4-9. *Output from the Geocoder.us Script*

```
Store: The Original Ron Jon Surf Shop
Latitude: 39.649509
Longitude: -74.177136

Store: One of a Kind Ron Jon Surf Shop
Latitude: 28.356433
Longitude: -80.608227

Store: Ron Jon Surf Shop - Sunrise
Latitude: 26.150513
Longitude: -80.316476

Store: Ron Jon Surf Shop - Orlando
Latitude: 28.466795
Longitude: -81.449860

Store: Ron Jon Surf Shop - Key West
Latitude: 24.560083
Longitude: -81.806069

Store: Ron Jon Surf Shop - California
Latitude: 33.781086
Longitude: -117.892520

Store: Ron Jon Cape Caribe Resort
Latitude: 2: couldn't find this address! sorry
Longitude:
```

When executing the code, the first thing you'll probably notice is that this request takes a long time to run. We believe this is a result of Geocoder.us rate limiting being based on

requests per minute instead of requests per day. In our testing, it took well over a minute to geocode just the seven points in our `stores` table.

The next thing you'll see if you look carefully is that the latitude and longitude results are the same as those from Yahoo only to three decimal places (on average). This is not a large difference and is the result of using different interpolation optimizations on the same data set, which we discuss in Chapter 11.

Notice that the original store has given us a single answer this time, instead of multiple answers, and that the resort has given us grief yet again, except in this case we didn't even get a best guess.

■ **Note** To determine just how large a distance difference the various geocoders give you for each of the results, you'll need to use the spherical distance equations (such as the Haversine method) we provide in Chapter 10.

Geocoder.ca

Geocoder.ca is similar to the service provided by Geocoder.us, but it is specifically targeted at providing information about Canada. (This service is in no way affiliated with Geocoder.us, and it uses a completely different data set provided by Statistics Canada.)

The people behind Geocoder.ca built it specifically for their own experiments with the Google Maps API when they had trouble finding a timely, accurate, and cost-effective solution for geocoding Canadian addresses. They obtained numerous sources of data (postal, census, and commercial) and cross-referenced everything to weed out the inevitable errors in each set. This means that the Geocoder.ca service is quite possibly the *most* accurate information for Canada so far. (However, now that Google's solution covers Canada with relatively good accuracy, we're afraid that this extremely comprehensive service will become marginalized.)

Geocoder.ca provides a lot of neat features such as intersection geocoding, reverse geocoding, and a suggestion system for correcting mistyped (or renamed) street names—none of which are provided by Google's geocoder, or any other geocoder for that matter. We don't cover any of these alternative features in this chapter, but you can find more information about Geocoder.ca at `http://geocoder.ca/` if you're interested.

Remember that there is still no free lunch, so as with the other services there are also limitations on the Geocoder.ca service. The free service is limited to between 500 and 2,000 lookups per day per source IP address, depending on server load (light days you get more, heavy days less). The developers are willing to extend the limits for nonprofit organizations, but everyone else will need to purchase an account for commercial uses. The cost is currently the same as Geocoder.us: 20,000 lookups for $50. Purchasing a commercial account might be an excellent way to cross-reference Google's multiple-result answers quickly, cheaply, and effectively.

An example of a query for geocoding the CN Tower in Toronto, Ontario, is as follows:

```
http://geocoder.ca/?&stno=301&addresst=Front%2BStreet%2BWest&city=Toronto➥
&prov=CN&postal=M5V2T6&geoit=XML
```

This yields the exceedingly simple XML result in Listing 4-10.

Listing 4-10. *Sample Response from Geocoder.ca*

```
<?xml version="1.0" encoding="UTF-8" ?>
<geodata>
        <latt>43.643865000</latt>
        <longt>-79.388545000</longt>
</geodata>
```

Notice that the XML response uses `latt` and `longt`. The trailing `t` is easy to miss when reading the raw XML.

For an example, the Ron Jon Surf Shop data will not work, since the chain has yet to open a store in Canada. Instead, we'll again use the CN Tower in Toronto, Ontario. The address for the CN Tower is 301 Front Street West, Toronto, Ontario M5V 2T6, Canada. Listing 4-11 shows a small Rake task for geocoding this single address, which could easily be looped and abstracted as in previous examples to do multiple addresses. To run this code, append it to your `lib/tasks/geocoding_tasks.rake` file, and type **rake geocoder_ca** on the command line. Feel free to substitute your own address if you live in the Great White North or know someone who does.

Listing 4-11. *Using Geocoder.ca to Locate the CN Tower in Toronto*

```
task  :geocoder_ca => :environment do
  street_no = "301"
  street = "Front Street West"
  city = "Toronto"
  prov = "ON"
  postal = "M5V2T6"

  url="http://geocoder.ca/?"
  url << "&stno=" << CGI.escape(street_no)
  url << "&addresst=" << CGI.escape(street)
  url << "&city=" << CGI.escape(city)
  url << "&prov=" << CGI.escape(prov)
  url << "&postal=" << CGI.escape(postal)
  url << "&geoit=XML"

  xml=open(url).read
  doc=REXML::Document.new(xml)

  puts "The CN tower is located here:"
  puts "Latitude: " << doc.root.elements['//latt'].text
  puts "Longitude: " << doc.root.elements['//longt'].text
end
```

The most important lines in Listing 4-11 are highlighted in bold. The first is that the Geocoder.ca service prefers you to split the street number from the street name. This isn't strictly necessary, but it does imply that greater accuracy can be achieved by doing so. The

second is that the `geoit` parameter *must* be included. At this point, there is no alternative value for this parameter, but there probably will be in the future. Lastly, when parsing the results, again, remember that the XML response uses `latt` and `longt`.

Listing 4-12 shows the output from Listing 4-11.

Listing 4-12. *Output from our Geocoder.ca Script*

```
The CN tower is located here:
Latitude: 43.643865
Longitude: -79.388545
```

If you compare this answer with the one Google gives you by clicking on a map (`43.642411,-79.386649`), you see that Geocoder.ca has done an excellent job of finding the correct coordinates for the CN Tower.

Services for Geocoding Addresses Outside Google's Coverage

For addresses outside the set provided by Google's geocoder, the job becomes much more difficult due to the lack of good, freely available data. In Chapter 11, you'll see how to create your own service from some sources of free data for the UK and the United States. Maybe some of you will be inspired to find data for your country and create a service for the rest of us.

For now, however, we're simply going to share a few of the geocoding services we've found for areas outside Google's coverage area. We can't guarantee the accuracy or completeness of data from these services, since we don't have any real addresses to test with, or enough knowledge of the local geography to wing it. We'll try to keep an updated list of services on our web site at `http://googlemapsbook.com/geocoders` as we hear about them. Let us know if you find or make more.

GeoNames

GeoNames has quite a few web services that might fit your needs. There is a full-text search of its database of place names, landmarks, and other geopositional data at `http://www.geonames.org/export/geonames-search.html`. However, you can also find (partial) postal code lookups for many countries (currently more than 50), as well as reverse geocoding solutions for finding the name of the country or closest named feature for a given latitude and longitude.

ViaMichelin

One interesting solution for geocoding addresses in western Europe is ViaMichelin (`http://www.viamichelin.com`). The company that runs this service is part of the same company that makes Michelin tires (remember the Michelin Man?). The service offers route calculation, geocoding, and even an alternative source of map data. ViaMichelin is in competition with Google when it comes to maps, but for European locations where Google does not yet have geocoding services, the ViaMichelin solution could mean the difference between a successful project and a failure.

Bulk Geocoders

There are bulk geocoding services available that will accept a CSV or an Excel file from you, determine latitude and longitude to the best of their ability, and give you the results a few hours to a few days later. Try a Google search on **bulk geocoding services** if you are interested in this option. The quality of the data varies (with provider, price, and country), so we suggest that you do your research before hiring one of them to geocode your points.

Persisting Lookups

As programmers, we hate wasting resources, and as service providers, we hate having our resources wasted. Therefore, for many of the examples in the rest of this book, you'll be computing the latitude and longitude once, and storing that information for later use. This saves your bandwidth by not requiring unnecessary OpenURI/API requests, and saves the bandwidth of the services you'll be using for geocoding. It also provides a much faster user experience for your map visitors, which is almost always the single largest factor in determining the success of a web site or service. Finally, it just makes good architectural sense to perform the computationally intensive geocoding operation once upfront and store the result for reuse.

■ **Caution** Persisting your point data is not *always* the right answer. Sometimes a point on the map is essentially a "one-time use" affair—for example, figuring out the distance between two arbitrary addresses. Another example is real-time data, such as plotting the current position of a GPS device. (Note, however, that Google restricts usage of the map API for real-time data. See Google's terms of service at `http://www.google.com/apis/maps/terms.html` for more information.)

Remember when we created the migration for the `stores` table, and we included `lat` and `lng` columns? Now is when we use those columns. Starting with the code in Listing 4-6, let's create another Rake task (Listing 4-13) which simply sets the latitude and longitude for each store after geocoding its address. After you run the task (`rake google_persist` on the command line), the `stores` table's `lat` and `lng` columns will be populated with the appropriate values. You will use the resulting latitude and longitude values in the next section.

Listing 4-13. *Persist Your Geocoding Data in the stores Table*

```
task :google_persist => :environment do
  api_key="YOUR_KEY"

  (Store.find :all).each do |store|

    puts "\nStore: #{store.name}"
    puts "Source Address: #{store.full_address}"

    xml=open("http://maps.google.com/maps/geo?q=#{CGI.escape(store.full_address)}➥
&output=xml&key=#{api_key}").read
    doc=REXML::Document.new(xml)
```

```
    puts "Status: "+doc.elements['//kml/Response/Status/code'].text

    if doc.elements['//kml/Response/Status/code'].text != '200'
      puts "Unable to parse Google response for #{store.name}"
    else
      lng,lat=doc.root.elements['//coordinates'].text.split(',')
      store.lat=lat
      store.lng=lng
      store.save
    end # end if result == 200
  end # end each store
end # end rake task
```

There are two changes we've made to the original task. First, we've taken out the looping for multiple results per address. If there are multiple results, we're just grabbing the first (and presumably best) result that Google gives us. Secondly, we're setting `store.lat` and `store.lng`, and calling `store.save`. As always, ActiveRecord makes update operations on your model extremely easy.

You can check whether the geocodings persist properly by invoking `ruby script/console` from the command line. Your input is in bold in this example interactive session:

```
>ruby script/console
Loading development environment.
>> (Store.find :all).each {|s| puts "#{s.lat},#{s.lng}"}
39.748586,-74.111764
28.356453,-80.60817
26.150899,-80.316233
28.469873,-81.450311
24.560287,-81.805817
33.782107,-117.889878
28.402944,-80.604093
... (Rails will also print out a hash representation of the stores)
```

A store finder page exemplifies the kind of application where doing the lookups once upfront and storing the results in your database makes perfect sense. After all, the data set is the same for everyone coming to the page. Stores are rarely added, deleted, or modified. It is clearly preferable to geocode the addresses once and reuse the location information whenever a visitor comes to your store-finder web page.

Building a Store Location Map

Now that you have your stores and their latitude and longitude coordinates, you're ready to make your map. This will be a very basic map, but it serves our demonstration nicely. You'll customize the marker `GIcon` using the Ron Jon Surf Shop logo, and use the info window to display the store's address and phone number to visitors when they click the marker.

Now it's time to create a new controller with the `ruby script/generate controller chap_four` command from the command line. All you need is a `map` action on `ChapFourController`, an associated `map` view, and a modified `application.js` file to plot the points with a custom logo. The following listings show the controller, the view, and the modified `application.js` file, respectively.

Note that the `application.js` file should replace any existing `application.js` file you have. Listing 4-14 shows the controller.

Listing 4-14. *app/controllers/chap_four_controller.rb*

```
class ChapFourController < ApplicationController
  def map
    @stores = Store.find :all
  end
end
```

Listing 4-15 shows the view.

Listing 4-15. *app/views/chap_four/map.rhtml*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"➥
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <script src="http://maps.google.com/maps?file=api&v=2&key=YOUR_API_KEY" ➥
type="text/javascript"></script>
  <%=javascript_include_tag 'application' %>
  <script type="text/javascript">
  var stores=<%=(@stores.collect {|s|s.attributes}).to_json%>;   </script>
</head>
<body>
  <div id="map" style="width: 800px; height: 500px"></div>
</body>
</html>
```

Listing 4-16 shows the modified `application.js` file.

Listing 4-16. *public/javascript/application.js*

```
var centerLatitude = 40.6897;
var centerLongitude = -95.0446;
var startZoom = 3;

var map;

var RonJonLogo = new GIcon();
RonJonLogo.image = 'http://book.earthcode.com/chapter4/StoreLocationMap/➥
ronjonsurfshoplogo.png';
RonJonLogo.iconSize = new GSize(48, 24);
RonJonLogo.iconAnchor = new GPoint(24, 14);
RonJonLogo.infoWindowAnchor = new GPoint(24, 24);

function addMarker(latitude , longitude, description) {
  var marker = new GMarker(new GLatLng(latitude, longitude), RonJonLogo);
```

```
  GEvent.addListener(marker, 'click',
    function() {
      marker.openInfoWindowHtml(description);
    }
  );

  map.addOverlay(marker);
}

function init() {
  map = new GMap2(document.getElementById("map"));
  map.addControl(new GSmallMapControl());
  map.setCenter(new GLatLng(centerLatitude, centerLongitude), startZoom);

  // loop through and add each store to the map
  for(i=0; i < stores.length; i++) {
    addMarker(stores[i].lat , stores[i].lng, stores[i].name);
  }
}

window.onload = init;
```

You should now be able to view the completed map in your browser at `http://localhost:3000/chap_four/map` as shown in Figure 4-1.

Figure 4-1. *The completed map of the Ron Jon Surf Shop U.S. locations*

There you have it. The best bits of all of our examples so far combined into a map application. Data is geocoded, stored in our database, and plotted quickly, based on JSON generated from the `stores` table.

Summary

This chapter covered using geocoding services with your maps. It's safe to assume that you'll be able to adapt the general ideas and examples here to use almost any web-based geocoding service that comes along in the future. From here on, we'll assume that you know how to use these services (or ones like them) to geocode and store your information efficiently.

This ends the first part of the book. In the next part, we'll move on to working with third-party data sets that have hundreds of thousands of points. Our examples will use the FCC's antenna structures database that currently numbers well over 100,000 points.

PART 2

Beyond the Basics

CHAPTER 5

Manipulating Third-Party Data

In this chapter, we cover two of the most popular ways of obtaining third-party data for use on your map: using downloadable character-delimited text files and screen scraping. To demonstrate manipulating data, we use a single database in this and the next two chapters (the FCC Antenna Structure Registration database). In the end, you'll have an understanding of the data that will be used for the sample maps, as well as how the examples might be generalized to fit your own sources of raw information.

In Appendix A, you'll find a list of other sources of free information that you could harvest and combine to make maps. You might want to thumb to this appendix to see some other neat things you could do in your own experiments and try applying the tips and tricks presented in this chapter to some other source of data. The scripts in this chapter should give you a great toolbox for harvesting nearly any data source, and the ideas in the next two chapters will help you make an awesome map, no matter how much data there is.

In this chapter, you'll learn how to do the following:

- Use Ruby command-line scripts to preprocess text files for import into the database
- Use MySQL command-line tools to import data much more quickly than with ActiveRecord
- Assemble multiple large files into a single file for import
- Parse the visible HTML from a web site and extract the parts that you care about—a process called *screen scraping*

Using Downloadable Text Files

For the next three chapters, we're going to be working with the U.S. Federal Communications Commission (FCC) Antenna Structure Registration (ASR) database. This database will help you highlight many of the more challenging aspects of building a professional map mashup.

So why the FCC ASR database? There are several reasons:

- The data is free to use, easy to obtain, and well-documented. This avoids copyright and licensing issues for you while you play with the data.
- There is a lot of data, allowing us to discuss issues of memory consumption and interface speed. At the time of publication, there were more than 120,000 records.

- The latitudes and longitudes are already recorded in the database, removing the need to cover the geocoding process, which we've already discussed in depth.
- None of the preceding items are likely to have changed since this book was published, serving as a futureproof example that should still be relevant as you read this.
- The maps you can make with this data look extremely cool (Figure 5-1).

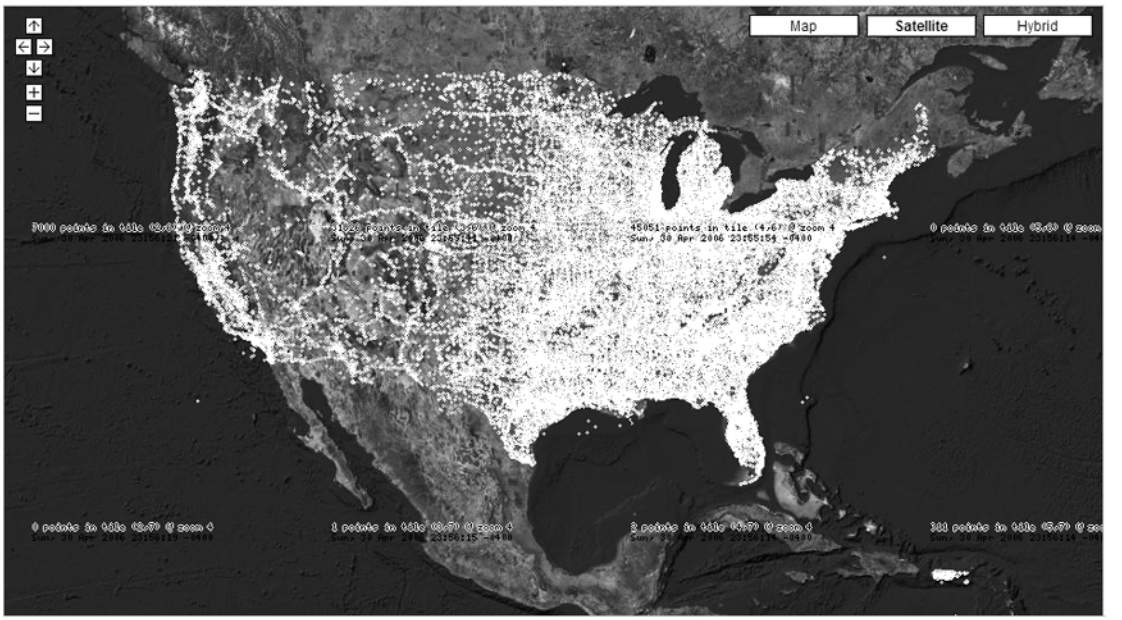


Figure 5-1. *A map built with FCC ASR data (which you will build in Chapter 7)*

Downloading the Database

The first thing you need to do is obtain the FCC ASR database. It's available at `http://wireless.fcc.gov/uls/data/complete/r_tower.zip`. This file is approximately 20MB when compressed.

After you've downloaded the file, unpack it and transfer `RA.dat`, `EN.dat`, and `CO.dat` into your working folder. You won't need the rest of the files for this experiment, although they do contain interesting data. If you're interested in the official documentation, feel free to visit `http://wireless.fcc.gov/cgi-bin/wtb-datadump.pl`.

Tables 5-1 through 5-3 outline the contents of the `RA.dat`, `EN.dat`, and `CO.dat` files. `RA.dat` (Table 5-1) is the key file, and the one you will use to bind the three together. It lists the unique identification numbers for each structure, as well as the physical properties, such as size and street address. `EN.dat` (Table 5-2) outlines the ownership of each structure, and `CO.dat` (Table 5-3) outlines the coordinates for the structure in latitude and longitude notation. The "Used in Our Example?" column in each table indicates the data you will be using.

Table 5-1. *RA.dat: Registrations and Applications*

Column	Data Element	Content Definition	Used in Our Example?
0	Record type	char(2)	--
1	Content indicator	char(3)	--
2	File number	char(8)	--
3	Registration number	char(7)	Yes
4	Unique system identifier	numeric(9)	Yes
5	Application purpose	char(2)	--
6	Previous purpose	char(2)	--
7	Input source code	char(1)	--
8	Status code	char(1)	--
9	Date entered	mm/dd/yyyy	--
10	Date received	mm/dd/yyyy	--
11	Date issued	mm/dd/yyyy	--
12	Date constructed	mm/dd/yyyy	--
13	Date dismantled	mm/dd/yyyy	--
14	Date action	mm/dd/yyyy	--
15	Archive flag code	char(1)	--
16	Version	integer	--
17	Signature first name	varchar(20)	--
18	Signature middle initial	char(1)	--
19	Signature last name	varchar(20)	--
20	Signature suffix	varchar(3)	--
21	Signature title	varchar(40)	--
22	Invalid signature	char(1)	--
23	Structure_street address	varchar(80)	Yes
24	Structure_city	varchar(20)	Yes
25	Structure_state code	char(2)	Yes
26	Height of structure	numeric(5,1)	Yes
27	Ground elevation	numeric(6,1)	Yes
28	Overall height above ground	numeric(6,1)	Yes
29	Overall height AMSL	numeric(6,1)	Yes
30	Structure type	char(6)	Yes
31	Date FAA determination issued	mm/dd/yyyy	--
32	FAA study number	varchar(20)	--
33	FAA circular number	varchar(10)	--
34	Specification option	integer	--
35	Painting and lighting	varchar(100)	--
36	FAA EMI flag	char(1)	--
37	NEPA flag	char(1)	--

Table 5-2. *EN.dat: Ownership Entity*

Column	Data Element	Content Definition	Used in Our Example?
0	Record type	char(2)	--
1	Content indicator	char(3)	--
2	File number	char(8)	--
3	Registration number	char(7)	Yes
4	Unique system identifier	numeric(9,0)	Yes
5	Entity type	char(1)	--
6	Licensee ID	char(9)	--
7	Entity name	varchar(200)	Yes
8	First name	varchar(20)	--
9	MI	char(1)	--
10	Last name	varchar(20)	--
11	Suffix	char(3)	--
12	Phone	char(10)	--
13	Internet address	varchar(50)	--
14	Street address	varchar(35)	Yes
15	P.O. box	varchar(20)	--
16	City	varchar(20)	Yes
17	State	char(2)	Yes
18	ZIP code	char(9)	Yes
19	Attention	varchar(35)	--

Table 5-3. *CO.dat: Physical Location Coordinates*

Column	Data Element	Content Definition	Used in Our Example?
0	Record type	char(2)	--
1	Content indicator	char(3)	--
2	File number	char(8)	--
3	Registration number	char(7)	Yes
4	Unique system identifier	numeric(9)	Yes
5	Coordinate type	char(1)	--
6	Latitude degrees	integer	Yes
7	Latitude minutes	integer	Yes
8	Latitude seconds	numeric(4,1)	Yes
9	Latitude direction	char(1)	Yes
10	Latitude_total_seconds	numeric(8,1)	--
11	Longitude degrees	integer	Yes
12	Longitude minutes	integer	Yes
13	Longitude seconds	numeric(4,1)	Yes
14	Longitude direction	char(1)	Yes
15	Longitude_total_seconds	numeric(8,1)	--

As you can see, we're not concerned with most of the data that is available in this database. Our main interest is the location and physical properties of each structure.

Working with Files

Now that you know what you want to use from the massive amount of data provided by the FCC, you need to break out those bits into something useful. For this task, we're going back to the command line. Command-line scripts are appropriate for relatively long-running operations that you don't want to invoke from the web interface. Tasks like these highlight Ruby's utility as a scripting language. As a stepping stone to our actual data import, let's make sure we can run a script and access some files. The code in Listing 5-1 shows this process.

Listing 5-1. *Reading and Writing a File*

```
# Simple test to read and write a file
input_file =  'RA.dat'
output_file = 'output.dat'

out = File.new(output_file, "w")
count=0
IO.foreach(input_file) do |line|
  columns = line.split('|')
  out.puts columns.values_at(1,2,3).join('|')
  count += 1
  break if count >=10
end
out.close
puts "done, see #{output_file}"
```

To use this code, save it as `file_test.rb` in the same directory as the `RA.dat` file you just downloaded and unzipped. Navigate to the directory on the command line and execute it with `ruby file_test.rb`. Then look for the newly created `output.dat` file in the same directory; if it contains the first three fields of the first ten lines of `RA.dat`, congratulations. You've successfully read, parsed, and written files using a stand-alone Ruby script.

■ **Tip** If you are running Linux (or Unix), you can begin stand-alone Ruby files with `#!/usr/bin/env ruby -w`, so Linux knows to execute the file with the Ruby interpreter. This is known as the *shebang* line. If you use the shebang line, you also need to ensure the file is executable (to make it executable, type **chmod +x file_test.rb**). You can then run the file without specifying the Ruby interpreter, that is, `./file_test.rb`. For more information on the shebang line, see `http://en.wikipedia.org/wiki/Shebang_%28Unix%29`.

You'll use code like this often when working with third-party data, so we wanted you to see the file operations in their simplest form. We'll use a variation on this pattern in the next section.

Correlating and Importing the Data

We need to get this data out of flat files and into the database. This may seem a straightforward enough operation, since we do this all the time with Rails' test fixtures. This particular set of data presents some special challenges, however, which make it more difficult to work with:

- *Size*: The files are between 100,000 and 200,000 lines each. If you've ever tried to import that many fixtures into your environment, you know that it can take a while.
- *Multiple files*: The data is inconveniently spread across three separate files, even though there is a 1:1:1 relationship between the items in each file. This makes it more difficult to import.
- *Unsorted*: If the three files were sorted identically, joining them up would be easy and quick: step through each file line by line, joining them into a complete object as you go. Unfortunately, the data is sorted differently in each file, so joining them up is not so trivial.

How to deal with these challenges? To deal with the sheer quantity of data, we will use the MySqlImport command-line tool. MySqlImport loads data at least an order of magnitude faster than raw SQL inserts (not to mention the ActiveRecord overhead that Rails imposes). How much faster is MySqlImport relative to ActiveRecord? In this example, we import just over 120,000 records. With MySqlImport, the process takes just over a minute. With ActiveRecord inserts, the same dataset takes one and a half *hours*.

In case you are an ActiveRecord purist (or if nearly two orders of magnitude speedup doesn't impress you), we present a pure ActiveRecord import in the sidebar titled "A Pure ActiveRecord Import" later in the chapter.

■ **Tip** PostgreSQL users have access to roughly equivalent functionality with the `copy` command. See `http://www.postgresql.org/docs/8.1/interactive/sql-copy.html` for more information if you are running PostgreSQL.

To deal with the multiple unsorted files, there are at least two possible approaches:

- *Import into three separate tables*: To make this work, you need to create appropriate ActiveRecord associations among the three tables. Then you can query the objects as usual and navigate between location, structure, and owner objects.
- *Assemble the files in-memory*: This involves loading all three files simultaneously into memory as Hashtables. The hashes act as an in-memory index, from which you can assemble the three files into one for import. This approach uses just one table and one model.

Both these approaches will work, and we will demonstrate both of them. However, assembling the files in-memory turns out to be the simpler solution, and the one we carry forward in coming chapters.

Before moving on, let's talk briefly about one approach we definitely recommend you *avoid*: importing one file into a table and subsequently updating the table with values from

the other files. While this may seem reasonable enough, it's extraordinarily slow: eight hours or more to complete the import. The two approaches we use here will only take a few minutes of processing time.

Let's get started with the three-table approach. We'll use the same project we've been working with in previous chapters. Create a new migration (`ruby script/generate migration CreateFCCTables` from the command line), and copy the code in Listing 5-2 into the migration file.

Listing 5-2. *Migration Code for the Example*

```
class CreateFccTables < ActiveRecord::Migration
  def self.up
    create_table :fcc_locations do |t|
      t.column :unique_si, :integer
      t.column :lat_deg, :integer
      t.column :lat_min, :integer
      t.column :lat_sec, :float
      t.column :lat_dir, :string, :limit=>1
      t.column :latitude, :float
      t.column :long_deg, :integer
      t.column :long_min, :integer
      t.column :long_sec, :float
      t.column :long_dir, :string, :limit=>1
      t.column :longitude, :float
    end
    add_index :fcc_locations, :unique_si

    create_table :fcc_owners do |t|
      t.column :unique_si, :integer
      t.column :name, :string, :limit=>200
      t.column :address, :string, :limit=>35
      t.column :city, :string, :limit=>20
      t.column :state, :string, :limit=>2
      t.column :zip, :string, :limit=>10
    end
    add_index :fcc_owners, :unique_si

    create_table :fcc_structures do |t|
      t.column :unique_si, :integer
      t.column :address, :string, :limit=>80
      t.column :city, :string, :limit=>20
      t.column :state, :string, :limit=>2
      t.column :height, :float
      t.column :elevation, :float
      t.column :ohag, :float
      t.column :ohamsl, :float
      t.column :structure_type, :string, :limit=>6
    end
    add_index :fcc_structures, :unique_si
  end
```

```
  def self.down
    drop_table :fcc_locations
    drop_table :fcc_owners
    drop_table :fcc_structures
  end
end
```

■ **Note** We are adding indexes on the `unique_si` column on all the tables. We will use this column as both a primary key for each model and as a foreign key to tie all the models together. Therefore, we want our tables to have the indexes for faster select and join operations.

After you run the migration, create a new Ruby program (call it `import_fcc.rb`) in the same folder as the `.dat` files. Copy the code from Listing 5-3 into your newly created `import_fcc.rb` file, and execute it from the command line (`ruby import_fcc.rb`).

Listing 5-3. *FCC ASR Conversion to SQL Data Structures*

```
require 'parsedate'

# We'll call this three times, once for each .dat file
# Notice that the main loop yields execution to the specialized line
# parser which each invocation of this function passes in
def process_file (input_file, output_file)
  File.new(output_file, "w") do |out|
    puts "starting on #{input_file}"
    IO.foreach(input_file) do |line|
     out.puts yield(line)
    end # end iteration through (and implicity close) input file
  end # implicitly close the output file
  puts " . . done with #{input_file}, created #{output_file}"
end

# RA.dat contains fcc structures. The block passed to process_file
# just selects a subset of fields
  process_file('RA.dat','fcc_structures.dat')  do |line|
  line.split('|').values_at(4,23,24,25,26,27,28,29,30).join('|')
end

# EN.dat contains owner data. The block passed to process_file
# just selects a subset of fields
process_file('EN.dat','fcc_owners.dat')  do |line|
  line.split('|').values_at(4,7,14,16,17,18).join('|')
end
```

```
# CO.dat contains locations data. The block passed to process_file
# selects a subset of fields, and translates the 'S' and 'W' in the
# lat/lng fields into positive/negative float values
process_file('CO.dat','fcc_locations.dat')  do |line|
  unique_si, lat_deg, lat_min, lat_sec, lat_dir, long_deg, long_min,➥
 long_sec, long_dir = line.split('|').values_at(4,6,7,8,9,11,12,13,14)
  sign = (lat_dir == 'S') ? -1 : 1
  latitude = sign * (lat_deg.to_f + lat_min.to_f / 60 + lat_sec.to_f/3600)
  sign = (long_dir == 'W') ? -1 : 1
  longitude = sign * (long_deg.to_f + long_min.to_f / 60 + long_sec.to_f/3600)
  [unique_si, lat_deg, lat_min, lat_sec, lat_dir, latitude, long_deg,➥
 long_min, long_sec, long_dir, longitude].join('|')
end

puts "Complete"
```

Running this script should take 45 seconds or so, and it will print out status on the three files as it goes. The result of running this should be three new files: `fcc_owners.dat`, `fcc_structures.dat`, and `fcc_locations.dat`. That's good, but we're not done yet; recall that our objective is to parse the data into a format suitable for MySqlImport to utilize. Let's run MySqlImport now. From the command line, execute the following three commands:

```
mysqlimport --delete --fields-terminated-by='|' -columns=➥
unique_si,lat_deg,lat_min,lat_sec,lat_dir,latitude,long_deg,long_min,➥
long_sec,long_dir,longitude --user=root --local maps_development fcc_locations.dat

mysqlimport --delete --fields-terminated-by='|' --columns=unique_si,name,address,➥
city,state,zip --user=root --local maps_development fcc_owners.dat

mysqlimport --delete --fields-terminated-by='|' -columns=➥
unique_si,date_constructed,date_dismantled,address,city,state,height,➥
elevation,ohag,ohamsl,structure_type --user=root --local➥
 maps_development fcc_structures.dat
```

These three commands together should take about 90 seconds to run, even on a laptop MySQL installation. Together they insert more than 500,000 rows into your database. That solves the problems related to the size of the data set.

MySqlImport is an extraordinarily useful tool for importing large quantities of data. For more information, see the documentation at `http://dev.mysql.com/doc/refman/5.0/en/mysqlimport.html`.

Using Your New Database Schema

Now that you've got the data into the database, you need to set up some models to access the data. Go ahead and create empty files for three models: `app/models/fcc_location.rb`, `app.models/fcc_owner.rb`, and `app/models/fcc_structure.rb`. The content of each of these three files is shown in Listings 5-4, 5-5, and 5-6. Note that you do *not* need to use `script/generate` to create these models, as we already have the migrations in place. If you do decide

to use `script/generate`, make sure you don't get confused later by the extra migrations lying around in your `db/migrate` directory.

Listing 5-4 shows the content of the `app/models/fcc_location.rb` model.

Listing 5-4. *app/models/fcc_location.rb*

```
class FccLocation < ActiveRecord::Base
  set_primary_key 'unique_si'
  has_one :fcc_owner, :foreign_key=>'unique_si'
  has_one :fcc_structure, :foreign_key=>'unique_si'
end
```

Listing 5-5 shows the content of the `app/models/fcc_owner.rb`.

Listing 5-5. *app/models/fcc_owner.rb*

```
class FccOwner < ActiveRecord::Base
  set_primary_key 'unique_si'
  has_one :fcc_location, :foreign_key=>'unique_si'
  has_one :fcc_structure, :foreign_key=>'unique_si'
end
```

Listing 5-6 shows the content of the `app/models/fcc_structure.rb`.

Listing 5-6. *app/models/fcc_structure.rb*

```
class FccStructure < ActiveRecord::Base
  set_primary_key 'unique_si'
  has_one :fcc_location, :foreign_key=>'unique_si'
  has_one :fcc_owner, :foreign_key=>'unique_si'
end
```

The structure of these three models is very flat; there are simple one-to-one relationships among all three models. Looking at the code, you can see that each model includes two `has_one` relationships to each of the other two models.

To enable the link between models, you need to set the primary key column to `unique_si` using the `set_primary_key` call you see in each model. This simply tells Rails to utilize a column other than the default `id` column as the primary key. Why can't you use the default `id` column? Because when you import each row in the tables, you don't know what value will be in the auto-incrementing `id` column in the other tables. The value of the `unique_si` column, on the other hand, is readily available during the import process for each file.

If you look at this structure and conclude that it would be cleaner and simpler to put all the data in one table, you are absolutely right. As outlined earlier, we'll do a one-table/one-model implementation as well.

Mapping Structures in Hawaii

Let's put our data to use by displaying a manageable subset of the data: the FCC structures in Hawaii. You already have the models, so create the controller, view, and JavaScript file:

- *Controller*: Run `ruby script/generate controller ChapFive` from the command line to create the controller. Add the code in Listing 5-7 to the resulting `app/controllers/chap_five_controller.rb` file.
- *View*: Create a new file `app/views/chap_five/map.rhtml`, and add the code in Listing 5-8. Note that this code is very similar to the map views you created earlier, so you can copy and paste most of it.
- *JavaScript*: You can reuse your `application.js` file from Chapter 2. If you want to center and zoom the map on the Hawaiian islands, you can set `centerLatitude = 20.9`, `centerLongitude = -156.4`, and `startZoom = 7` in `application.js`.

Listing 5-7. *map Action to Add to app/controllers/chap_five_controller.rb*

```
def map
  structures=FccStructure.find_all_by_state 'HI', :include=>➥
[:fcc_owner,:fcc_location]
  @towers=Array.new
  structures.each do |s|
    @towers.push({:latitude=>s.fcc_location.latitude,
:longitude=>s.fcc_location.longitude, :name=>s.address})
  end
end
```

The first line in the `map` action finds structures in the state of Hawaii. The `:include` parameter tells ActiveRecord to load the other two models (`fcc_owner` and `fcc_location`) at the same time. The loop takes the resulting array of models and simplifies it into an array of hashes. The hashes have just three keys: `latitude`, `longitude`, and `name`. You extract these keys in preparation for outputting the `@towers` array in JSON, which you'll do in the view. You could output the entire ActiveRecord result set, but you would end up with much more information than you need and a bloated JSON data structure on your page.

Listing 5-8. *app/views/chap_five/map.rhtml*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"➥
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <script src="http://maps.google.com/maps?file=api&v=2&key=YOUR_ KEY_HERE"➥
type="text/javascript"></script>
  <%=javascript_include_tag 'application'%>
  <script type="text/javascript">
    var markers=<%=@towers.to_json%>;
  </script>
</head>
<body>
  <div id="map" style="width:800px;height:600px"></div>
</body>
</html>
```

The bold line in Listing 5-8 outputs the `@towers` array in JSON format and assigns it to the `markers` JavaScript variable. The JavaScript code in `application.js` (as mentioned previously, you can reuse your `application.js` from Chapter 2) is expecting the `markers` variable and uses it to plot markers on the map.

Navigate to `http://localhost:3000/chap_five/map` to see the results, which should look like Figure 5-2.

Figure 5-2. *The FCC structures in Hawaii*

Chapter 6 expands on this example and will help you build a better user interface for your map.

Assembling the Files in Memory

We just stepped through the process of importing the three FCC files into three different database tables and creating ActiveRecord associations to glue them together at query time. Another option is to assemble the three files into one and then import the resulting file into a single table. You might think that this would be a prohibitively memory-consuming process, but it's actually perfectly feasible with a typical developer workstation.

The process is straightforward: load two of the three files into an in-memory Hashtable, using their `unique_si` field as the *hash key*. The hash key points to the full contents of the line. Then iterate through the third file (there's no need to load it completely into memory like the others), and do a lookup on the `unique_si` in the two Hashtables you created from the other files. Finally, output a new, combined file line by line. Listing 5-9 has the code to do this.

Listing 5-9. *Assembling All Three Files in Memory*

```
puts "starting . . ."
# open first two files and read them into a large hash of lines
owners = Hash.new
```

```
IO.foreach('EN.dat') do |line|
  fields=line.split('|')
  id = fields[4]
  owners[id]=fields
end

puts "#{owners.size} owners"

structures = Hash.new
IO.foreach('RA.dat') do |line|
  fields=line.split('|')
  id = fields[4]
  structures[id]=fields
end

puts "#{structures.size} structures"

# open the output file
out = File.new('towers.dat', 'w')

# open the locations file, iterate through it,
# and correlate with hashes from other two files
IO.foreach('CO.dat') do |line|
  l_fields=line.split('|')
  id = l_fields[4]
  o_fields=owners[id]
  s_fields=structures[id]

  lat_deg, lat_min, lat_sec, lat_dir, long_deg, long_min, long_sec, long_dir➥
=l_fields.values_at(6,7,8,9,11,12,13,14)
  sign = (lat_dir == 'S') ? -1 : 1
  latitude = sign * (lat_deg.to_f + lat_min.to_f / 60 + lat_sec.to_f/3600)
  sign = (long_dir == 'W') ? -1 : 1
  longitude = sign * (long_deg.to_f + long_min.to_f / 60 + long_sec.to_f/3600)

  # print the combined fields to the output file
  # fields are: latitude,longitude, owner name, owner street address,
  # owner city, owner state, owner zip,
  # address, city, state, height, elevation, ohag, ohamsl, structure_type
  out.puts [id,latitude,longitude,o_fields.values_at(7,14,16,17,18),➥
s_fields.values_at(23,24,25,26,27,28,29,30)].flatten.join('|')

end

out.close
puts "Done"
```

Put this code in `assemble.rb`, and run it from the directory where the `.dat` files reside. If you monitor the memory consumption of the Ruby process while executing this code, you may see it take between 200MB and 350MB. Still, on a reasonably powerful developer's laptop (1.8GHz processor, 768MB RAM), the script completed in two to three minutes without any problems.

You know what needs to be done from here, right? You need to create and run a migration, import the data into the resulting table with MySqlImport, and create a model to interact with the new table. Listing 5-10 shows the migration.

Listing 5-10. *Migration for the Tower Model*

```
class Towers < ActiveRecord::Migration
  def self.up
    create_table :towers do |t|
      t.column :latitude, :float
      t.column :longitude, :float
      t.column :owner_name, :string, :limit=>200
      t.column :owner_address, :string, :limit=>35
      t.column :owner_city, :string, :limit=>20
      t.column :owner_state, :string, :limit=>2
      t.column :owner_zip, :string, :limit=>10
      t.column :address, :string, :limit=>80
      t.column :city, :string, :limit=>20
      t.column :state, :string, :limit=>2
      t.column :height, :float
      t.column :elevation, :float
      t.column :ohag, :float
      t.column :ohamsl, :float
      t.column :structure_type, :string, :limit=>6
    end

    add_index :towers, :state
    add_index :towers, :latitude
    add_index :towers, :longitude
  end

  def self.down
    drop_table :towers
  end
end
```

Listing 5-11 shows the MySqlImport command.

Listing 5-11. *MySqlImport Code*

```
mysqlimport --delete --fields-terminated-by="|" --user=root➥
--local maps_development towers.dat
```

Listing 5-12 shows the code for the new model.

Listing 5-12. *The Tower Model Code*

```
class Tower < ActiveRecord::Base
  def to_json
    self.attributes.to_json
  end
end
```

Finally, you need to replace the `map` action you created earlier with a simplified version. Listing 5-13 shows a new `map` action that utilizes the new `Tower` model.

Listing 5-13. *map Action Utilizing the Tower Model*

```
def map
  @towers=Tower.find_all_by_state 'HI'
end
```

Assembling the files in-memory seems to be the best approach for this data. There are fewer moving parts to the end solution, and the script to assemble the data isn't any more complicated than the script to parse the three files independently. We will assume for future examples that you have successfully executed the single-table technique. We will be using the `Tower` model (and the data you just loaded into it) in the next two chapters.

A PURE ACTIVERECORD IMPORT

Earlier in this chapter, we said we would present a pure ActiveRecord import. The code for this is shown here. Put it into a `.rake` file in your `lib/tasks` directory (we put ours in `lib/tasks/tower_import.rake`), and change `path/to/your/towers.dat` to point to your `towers.dat` file. Finally, run the task from the command line with `rake import_towers`.

```
task :import_towers => :environment do
  start = Time.now
  count=0
  IO.foreach("path/to/your/towers.dat") do |line|
    c = line.split('|')
    Tower.create(
      :latitude =>c[1],
      :longitude =>c[2],
      :owner_name =>c[3],
      :owner_address =>c[4],
      :owner_city =>c[5],
      :owner_state =>c[6],
      :owner_zip =>c[7],
      :address =>c[8],
      :city =>c[9],
      :state =>c[10],
      :height =>c[11],
      :elevation =>c[12],
```

```
      :ohag =>c[13],
      :ohamsl =>c[14],
      :structure_type =>c[15])

    count += 1
    puts "imported #{count} towers in #{Time.now - start} seconds"➥
if count % 200 == 0

  end
end
```

This import took more than an hour and a half on our development laptop, relative to just more than a minute for the MySqlImport with the same dataset.

A SQL View

There is one final way of utilizing the data that is midway between the two previous approaches. You can use a MySQL view and have ActiveRecord select directly from that. A *view* is a temporary table that is primarily—in this case, exclusively—used for *retrieving* data. A view is basically the cached result of a query like the one in Listing 5-5, without the state-specific data limitation. You can select from a view in the same way that you can select from an ordinary table, but the actual data is stored across many different tables. So in this case, we'll be referencing three different tables behind the scenes, but our application will only know about a single model.

There are several disadvantages to using a view. One is lower performance than the single-table solution we already implemented. And views can have subtle problems, especially when combined with Rails testing fixtures. Generally speaking, until Rails fully supports views, we recommend avoiding them unless there are no other options.

■ **Note** Using a SQL view in this way is possible only with MySQL 5.0.1 and later, PostgreSQL 7.1.x and later, and some commercial SQL databases. If you're using MySQL 3.x or 4.x and would like to use the new view feature, consider upgrading.

Listing 5-14 shows the MySQL 5.x statements needed to create the view.

Listing 5-14. *MySQL Statement to Create a View on the Three Tables*

```
CREATE VIEW towers AS
  SELECT
    fcc_locations.unique_si as id,
    fcc_locations.longitude as longitude,
    fcc_locations.latitude as latitude,
```

```
    fcc_owners.name as owner_name,
    fcc_owners.city as owner_city,
    fcc_owners.state as owner_state,
    fcc_owners.zip as owner_zip,
    fcc_structures.city as city,
    fcc_structures.state as state,
    fcc_structures.height as height,
    fcc_structures.height as height,
    fcc_structures.structure_type as structure_type
  FROM (fcc_locations inner join fcc_owners on fcc_locations.unique_si➥
= fcc_owners.unique_si)
      inner join fcc_structures on fcc_locations.unique_si➥
= fcc_structures.unique_si
```

So why is a view better than the multitable `SELECT`? Basically, it precomputes all of the correlations between the various tables and stores the answer for later use by *multiple future queries*. Therefore, when you need to select some chunk of information for use in your script, the correlation work has already been done, and the query executes much faster. However, you should realize that creating a view for a single-run script doesn't make much sense, since the value is realized in a time/computation savings *over time*.

■ **Tip** For more information on the creation of views in MySQL, visit `http://dev.mysql.com/doc/refman/5.0/en/create-view.html`. To see the limitations on using views, visit `http://dev.mysql.com/doc/refman/5.0/en/view-restrictions.html`. For more information on views in PostgreSQL, visit `http://www.postgresql.org/docs/8.1/static/sql-createview.html`.

Screen Scraping

Sometimes the data you want to use just isn't available in a nice, neat little package or service. In these cases, you can try searching the Web for the data you want, and you might find part or all of it on someone else's web site. If it's not available for download, as a web service, or for purchase, you might consider parsing the visible HTML and extracting the parts that you care about. This process is called *screen scraping*, because you are writing a program that pretends to be a normal, legitimate visitor but is really harvesting the data and usually storing it in your own database.

Accomplishing this is different for every single source of data, but we'll try to give you the basic tools you'll need to be successful. The basic idea is to download the pages in sequence, then use loops and HTML parsing to find and extract the interesting bits. Typically, you will store the extracted data in a database to avoid going back to the source of the information each time it's needed.

COPYRIGHT AND LEGAL ISSUES

There are legal and ethical concerns to consider when scraping, and neither the authors of this book nor Apress condone information or intellectual property theft or copyright infringement in any form. Please always ask for permission from site owners before scraping their sites. Sometimes owners would prefer to provide you with the data in a less bandwidth-intensive (and more convenient for you) way, or they may have terms and conditions for using their data (like reciprocal links or copyright attributions).

There are many legitimate reasons to use screen scraping to obtain data. Among other reasons, site owners may not have the resources or the skills to create a web service or an API for their data. Therefore, they might say you're welcome to take any data you want, but they can't help you get it into a more convenient format.

Regardless of the reason for scraping, you should always get written permission. Simply because the data is available without a fee on a web site does not mean that you are free to take it and republish it at your whim, *even if you do not charge any sort of fee*. Consult a lawyer if you can't get permission; otherwise, you might find that your hobby map turns into a crushing lawsuit against you.

Our Scraping Tool: scrAPI

We're going to use the excellent scrAPI library by Assaf Arkin to power our screen scraping. The advantage of scrAPI is that it lets you select and process elements on the page in a structured way, utilizing DOM and CSS constructs you're probably already familiar with: tag names, attributes, IDs and class names. Read more on scrAPI at its home on RubyForge at `http://rubyforge.org/projects/scrapi/`, or on the Cheat Sheet available at `http://labnotes.org/svn/public/ruby/scrapi/cheat/scrapi.html`.

scrAPI is not the only scraping library available. You may also want to look at Why's Hpricot (`http://code.whytheluckystiff.net/hpricot`) as an alternative.

To install scrAPI as a RubyGem, execute the following from a command line:

```
gem install scrapi
```

If you are asked if you want to install the `Tidy` dependency, install it as well.

■ **Caution** If you are running on Windows, you must make a change to the RubyGem installation for scrAPI to run correctly. You need to locate the directory for the `Tidy` library—the path will be similar to `C:\ruby\lib\ruby\gems\1.8\gems\scrapi-1.2.0\lib\tidy\`, depending on your system—and delete or rename the `libtidy.so` file in this directory.

A Scraping Example

As an example, you'll be taking a list of latitudes and longitudes for the capital cities of many countries in the world. The page that you'll scrape is located at `http://googlemapsbook.com/chapter5/scrape_me.html`. It's not the most challenging scraping example, but it will serve our purposes.

The first thing you need to do is look at the source of the page. The easiest way to do this is to view or save the source in your browser.

Now you need to do some analysis of the HTML of this page to decide what you can do with it. Listing 5-15 shows the important bits for our discussion.

Listing 5-15. *Snippets of HTML from the Sample Scraping Page*

```
(After about 10 lines of header HTML you'll find this...)
<!-- Content Body -->
<table border="1" width="100%">
<tr>
<td >Country</td>
<td >Capital City</td>
<td >Latitude</td>
<td >Longitude</td></tr>

<tr><td class="latlongtable">Afghanistan</td>
<td class="latlongtable">Kabul</td>
<td class="latlongtable">34.28N</td>
<td class="latlongtable">69.11E</td></tr>

<tr><td class="latlongtable">Albania</td>
<td class="latlongtable">Tirane</td>
<td class="latlongtable">41.18N</td>
<td class="latlongtable">19.49E</td></tr>

<tr><td class="latlongtable">Algeria</td>
<td class="latlongtable">Algiers</td>
<td class="latlongtable">36.42N</td>
<td class="latlongtable">003.08E</td></tr>

(and 190 countries later...)

<tr><td class="latlongtable">Zambia</td>
<td class="latlongtable">Lusaka</td>
<td class="latlongtable">15.28S</td>
<td class="latlongtable">28.16E</td></tr>

<tr><td class="latlongtable">Zimbabwe</td>
<td class="latlongtable">Harare</td>
<td class="latlongtable">17.43S</td>
<td class="latlongtable">31.02E</td>
</tr>
</table>
<!-- Content Body End -->
```

So how do you extract the information that you care about? This page is pretty straightforward, since all the data you want is within a single table on the page. You can direct scrAPI to process the table (by tag name, since there's only one table) and to iterate through each row in the table.

You do need to do a little string manipulation for this example outside of the simple scrape. Specifically, you need to determine the sign of the latitude and longitude measurements based on the N/S E/W labels. You can see the required code in Listing 5-16; put this in a stand-alone `scrape.rb` file and place it anywhere on your file system.

Listing 5-16. *Screen Scraping Example with scrAPI*

```
require 'scrapi'
require 'open-uri'

# retrieve html
url='http://googlemapsbook.com/chapter5/scrape_me.html'
html = open(url).read
puts "retrieved html from #{url}"

# helper function to scrape individual rows in the table
get_details= Scraper.define do
  process "td:nth-child(1)", :country => :text
  process "td:nth-child(2)", :capital => :text
  process "td:nth-child(3)", :lat => :text
  process "td:nth-child(4)", :lng => :text
  result :country, :capital, :lng, :lat
end

# defining the scraper
city_scaper = Scraper.define do
  array :cities
  process "table tr", :cities => get_details
  result :cities
end

cities = city_scaper.scrape(html)

cities.shift

cities.each do |c|
  lat = c.lat.to_f
  lat *= -1 if c.lat.match(/S/)
  lng = c.lng.to_f
  lng *= -1 if c.lat.match(/W/)
  puts "#{c.capital},#{c.country} : #{lat},#{lng}"
end

puts 'done'
```

Running this code will retrieve the HTML from `http://googlemapsbook.com/chapter5/scrape_me.html`, parse out the desired information, and print the results back to the console.

Saving the Results in Your Database

Of course, you need to store the data in the database to make it useful. To do that, you need three things: a new model, a new table, and a way for your script to access your Rails environment. Generate a `CapitalCity` model and migration with `ruby script/generate model CapitalCity`. You don't need to modify the resulting `app/models/capital_city.rb` file; just generating it is sufficient. Place the code in Listing 5-17 into the `db/migrate/005/create_capital_cities.rb` file.

Listing 5-17. *db/migrate/005_create_capital_cities.rb Migration*

```
class CreateCapitalCities < ActiveRecord::Migration
  def self.up
    create_table :capital_cities do |t|
      t.column :country, :string, :null=>false
      t.column :capital, :string, :null=>false
      t.column :lat, :float, :default=>0
      t.column :lng, :float, :default=>0
    end
  end

  def self.down
    drop_table :capital_cities
  end
end
```

Finally, you need to give your scraping script access to the Rails environment. An easy way to do this (and to keep your scripts centralized), is to create a new Rake task, as you did with the geocoding tasks in Chapter 4. Create a new file, `/lib/tasks/scraping_tasks.rake`. Listing 5-18 shows the contents of this file, with changes from the stand-alone scraping code in bold.

Listing 5-18. *scraping_tasks.rake*

```
require 'scrapi'
require 'open-uri'

task :scrape_capital_cities => :environment do
  # retrieve html
  url='http://googlemapsbook.com/chapter5/scrape_me.html'
  html = open(url).read
  puts "retrieved html from #{url}"
```

```
  # helper function to scrape individual rows in the table
  get_details= Scraper.define do
    process "td:nth-child(1)", :country => :text
    process "td:nth-child(2)", :capital => :text
    process "td:nth-child(3)", :lat => :text
    process "td:nth-child(4)", :lng => :text

    result :country, :capital, :lat, :lng
  end

  city_scraper = Scraper.define do
    array :cities
    process "table tr", :cities => get_details
    result :cities
  end

  cities= city_scraper.scrape(html)

  cities.each do |c|
    puts "saving city #{c.capital}"
    lat = c.lat.to_f
    lat *= -1 if c.lat.match(/S/)
    lng = c.lng.to_f
    lng *= -1 if c.lat.match(/W/)

    CapitalCity.create(:country=>c.country, :capital=>c.capital,➥
 :lat=>lat, :lng=>lng)
  end

  puts 'done'
end # end rake task
```

This Rake task can be executed by navigating to your application's base directory and typing `rake scrape_capital_cities` in the command line.

■ **Note** We hereby explicitly grant permission to any person who has purchased this book to use the information contained in the body table of `scrape_me.html` for any purpose (commercial or otherwise), provided it is in conjunction with a map built on the Google Maps API and conforms to Google's terms of service. We make no warranties about the accuracy of the information (in fact, there is one deliberate error) or its suitability for any purpose.

Screen Scraping Considerations

You need to consider a few things when doing screen scraping:

- If you intend to scrape a dynamic source on a schedule, or repeatedly over the course of time, you'll need to build in a lot of error checking. For example, our code would completely break if we made a change as simple as adding another table to the page. Fortunately, in real life, elements are usually rich with ID and class descriptors, and you should anchor your scraping activity to these whenever possible.
- Not all sources of data are going to be 100% accurate. For example, we've deliberately made a mistake for Ottawa, Canada, changing the sign from N to S, thereby flipping it below the equator. This causes our import script to treat the latitude as negative instead of positive. These kinds of mistakes are likely to happen with *any* data source you use, and in most cases, they will need to be corrected manually after the import.
- Sometimes the data is static or from a single source, and writing a program to do the work doesn't make sense. If the problem looks simple, you might try using your code editor's built-in search-and-replace functions. The search-and-replace approach would certainly be a viable alternative to the programmatic approach we took in Listing 5-10.

Summary

As you can see, there are a lot of ways to get the information you need to create a successful map. We encourage you to look at Appendix A, where we've collected a wide variety of different sources of information for common (and not so common) mapping applications. You'll find things such as political boundaries and the locations of airports, schools, and churches, as well as data on lakes and rivers.

In the next chapter, we'll continue with the example from Listing 5-5 and build a proper user interface. We'll show you how to do some fancy things with CSS and DOM manipulation. In Chapter 7, we'll round out this example with a thorough discussion of ways to handle such vast amounts of data on a map simultaneously and reminisce about the days when Google Maps API version 1 gave us a practical limit of 50 to 75 pins and a crash-the-browser limit of just a couple hundred. Progress is wonderful.

CHAPTER 6

Improving the User Interface

In this chapter, you'll use the FCC Antenna Structure Registration (ASR) data you collected in Chapter 5 and create a mashup that really shines. What kind of interface surrounds a helpful map? What tricks can you do with a little more CSS and JavaScript? What kinds of things besides markers can you put on a map to increase its usefulness? You'll find some suggestions in this chapter.

This chapter begins where the middle of Chapter 5 left off, but if you're starting here, it's easy to catch up. We will use a controller much like we have for the previous chapters, together with a `map` view to display the map. We will continue to use the same project we've used throughout the book.

In this chapter, you'll learn how to use CSS and JavaScript to enhance your maps as follows:

- Have your map adjust its size to fill any browser
- Add a toolbar that hovers over the map
- Create side panels for your map
- Allow users to selectively view or hide groups of data points

Create a controller for your work in this chapter by typing **ruby script/generate controller chap_six** on the command line. Create a new `map` action on this controller with the code in Listing 6-1.

Listing 6-1. *map Action in app/controllers/chap_six_controller.rb*

```
def map
  @towers=Tower.find :all, :conditions=>['state = ? AND latitude < ?➥
     AND longitude > ?  AND latitude > ? AND longitude < ?',➥
     'HI', 20.40, -156.34, 18.52, -154.67]
end
```

Recall that in Chapter 5, we created the `Tower` model to represent antenna structures in the FCC ASR data. The `map` action utilizes the `Tower` model to retrieve all the antenna structures on Hawaii's Big Island. The numbers in the query represent the bounding latitude and longitude; the first latitude/longitude pair represents the upper left (northwest) corner, and the second latitude/longitude pair represents the lower right (southeast) corner. Note that we're also narrowing the query based on `state` being equal to `'HI'`; this is to speed up the query. If the database can restrict the result set to Hawaii towers only, it has a much smaller set of rows on which to perform the numeric comparisons to get the final result set.

CSS: A Touch of Style

CSS is the modern method of choice for controlling the visual appearance of an XML document. We'll put our CSS in a separate file: `public/stylesheets/style.css`.

In your `map` view, you'll need to add a reference to an external style sheet, as shown in Listing 6-2. Since its appearance will momentarily be controlled by this CSS file, it's also possible to remove the explicit size from the `map div`.

Listing 6-2. *views/chap_six/map.rhtml*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"➥
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
     <script src="http://maps.google.com/maps?file=api&v=2&key=API_KEY"➥
type="text/javascript"></script>
     <%=javascript_include_tag 'prototype','application' %>
     <script type="text/javascript">
       var markers=<%=@towers.to_json%>;
     </script>
     <%=stylesheet_link_tag 'style'%>
</head>
<body id="body">
     <div id="map"></div>
</body>
</html>
```

Two other things are going on in this `map` view: we are including `prototype.js` via the `javascript_include_tag`, and we are outputting the markers in JSON format, just like we did in Chapter 5. We are including `prototype.js` because we are going to utilize some of its methods for the user interface in this chapter.

Without the `style` attribute, the `map div` collapses to nothing. Clearly, you need to actually *create* the style sheet and reapply the size declarations that were removed. Listing 6-3 shows the contents of the `public/stylesheets/style.css` file.

Listing 6-3. *public/stylesheets/style.css to Give the Map Dimensions*

```
#map {
    width: 500px;
    height: 400px;
}
```

■ **Tip** The # character in a style sheet means that you are defining an ID selector, so the `#map {...}` style will apply to the element with an ID of `map`, that is, `<div id="map"></div>`. On the other hand, if you want a style to apply to a *class* of elements, use the dot, which is the class selector. For example, `.my-class{...}` in your style sheet will apply to `<div class="my-class"></div>`. You can have multiple elements with the same class, and the style will apply to all of them. You'll spend most of your time with CSS writing either ID selectors, class selectors, or some combination thereof. To learn more about CSS, start with the tutorials at `http://www.w3schools.com/css/css_intro.asp`.

With the action, view, and CSS file in place, you just need an appropriate `application.js` to make sure your map initializes properly. The JavaScript code in Listing 6-4 should go in the usual `public/javascripts/application.js`. It contains the standard map initialization code you've seen before, utilizing the `markers` JavaScript variable to place points on the map.

Listing 6-4. *public/javascripts/application.js*

```
var map;
var centerLatitude = 19.6;
var centerLongitude = -155.5;
var startZoom = 9;
var markerHash={};
var currentFocus=false;

function addMarker(latitude, longitude, id) {
    var marker = new GMarker(new GLatLng(latitude, longitude));

    GEvent.addListener(marker, 'click',
        function() {
           focusPoint(id);
        }
    );

    map.addOverlay(marker);
    return marker;
}
```

```
function init() {
    map = new GMap($("map"));
    map.addControl(new GSmallMapControl());
    map.setCenter(new GLatLng(centerLatitude, centerLongitude), startZoom);
    for(i=0;i<markers.length; i++) {
      var current =markers[i];
      marker=addMarker(current.latitude, current.longitude,current.id);
      markerHash[current.id]={marker:marker,address:current.address,visible:true};
    }
}
window.onload=init;
```

In this code, note that we declare a global object called `markerHash` (the {} notation is equivalent to an empty object in JavaScript), and that we build up the object as we iterate through the `markers` array. We'll utilize this variable later, so just bear in mind when you see it next that we declared it here during initialization. Also, note that the `focusPoint` function is undefined at this point, which means that if you click a marker you will get a JavaScript error. We will define this function later in the section "Populating the Side Panel."

Going to `http://localhost:3000/chap_six/map`, you should see a map centered on Hawaii's Big Island, populated with 80 or 90 markers. Now, let's look at some CSS-based enhancements to the map.

Maximizing Your Map

A surprising number of Google Maps projects seem to use fixed-size maps. But why lock the users into particular dimensions when their screen may be significantly smaller or larger than yours? It's time to meet the map that fills up your browser, regardless of its screen size. Try swapping out your `style.css` file for Listing 6-5.

Listing 6-5. *Style.css for a Maximized Map*

```
html, body {
     margin: 0;
     padding: 0;
     height: 100%;
}

#map {
     position: absolute;
     top: 0;
     left: 0;
     width: 100%;
     height: 100%;
}
```

As you can see in Figure 6-1, the map is now completely flexible and fills any size of browser screen.

Figure 6-1. *Our map fills up the browser at 800 × 600.*

This method is particularly ideal for situations where a map is being used as part of a slide show or on a kiosk. However, it also works in the web page context, especially when combined with the trick described in the next section.

Tip Once you have the map maximized, you might notice how Internet Explorer 6 likes to show a disabled vertical scrollbar on our perfectly fitted page. Under most circumstances, this is actually desired behavior, since it means that centered sites are consistent with both short and long content. In our case, however, you really don't want it there. Fortunately, banishment is achieved with a pretty straightforward rule: `html { overflow: hidden; }`.

Adding Hovering Toolbars

CSS's `position` declaration opens up a world of options for styling your web pages. Using `position`, it's possible to layer multiple elements on top of one another, including text, images, and even scrolling Flash movies and scrolling `div` elements.

For the map, this means you can make content of various kinds hover on top of the map that the API generates. For comparison, Windows Live Local uses a full-screen map with translucent control widgets; check it out at `http://local.live.com/`.

Continuing the example from Listing 6-2, change your `map.rhtml` file to include some markup for a toolbar, as shown in Listing 6-6.

Listing 6-6. *map.rhtml with Added Markup for a Toolbar*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"➥
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
     <script src="http://maps.google.com/maps?file=api&v=2&key=API_KEY"➥
type="text/javascript"></script>
     <%=javascript_include_tag 'prototype','application' %>
     <script type="text/javascript">
       var markers=<%=@towers.to_json%>;
     </script>
     <%=stylesheet_link_tag 'style'%>
</head>
<body id="body">
     <div id="map"></div>
     <div id="toolbar">
          <h1>Cell-Tower Locations</h1>
          <ul id="options">
               <li><a href="#">Towers</a></li>
               <li><a href="#">Poles</a></li>
               <li><a href="#">Masts</a></li>
               <li><a href="#">Other</a></li>
          </ul>
     </div>
</body>
</html>
```

And now, some CSS magic to take that markup and pull the toolbar up on top of the map. Add the styles in Listing 6-7 to your `style.css` file.

Listing 6-7. *Styles for a Floating Toolbar*

```
#toolbar {
     position: absolute;
     top: 20px;
     left: 60px;
     width: 400px;
     padding: 5px;
     background: white;
     border: 1px solid black;
}
```

You can see in Figure 6-2 that we've added a few more styles to make the toolbar's menu and titles prettier, but they're not critical to the layout example here. The important thing to note is the `position: absolute` bit. A block-level element such as a `div` naturally expands to fill all of the width it has available, but once you position it as absolute or float it, it no longer exhibits that behavior. So, unless you want it shrink-wrapping its longest line of text, you'll need to specify a width as either a fixed amount or some percentage of the window width.

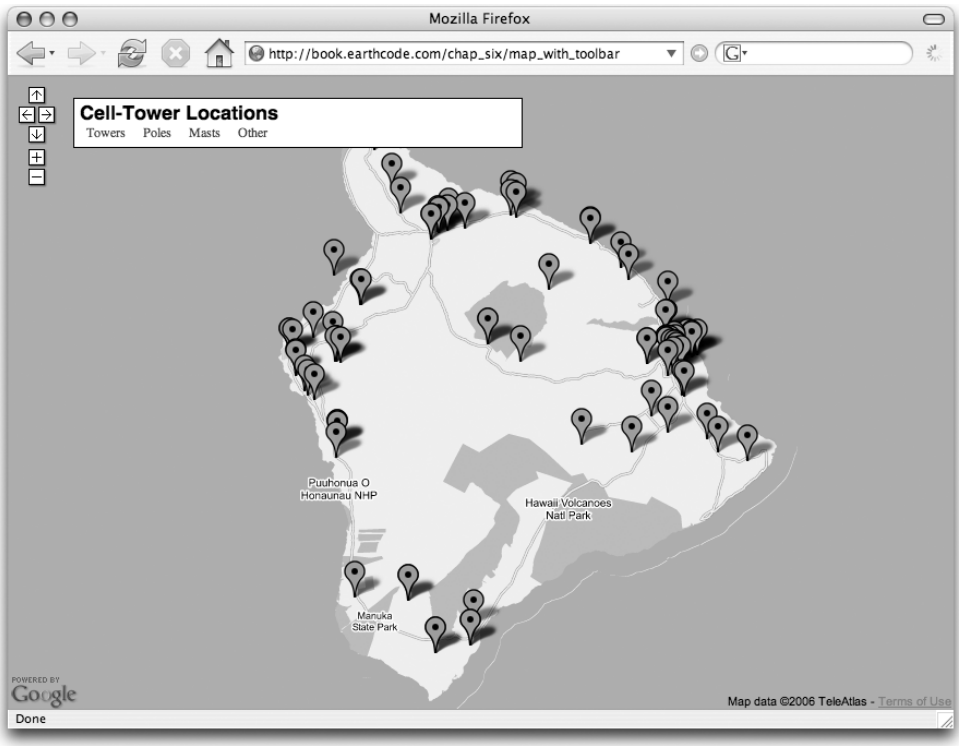


Figure 6-2. *Some styles for the toolbar*

WHAT ABOUT A FULL-WIDTH TOOLBAR?

Shouldn't it be possible to create a bar that's some fixed amount *less* than 100% of the available width? What about a floating toolbar that starts exactly 60 pixels from the left edge and then goes to exactly 40 pixels from the right edge?

It's possible in two different ways. You will see how to accomplish sizing maneuvers such as this using JavaScript. However, you can also create a full-width toolbar using just CSS. It's a little hairy, but there's certainly convenience (and possibly some pride, too) in keeping the solution all CSS.

The gist of the approach is that you need to "push in" the width of the absolutely positioned toolbar so that when it has a declared width of 100%, the 100% is 100% of the exact width you want it to have, rather than 100% of the browser's entire client area.

The toolbar `div` will need an extra wrapper around it, to do the "pushing in." So start by changing your markup:

```
<div id="toolbar-wrapper">
     <div id="toolbar">
     ...
     </div>
</div>
```

Now add the following styles to the `style.css` file:

```
#toolbar-wrapper {
     margin-right: 100px;
     position: relative;
}

#toolbar {
     width: 100%;
     ...
}
```

The right margin on the toolbar wrapper causes the toolbar itself to lose that horizontal space, even though the toolbar is ultimately being sucked out of the main document flow with `position: absolute`.

Creating Collapsible Side Panels

A common feature on many Google Maps mashups is some kind of side panel that provides supplementary information, such as a list of the pins being displayed. You can implement this simple feature in a number of ways. Here, we'll show you one that uses a little CSS and JavaScript to make a simple, collapsible panel.

First, the new side panel will need some markup. Modify the body section of Listing 6-2 to look like Listing 6-8.

Listing 6-8. *Index Body with Added Markup for a Side Panel*

```
<body id="body">
     <div id="map-wrapper">
          <div id="map"></div>
     </div>
```

```
    <div id="toolbar">
        ...
    </div>
    <div id="sidebar">
        <p>Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Proin
        accumsan condimentum dolor. Vestibulum ante fabicum...</p>
    </div>
</body>
```

Notice that we have added two new `div` elements to the markup: `map-wrapper` and `sidebar`. In our style sheet, we can reference these two elements with ID selectors: `#map-wrapper` and `#sidebar`. To style this, you use almost the same trick as for the floating toolbar. This time, that wrapper has a margin that pushes the `map` `div` out of the way, so the elements appear beside each other, rather than overlapping. Listing 6-9 shows the CSS to add to `style.css`.

Listing 6-9. *New Styles for the Side Panel*

```
#map-wrapper {
    position: relative;
    height: 100%;
}

#sidebar {
    position: absolute;
    top: 0;
    width: 300px;
    height: 100%;
    overflow: auto;
}

#map-wrapper { margin-right: 300px; }
#sidebar {
    right: 0px;
    display:block;
}

body.sidebar-off #sidebar { display: none; }
body.sidebar-off #map-wrapper { margin-right: 0px; }
```

If you fill up the side panel with some more content, you can see how the `overflow` declaration causes it to scroll. It behaves just like a 1997-era frame, but without all the hassle of broken Back buttons and negative frame stigma.

■ **Note** Listing 6-9 provides only the simplest styles for this side panel. You'll find when you try to apply a right-side padding to `#sidebar` that it pushes in not just the content but also the scrollbar, which is an undesirable effect. Fortunately, it's an easy fix: just nest a `sidebar-contents` `div` inside the main side panel, and then put your styles on that. Alternatively, you can use the CSS selector `#sidebar p` to give special margins to all paragraphs residing inside.

Notice that we've defined two sets of styles for `#sidebar` and `#map-wrapper` in our CSS. The first set of styles is the default, which will be used in the absence of any more-specific CSS rules. The second set of styles (in bold in the listing) apply when the `body` element in the document has the `sidebar-off` class applied to it. The name of the game here is to effect multiple changes in your document by applying a single class change to the document body. In this case, when the `sidebar-off` class is applied to the body, the CSS effects two visible changes: the `sidebar` element is no longer displayed (its `display` property is set to `none`); and the `map-wrapper` element expands to the right (by changing its `margin` from `300px` to `0px`).

The effect is demonstrated in Figure 6-3, where we use the Firefox DOM Inspector to change the `body` element's `class` attribute, and suddenly the side panel vanishes. This example is rather trivial, since there are only two changes taking place. However, imagine a scenario in which you want many subtle style changes, depending on whether the side panel is visible. Controlling these changes through CSS rules is an easy way to manage the interface, even as the interface becomes more complex.

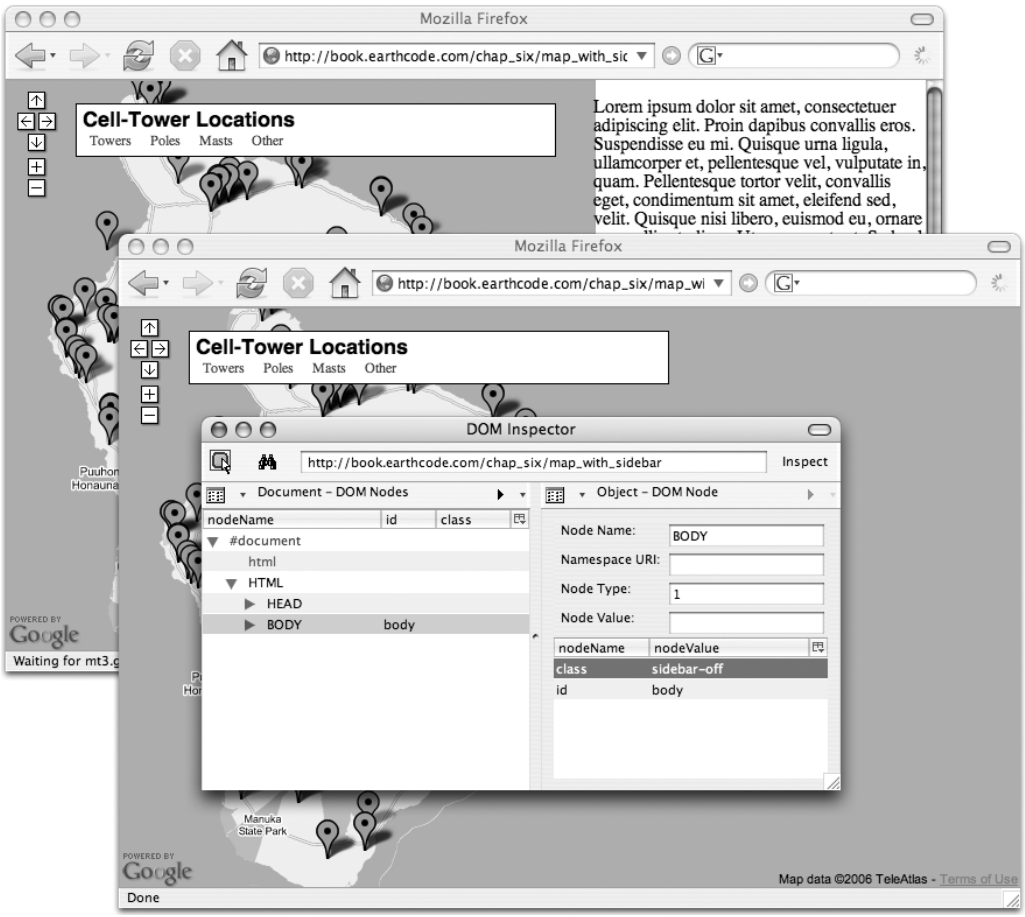


Figure 6-3. *The side panel obeys the body's class.*

Scripted Style

With the examples of the previous section in mind, we'll now examine a few ways to augment those CSS tricks with a little JavaScript.

Switching Up the Body Classes

You've seen that changing the body's `class` attribute can effect multiple changes in the interface. Now you just need to change the class through JavaScript triggered by a user event. To change the class, use Prototype's handy `addClassName` and `removeClassName` methods. You can embed the function calls in links in the toolbar, as shown in Listing 6-10.

Listing 6-10. *Side Panel Controls Added to map.rhtml*

```
<div id="toolbar">
    ...
  <ul id="sidebar-controls">
    <li><%=link_to_function 'hide',"Element.addClassName('body','sidebar-off')",➥
{:id=>'button-sidebar-hide'}%></li>
    <li><%=link_to_function 'show',"Element.removeClassName('body',➥
'sidebar-off')",{:id=>'button-sidebar-show'}%></li>
  </ul>
</div>
```

The `link_to_function` JavaScript helper takes a snippet of JavaScript code and attaches it to the `click` event of the link. In this case, the JavaScript snippet is a call to Prototype's `Element.addClassname` or `removeClassName` methods. The last argument to `link_to_function` is a hash of options for the HTML element the helper generates. We're just setting the ID of the resulting link here, but if you want to specify a `class` or a `style`, you would do so in this hash as well.

One important note: the first argument to `Element.addClassName` and `Element.removeClassName` is the ID of the element to change. It does *not* take the tag name, as our example may appear to communicate. Our code works because we have given the body tag `id=body` (refer back to Listing 6-2 to see where the ID is set).

ALTERNATIVES FOR ATTACHING EVENTS

In our toolbar, we use the built-in Rails `link_to_function` JavaScript helper to attach events to the show and hide links. There are alternative ways to attach events, including the following:

- Attach them directly in your JavaScript file: `$('button-sidebar-hide').onclick = function() {Element.addClassName('body','sidebar-off')};`.
- Utilize RJS (Rails JavaScript) templates and callbacks to move more of the logic server-side, thereby minimizing the amount of JavaScript in your application.
- Utilize the Unobtrusive JavaScript plug-in for Rails (`http://www.ujs4rails.com/`) to retain the convenience of the `link_to_function` helpers, but also to keep the JavaScript separate from the document markup.
- Utilize alternative JavaScript libraries (such as Behavior or jQuery) that are well-suited for attaching events unobtrusively.

The way we're doing it here (using the inline `link_to_element` JavaScript helper) is probably the most popular approach for Rails programmers, by virtue of it being the Rails default. The downside to this approach is that it muddies the resulting HTML with inline JavaScript code and it scatters your JavaScript event handlers between your JavaScript file and your `.rhtml` file(s). In many environments other than Rails, those would be considered significant downsides.

The approach you use will probably depend on the richness of your application's JavaScript-based functionality. Our experience has been that map-based applications are necessarily very rich in JavaScript code, simply because the Google Maps API itself is JavaScript. Once you have a large part of your application in JavaScript (the map handling code), it tends to change the center of gravity of your application and you use the inline JavaScript helpers and RJS less than you would otherwise. Your experience—and preferences—may vary, which is one of the reasons we're demonstrating the `link_to_function` approach in this example.

Finally, you can add some styles to spruce up the buttons a little. Using CSS, it's trivial to hide (or otherwise restyle) whichever button corresponds to the mode you're already in:

```
body.sidebar-off #button-sidebar-show { display: block; }
body.sidebar-off #button-sidebar-hide { display: none; }
```

Using these styles makes the two buttons appear to be the same one, as you can see in Figure 6-4.

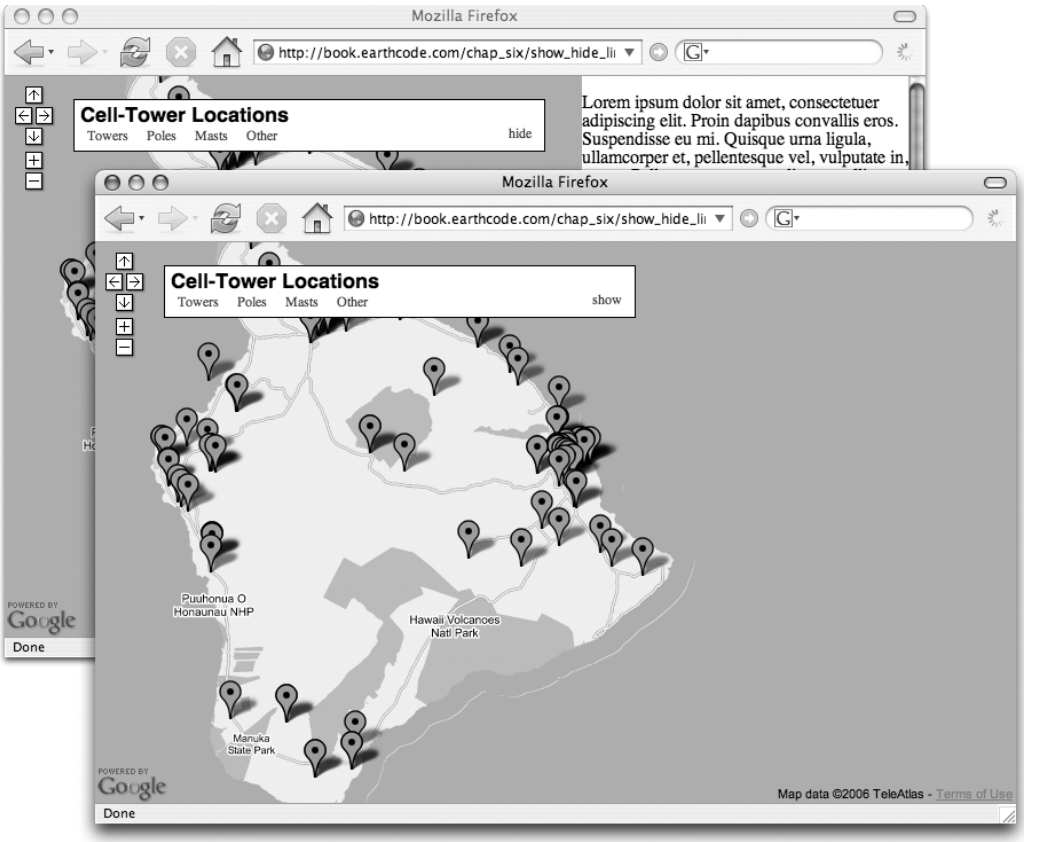


Figure 6-4. *The show/hide buttons behave as one, toggling the visibility of each other and of the side panel.*

Resizing with the Power of JavaScript

As you saw earlier, CSS gives you a significant amount of control over a page's horizontal layout. However, control over the vertical spacing is lacking. With only a style sheet, you can make a map the entire height of the window, or some percentage of that height, but you *cannot* make it be "from here to the bottom," or "100% minus 90 pixels." With JavaScript, however, this is very much possible.

JavaScript is an event-driven programming language. You don't need to be checking for things to happen all the time; you simply "hook" functionality onto various events triggered by the web browser.

With that in mind, all of the examples so far have already made use of the event `window.onload` to initialize the API and plot points on its map. What you're going to do next is hook some resizing functionality onto the event `window.onresize`. This code will execute when the window changes shape and resize the map to fit it.

Unfortunately, as is very obvious in the `windowHeight()` function of Listing 6-9, it has taken browser makers a long time to agree on how to expose the height of the client area to JavaScript. The method we've used here is the product of some exceptional research by Peter-Paul Koch (see `http://www.quirksmode.org/viewport/compatibility.html`). Incidentally, it's almost identical to the one Google itself uses to control the height of the Google Maps site's main map and side panel.

Pull up your `application.js` file and add the code shown in Listing 6-11 to it.

Listing 6-11. *Filling Vertical Space with the onresize Event*

```
function windowHeight() {
    // Standard browsers (Mozilla, Safari, etc.)
    if (self.innerHeight)
        return self.innerHeight;
    // IE 6
    if (document.documentElement && document.documentElement.clientHeight)
        return y = document.documentElement.clientHeight;
    // IE 5
    if (document.body)
        return document.body.clientHeight;
    // Just in case.
    return 0;
}

function handleResize() {
    var height = windowHeight();
    height -= document.getElementById('toolbar').offsetHeight - 30;
    document.getElementById('map').style.height = height + 'px';
    document.getElementById('sidebar').style.height = height + 'px';
}

function init() {
    ...

    handleResize();
}

window.onresize = handleResize;
```

The `handleResize()`function itself is actually pretty straightforward. The `offsetHeight` and `offsetWidth` properties are provided by the browser, and return—in pixels—the dimensions of their element, including any padding. Finding the correct height for the map and side panel

is simply a matter of subtracting that from the overall client window height, and then also removing the 30 pixels of padding that appear in three 10-pixel gaps between the top, the toolbar, the content area, and the bottom.

■ **Note** It's awkward to be individually assigning heights to the map and side panel. It would be cleaner if we could just assign the calculated height to a single wrapper and then set the children to each be permanently `height: 100%`. Indeed, such an approach works splendidly with Firefox. Unfortunately, Internet Explorer isn't able to get it quite right, so we're forced to use the slightly less optimal method of Listing 6-10.

Back in Listing 6-6, we placed the toolbar markup *after* the `map div` itself. This was partly arbitrary and partly because it's a convention to put the layers that are closer to the user later in the document. Now, however, the layering is to be removed in favor of a tiled approach, closer to what the Google Maps site itself uses. It's natural, then, to move the toolbar markup to before the map.

Also, we've added that `content` wrapper around the map and side panel. This is technically superfluous, but having a bit of extra markup to work with really helps to keep the style sheet sane. It's nearly always better to add wrappers to your template than to fill your CSS with ugly browser-specific hacks. (Some might disagree with us on this, but remember that wrappers are futureproof, while hacks can break with each new browser release.)

You can view the complete CSS changes that accompany Listing 6-12 on this book's web site, but it's not a dramatic departure from the styles of Listing 6-7. The changes are mostly aesthetic, now that the `handleResize()` method lets us do things such as put a nice 10-pixel margin between the key elements.

Listing 6-12. *Index Body with Markup Changes for Paneled Layout*

```
<body>
   <div id="toolbar">
          ...
   </div>
   <div id="content">
      <div id="map-wrapper">
        <div id="map"></div>
      </div>
      <div id="sidebar">
          ...
      </div>
   </div>
</body>
```

You can see how this example looks in Figure 6-5.

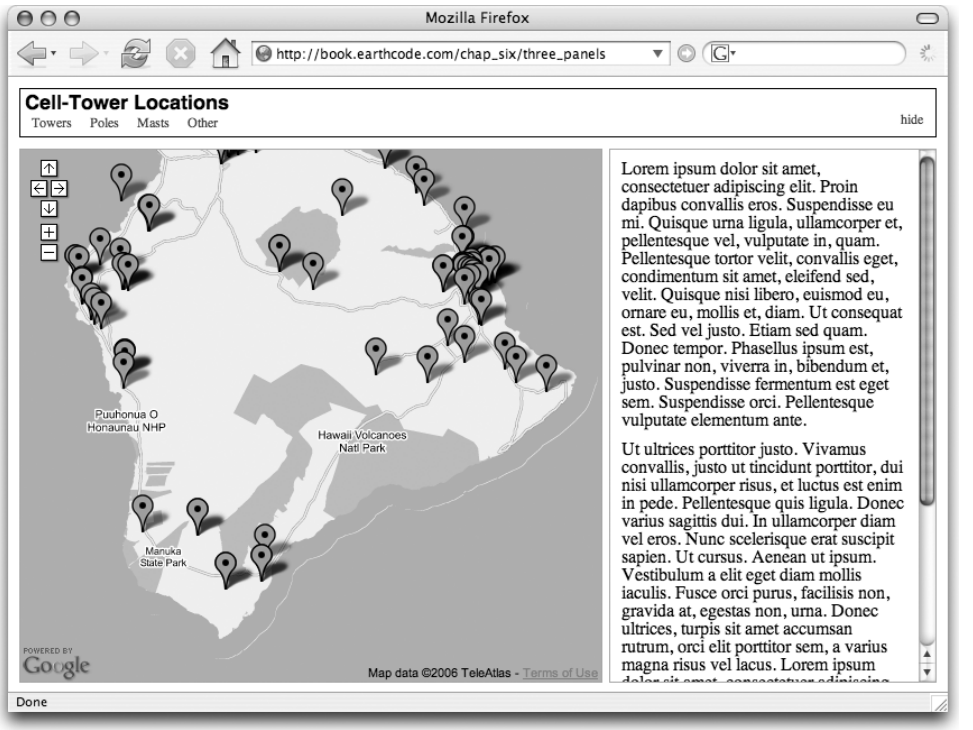


Figure 6-5. *The map area is divided into three elegant panels, one of which is collapsible.*

Populating the Side Panel

With our new side panel up and running, it would be good to get some actual content in there. A typical side panel use would be to present a list of all the markers plotted. This is particularly helpful when the markers are distributed in clusters. For example, a user could zoom in on an urban area to view a number of points bunched together, but she would be made aware that points exist elsewhere because that additional display has them listed.

To display the list of markers, you just need to edit the sidebar section of the `map.rhtml`, as shown in Listing 6-13.

Listing 6-13. *Generating the Side Panel in map.rhtml*

```
<div id="content">
    ...
  <div id="sidebar">
    <ul id="sidebar-list">
      <%@towers.each do |tower|%>
        <li id="sidebar-item-<%=tower.id%>">
          <%=link_to_function "<strong>#{tower.address}</strong> #{tower.city},➥
             #{tower.state} (#{tower.height}m", "focusPoint(#{tower.id})"%>
        </li>
      <%end%>
```

```
    </ul>
  </div>
</div>
```

Notice that our `link_to_function` is expecting a JavaScript function called `focusPoint`. Let's add that function to `application.js` now, as shown in Listing 6-14.

Listing 6-14. *The focusPoint Function in public/javascripts/application.js*

```
function focusPoint(id){
  markerHash[id].marker.openInfoWindowHtml(markerHash[id].address);
}
```

The function is just one line of code, but it's fairly dense. Recall that in the `init` function (which we defined back in Listing 6-4), we assembled an object called `markerHash`. Now, we're using `markerHash` to look up the current marker by ID, and calling the `openInfoWindowHtml` method on the marker we find. The `address` attribute (which we pass to `openInfoWindowHtml` for display) is also kept in the `markerHash` object for precisely this purpose. The key is that the `markerHash` contains both things we need: the `Marker` object itself, and the address to display in the info window.

The result of this will look similar to Figure 6-6. Note that we've enhanced the side panel with a few extra styles.

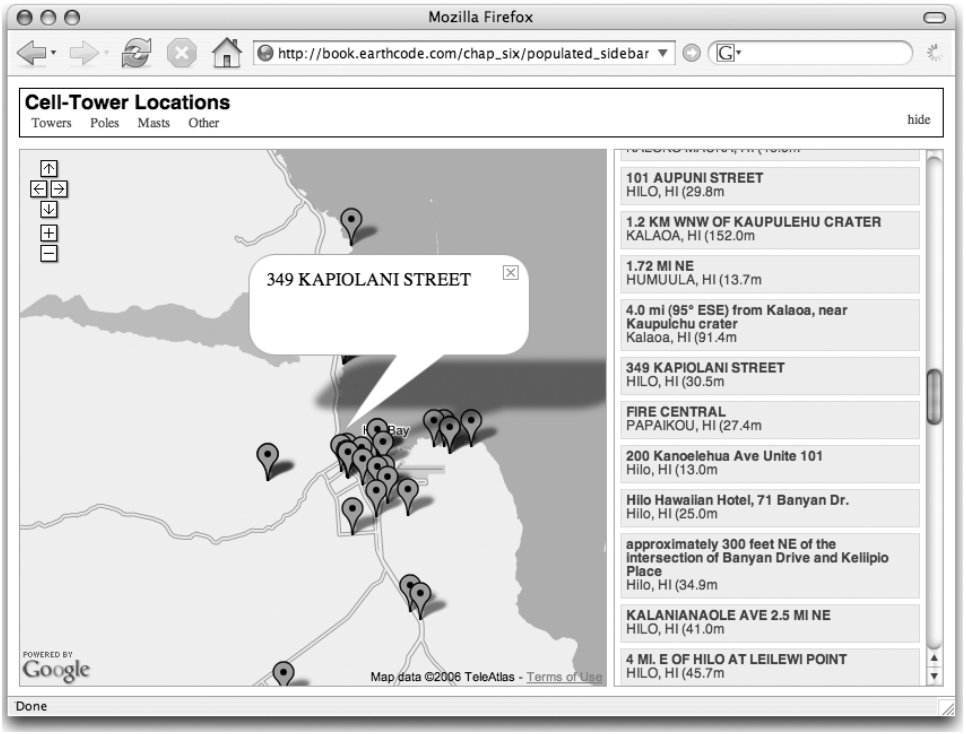


Figure 6-6. *The side panel populated with marker details*

Getting Side Panel Feedback

So far, users can interact with both the side panel item *and* the marker itself. However, they're receiving feedback through only the map marker—its info window pops up. It would be ideal if we could enhance this behavior by also highlighting the current point in the side panel list. Let's revisit the `focusPoint` function to provide this functionality. The code in Listing 6-15 should replace your current `focusPoint` function.

Listing 6-15. *An Enhanced focusPoint Function*

```
function focusPoint(id){
  if (currentFocus) {
    Element.removeClassName("sidebar-item-"+currentFocus,"current");
  }
  Element.addClassName("sidebar-item-"+id,"current");
  markerHash[id].marker.openInfoWindowHtml(markerHash[id].address);
  currentFocus=id;
}
```

The `currentFocus` global variable is the key to this problem. The first time the `focusPoint` function is called, `currentFocus` is `false` (see Listing 6-4 near the top to see where this variable is declared). On subsequent clicks, `currentFocus` is set to the ID of the previously clicked item. At that point, the `if` statement evaluates to `true`, and the previously focused item is defocused before applying focus to the newly clicked item.

Visually, *focus* means applying a class called `current` to a list item in the side panel. But first, we have to *remove* the `current` class from the last item selected. The `if` statement filters out the very first click when `currentFocus` isn't set yet.

And once again, with a few styles thrown in, you can see the results in Figure 6-7. The visual treatment in the side panel is subtle in this screenshot, but you can see that the border is stronger on the selected side panel item.

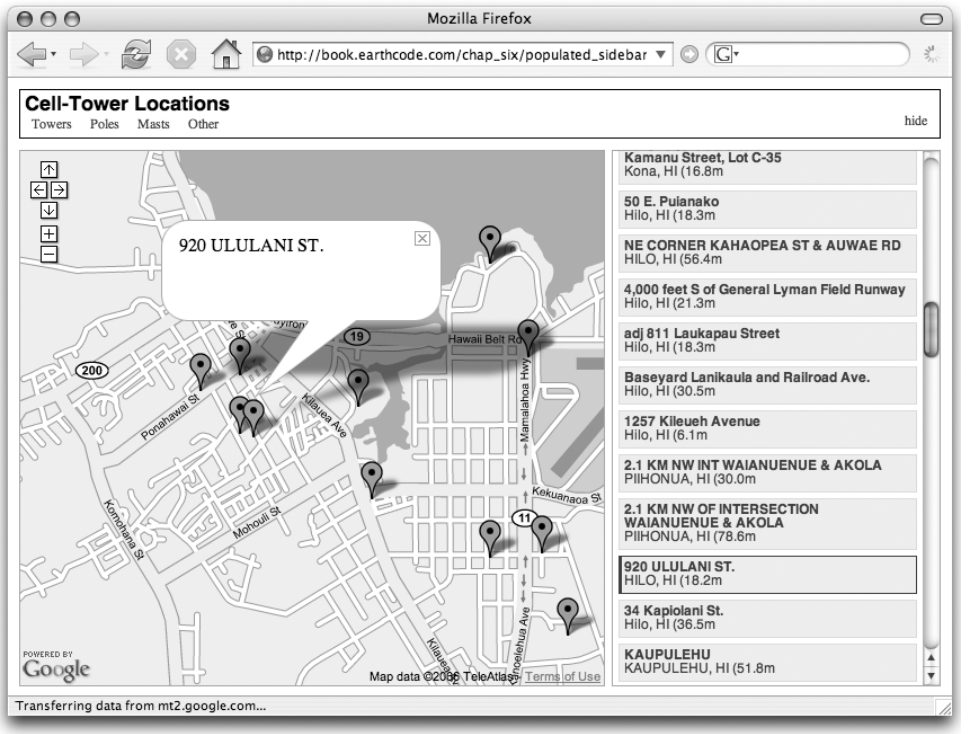


Figure 6-7. *The selected item in the side panel is highlighted.*

Data Point Filtering

One more area of the application still shows dummy content: the links underneath Cell-Tower Locations in the top toolbar. Right now these links are static. Let's make them reflect the actual structure types in the database and filter the map markers and side panel list when the links are clicked.

We're going to do the filtering browser-side, completely in JavaScript. Depending upon the application, this may or may not be the optimal approach. As an alternative, you may want to implement an Ajax callback to retrieve new side panel HTML and a new JSON data structure for the markers.

We are not making an Ajax call here; we're just using JavaScript to selectively limit what is displayed. When the entire data set for Hawaii is less than 40KB, what would be the point of breaking it up into multiple server calls? When you grab it in one big lump, it makes for a more seamless user interface since there's no waiting around for network latency on a 5KB file.

To provide the filtering mechanism, the first step is to identify the unique structure types in our dataset. To accomplish this, add the bold line of code in Listing 6-16 to your `app/controllers/chap_six_controller.rb` file.

Listing 6-16. *map Action in app/controllers/chap_six_controller.rb*

```
def map
  @towers=Tower.find :all, :conditions=>['state = ? AND latitude < ?➥
     AND longitude > ?  AND latitude > ? AND longitude < ?',➥
     'HI', 20.40, -156.34, 18.52, -154.67]

  @types=@towers.collect{|tower| tower.structure_type}.uniq
end
```

This line of code iterates through the `@towers` array, using the array's `collect` method to extract the `structure_type` from each item. Calling `uniq` on the result yields an array of unique structure types. We'll use this to render the links in the toolbar. Listing 6-17 shows the code in `app/views/chap_six/map.rhtml`.

Listing 6-17. *Showing @types in app/views/chap_six/map.rhtml*

```
<div id="toolbar">
  <h1>Cell-Tower Locations</h1>
  <ul id="filters">
    <li><%=link_to_function 'All', "filter('All')"%></li>
    <%@types.each do |type|%>
      <li><%=link_to_function type, "filter('#{type}')"%></li>
    <%end%>
  </ul>
 ...
```

In addition to iterating through the `@types` array and outputting links to the `filter` function, we are also outputting an All link, which will display all the structures in the dataset.

From here, we just need to implement the `filter()` function in JavaScript. Listing 6-18 shows the code to add to `public/javascripts/application.js`.

Listing 6-18. *The* `filter()` *Function in public/javascripts/application.js*

```
function filter(type){
  for(i=0;i<markers.length;i++) {
    var current=markers[i];
    if (current.structure_type == type || 'All' == type) {
```

```
        if (!markerHash[current.id].visible) {
          Element.show("sidebar-item-"+markers[i].id);
          map.addOverlay(markerHash[current.id].marker);
          markerHash[current.id].visible=true;
        }
      } else {
        if (markerHash[current.id].visible) {
          Element.hide("sidebar-item-"+markers[i].id);
          map.removeOverlay(markerHash[current.id].marker);
          markerHash[current.id].visible=false;
        }
      }
    }
}
```

Let's analyze what's going on in this function:

- The `for` loop iterates through all of the items in the `markers` array. Recall that the `markers` array was outputted as JSON in our view (see Listing 6-2).
- Inside the loop, there are just two possibilities: either we show the current marker or we hide the current marker. If the passed type is `all` we really don't want to discriminate; we'll show each marker regardless of its type.
- The `markerHash` object plays an important role, regardless of whether we are hiding or showing the current marker. Recall that `markerHash` was declared as a global variable and populated in the `init` method. If you look back at the population of `markerHash` (in Listing 6-4), you'll see that each entry contains a property called `visible`. At initialization time, each `visible` property was set to `true`. Once the loop determines whether the current marker should be shown, it uses the `visible` property to determine whether the marker is already displayed or not; after all, you don't want to add a marker to the map if it is already there.
- There are two steps to showing a marker. First, the list item in the side panel needs to be revealed, for which we use Prototype's `Element.show()` method. `Element.show()` takes a DOM element ID as a parameter, for which we use the marker's ID. Second, the marker itself needs to be added to the map, which we accomplish through the standard `map.addOverlay` method.
- The logic for removing markers is similar to the logic for showing markers but reversed.

The final result, with filtering, is shown in Figure 6-8.

Figure 6-8. *Marker filters in action*

You now have a reasonably full-featured map interface. You have an expandable full-page map, a collapsible side panel, side panel items that work in concert with the markers on the map, and, now, marker type filtering. You can use these techniques as a model for many different kinds of Google Maps mashups and applications.

RJS and Draggable Toolbars

One of the great things about Ruby on Rails is its flexibility. There are multiple ways of doing almost anything you need to do, and the functionality we've implemented in this chapter is no exception. Before we wrap up, let's look at two additional points to help you approach user interface challenges you may encounter.

RJS Templates and Partials

RJS is the JavaScript equivalent of RHTML: template files that are evaluated server-side through the Ruby interpreter before being served up to the browser. A common way of implementing the collapsible side panel and filtering is to utilize a `link_to_remote` callback to an RJS template, which would then effect the desired changes in the user interface. Is RJS a good choice for our application?

RJS would certainly be a reasonable choice. For this application, however, there would be a downside. RJS calls always involve a remote request (via XMLHTTP) back to the server, and in our application we can avoid the network latency by implementing the functionality directly in JavaScript.

The longer answer is that in our experience the amount of JavaScript coding necessarily involved with a Google Maps–based application changes the application's center of gravity, and you end up coding more in JavaScript. This is especially true as you make more sophisticated user interfaces, such as the JavaScript map resizing function we implemented in this chapter.

Other factors could change the equation. For example, if the filtering mechanism were more complex—perhaps a full-text search operation on marker names or data—it might make more sense to perform the filtering operation server-side and expose the interface updates in an RJS template.

Draggable Toolbars

Could you make the toolbar and/or side panel draggable? You certainly could—and with a minimal amount of effort—using the bundled Scriptaculous library. First, include all the default Rails JavaScript files with `<%=javascript_include_tag :defaults %>`. Next, include the following line in your view: `<%= draggable_element(:toolbar, :revert=>false) %>`. We aren't focusing on Scriptaculous here, but there are a lot of good examples available at `http://script.aculo.us/`.

Though it's relatively easy to make an element draggable, we caution you against doing it just because you can. It pays to ask whether "draggability" will actually enhance the user's experience. And if you do decide to implement it, think through the changes you will need to make to fully integrate the functionality. For example, if the top toolbar is draggable, what will happen to the space it occupies when you drag it away? Will the map and sidebar snap upward and expand vertically to take up the space? And what will happen when the toolbar is returned to the top? Will the map snap back down to give it space? Within what tolerance will the snap-back happen?

Remember also that draggable elements' positions don't automatically persist across page views. Which means that a toolbar dragged someplace will revert to its original place if the user refreshes the page. If that behavior confuses your users, you would probably have to implement some sort of persistence mechanism for the position, perhaps a behind-the-scenes Ajax call to store the new position in the session.

This isn't to discourage you from experimenting with features such as drag-and-drop, only to get you to think through the full effects on the user experience.

Summary

In this chapter, we took a look at a number of JavaScript and CSS techniques for making your maps better-looking and more functional. We made the map adjust its size to fill the browser window, and we added a toolbar that hovers over the map. We created a side panel for the map, and implemented JavaScript to show and hide the side panel. Finally, we created a JavaScript filtering mechanism to allow users to selectively hide and show groups of markers on the map.

In Chapter 7, you'll continue to develop this code, focusing on how to deal with the vastness of the full U.S.-wide FCC ASR database.

CHAPTER 7

Optimizing and Scaling for Large Data Sets

So far in the book, we've looked at the basics of the Google Maps API and shown how it's possible to retrieve and store data for your map. You've probably come up with some great ideas for your own map applications and started to assemble the information for your markers. And you may have found that your data set is overwhelmingly large—far larger than the simple examples you've been experimenting with so far.

In the previous chapters, you've been experimenting with the FCC data in the Antenna Structure Registration (ASR) database. As you've probably noticed, the FCC tower information is a rather large data set, containing more than 115,000 points across the United States. If you tried to map the towers using one `GMarker` per point, the map, or even the user's computer, would simply crawl to a halt.

When your data grows from a dozen to a few thousand points, or even hundreds of thousands of points, you need to select the best way to present your information without confusing or frustrating your users. This chapter presents a variety of methods for working with larger data sets such as the FCC tower data. The methods you'll learn will provide your users with an interactive experience while maintaining a sensible overhead in your web application.

When dealing with large data sets, you need to focus on three areas of your application: the communication between the server and the browser, the server side, and the client side. In this chapter, you'll learn techniques for each of these areas as follows:

- Streamline the data flowing between your server and client's web browser
- Optimize your server-side Rails actions and data storage
- Improve the users' experience with the client-side JavaScript and web browser

Understanding the Limitations

Before we discuss how to overcome any limitations that arise from dealing with large data sets, you should probably familiarize yourself with what those limitations are. When we refer to the *limits of the API*, we don't mean to imply that Google is somehow disabling features of the map and preventing you from doing something. What we're referring to are the *ambiguous* limits that apply to any web-based software, such as the software's ability to run in the client's web browser.

If you're developing your map application on a cluster of supercomputers, the limitations of your computer are going to be different than those of someone who is browsing on an old 486 laptop with just a few megabytes of RAM. You'll never know for sure what type of computer your users are going to have, so remember that not everyone is going to experience a map in the same way. For this chapter, we'll focus on the limitations related to plotting larger-than-normal data sets on an average home computer. These issues are mainly performance-related and occur when there are too many `GOverlay` objects on the map at one time.

Overlays are objects that build on the API's `GOverlay` class and include any items added to the map using the `GMap2.addOverlay()` method. In the Google Maps API, Google uses overlays for `GMarker` objects, `GPolyline` objects, and info windows, all of which you've probably been playing with a lot as you've progressed through this book. In each case, the overlay is built into the JavaScript class and in some cases may include shadows or translucent images. Along with the API overlays, the map may also contain custom overlays that you've built yourself. You can implement your own overlays, using the API's `GOverlay` object to display all sorts of information. In fact, one of the methods you'll explore in this chapter uses a custom overlay to display detailed information using a transparent GIF.

Here is a summary of the relevant limits:

`GMarker` *limits*: If you're going to display only markers on your map, the *maximum* number to try for the average user is around 100; however, performance will be slow on anything but the latest computer hardware. Loading markers and moving them around with JavaScript is an expensive operation, so for better performance and reliability, try to keep the number to around 50 to 75 `GMarker` objects on the map at one time—even fewer if you're combining them with `GPolyline` objects.

`GPolyline` *limits*: Too many `GPolyline` objects will slow the map in the same way as do too many markers. The difference with polylines is in the number of points in the lines, not the number of lines. One really long line with a bunch of points will slow the map down just as much as a few little lines. Load a maximum of 100 to 150 points, but keep in mind that using around 50 to 75 will make your application run a lot smoother. If your application requires a large, complicated set of polygons with hundreds of points, check out the server-side overlay and tile solutions described in this chapter. The examples demonstrate generating your own overlays and tiles, but the embedded images don't need to be limited to just markers—you could draw complicated images, lines, and shapes as well.

Info window limits: As you saw in Chapter 3, there's only one instance of an info window on the map at any given time, so there are no direct limits on the info window with regard to performance. However, remember that the info window adds more complexity to the map, so if you try to slide the map around while the window is open, the map may begin to slow down.

Streamlining Server-Client Communications

Throughout the book, we've mentioned that providing an interactive experience to your users is a key characteristic of your mapping application's success. Adding interactivity often means creating more requests back and forth between the client's web browser and the server. More requests mean more traffic and, accordingly, a slower response, unless you invest in additional resources such as hardware to handle the load. To avoid making these investments, yet still improve response time, you should always streamline any process or data that you'll be using to communicate with the client.

As you've seen, Ajax doesn't need to talk in XML. You can send and receive any information you want, including both HTML and JavaScript code. Initially, many web developers make the mistake of bloating their server responses with full and often verbose JavaScript. Bloating the response with JavaScript is easy on you as a developer but becomes a burden on both the server and the client. For example, the response from the server could add ten markers to your map by sending the following:

```
map.addOverlay(new GMarker(new GLatLng(39.49,-75.07)));
map.addOverlay(new GMarker(new GLatLng(39.49,-76.24)));
map.addOverlay(new GMarker(new GLatLng(39.64,-74.29)));
map.addOverlay(new GMarker(new GLatLng(40.76,-73.00)));
map.addOverlay(new GMarker(new GLatLng(40.83,-74.47)));
map.addOverlay(new GMarker(new GLatLng(40.83,-74.05)));
map.addOverlay(new GMarker(new GLatLng(40.83,-72.60)));
map.addOverlay(new GMarker(new GLatLng(40.83,-76.64)));
map.addOverlay(new GMarker(new GLatLng(41.17,-71.56)));
map.addOverlay(new GMarker(new GLatLng(41.26,-70.06)));
```

The problem with sending all this code in your response becomes apparent as your data set scales to larger and larger requests. The only unique information for each point is the latitude and longitude, so that's all you really need to send. The response would be better trimmed and rewritten using the JSON objects, such as the following, introduced in Chapter 2:

```
var points = {
    {lat:39.49,lng:-75.07},
    {lat:39.49,lng:-76.24},
    {lat:39.64,lng:-74.29},
    {lat:40.76,lng:-73.00},
    {lat:40.83,lng:-74.47},
    {lat:40.83,lng:-74.05},
    {lat:40.83,lng:-72.60},
    {lat:40.83,lng:-76.64},
    {lat:41.17,lng:-71.56},
    {lat:41.26,lng:-70.06},
}
```

By sending only what's necessary, you decrease every line from about 55 characters to just 23, an overall reduction of 32 characters per line and a savings of about 9KB for a single request with 300 locations. Trimming your response and generating the markers from the data in the response will also give your client-side JavaScript much more control over what to do with the response. If you're sending a larger data set of 1,000 points, you can easily see how you could save megabytes in bandwidth and download time. If your application is popular and receives a lot of traffic, small savings can add up over time.

Reducing data bloat is a fairly easy concept and requires little, if any, extra work. Though you may shrug it off as obvious, remember to think about it the next time you build your web application. Less bloat will make your application run faster. Plus, it will also make your code much easier to maintain as JavaScript operations will be in one place rather than spread around in the server response.

Optimizing Server-Side Processing

When building a map with a large and complex set of data, you'll most likely be interacting with the server to retrieve only a small subset of the available information. The trick, as you will soon see, is in how you request the information combined with how it's processed and displayed. You could retrieve everything from the server and then display everything in your client's web browser, but as we mentioned earlier, the client will slow to a crawl, and in many cases just quit. To avoid slowing the map and annoying your users, it's important to optimize the method of your requests.

To easily search, filter, and categorize the information displayed on the map, make sure your database has the appropriate data types for each of the fields in your database table. For example, if you have a `lat` and a `lng` column, make sure they're floats with the appropriate precision for your data. Using the proper data types will allow the database to better optimize the storage and retrieval of your information, making it a lot quicker to process each request. Also be sure to use indexing on frequently requested columns or other database-specific optimizations on your data.

Once your database is flush with information, your requests and queries will most likely be retrieving information about points within a particular latitude and longitude boundary. You'll also need to consider how much information you *want* to display vs. how much information is actually *possible* to display. After you've decided on an appropriate balance of wants vs. needs, you'll need to pick the solution that best fits your data. Here, we'll explore five possible solutions:

- Server-side boundary method
- Server-side common-point method
- Server-side clustering
- Custom detail overlay method
- Custom tile method

These approaches have varying degrees of effectiveness, depending on your database of information and the context of the map. We'll describe each method and then point out its advantages and disadvantages.

Server-Side Boundary Method

The boundary method involves requesting only the points within a specific boundary, defined using some relevant reference such as the viewport of the visible map. The success of the boundary method relies on highly dispersed data at a given zoom level.

If you have a large data set and the information is relatively dispersed over the globe, you can use the `GLatLngBounds` of the `GMap2` object as a boundary for your query. This essentially restricts the data in your response to those points that are within the onscreen viewable area of the map. For globally dispersed data at zoom level 1, where the map covers the entire globe, you'll see the whole world at once, so plotting the data set using markers is still going to go beyond the suggested 100 marker limit and cause problems, as shown in Figure 7-1.

At closer zoom levels, say 5 or higher, you'll have a smaller portion of the markers on the map at one time, and this method will work great, as shown in Figure 7-2. The same would apply for localized data dispersed across a smaller area or large, less dispersed data, but you'll need to zoom in much closer to have success.

Figure 7-1. *Server-side boundary method with the entire world at zoom level 1*

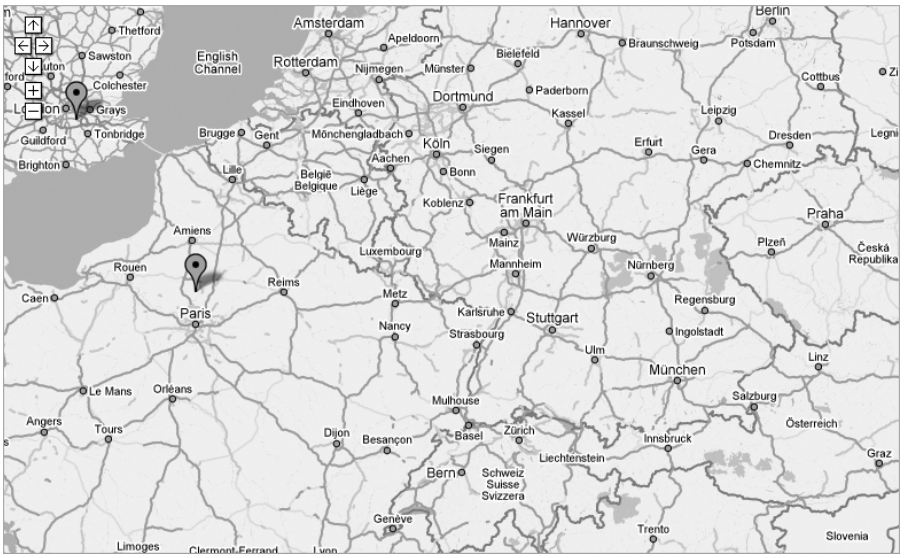


Figure 7-2. *Server-side boundary method at a closer zoom level*

To experiment with a smaller, globally dispersed data set, suppose you want to create a map of capital cities around the world. There are 192 countries, so that would mean 192 markers to display. Capital cities are an appropriate data set for the boundary method because there are relatively few points and they are dispersed throughout the globe. If you adjust the zoom of the map to something around 5, you'll have only a small portion of those points on the map at the same time.

■ **Tip** The boundary method is usually used in combination with one of the other solutions. You'll notice that in many of the server-based methods, the first SQL query still uses the boundary method to initially limit the data set to a particular area, and then additional optimizations are performed.

Listings 7-1 and 7-2 contain a working example of the server-side boundary method (`http://book.earthcode.com/chap_seven/server_bounds`) using the SQL database of capital city locations you created in Chapter 5 (in the screen scraping example). If you haven't created the database from Chapter 5, you can quickly do so using the Chapter 7 `capital_cities_seed.sql` file in the supplemental code for the book.

Listing 7-1. *public/javascripts/application.js (Replaces Existing File)*

```
var map;
var centerLatitude = 49.224773;
var centerLongitude = -122.991943;
var startZoom = 4;

function init() {
    map = new GMap2(document.getElementById("map"));
    map.addControl(new GSmallMapControl());
    map.setCenter(new GLatLng(centerLatitude, centerLongitude), startZoom);

    updateMarkers();

    GEvent.addListener(map,'zoomend',function() {
        updateMarkers();
    });

    GEvent.addListener(map,'moveend',function() {
        updateMarkers();
    });
}

function updateMarkers() {
    //remove the existing points
    map.clearOverlays();
```

```
    //create the boundary for the data
    var bounds = map.getBounds();
    var southWest = bounds.getSouthWest();
    var northEast = bounds.getNorthEast();
    var url = '/chap_seven/cities_within_bounds?ne=' + northEast.toUrlValue() +➥
        '&sw=' + southWest.toUrlValue();

    //log the URL for testing
    GLog.writeUrl(url);

    //retrieve the points using Ajax
    var request = GXmlHttp.create();
    request.open('GET', url, true);
    request.onreadystatechange = function() {
         if (request.readyState == 4) {
              var data = request.responseText;
              var points = eval('('+data+')');

              //create each point from the list
              for (var i=0; i < points.length; i++) {
                var point = new GLatLng(points[i].lat,points[i].lng);
                var marker = createMarker(point,points[i].capital);
                map.addOverlay(marker);
              }
         }
    }
    request.send(null);
}

function createMarker(point, html) {
    var marker = new GMarker(point);
    GEvent.addListener(marker, 'click', function() {
      var markerHTML = html;
      marker.openInfoWindowHtml(markerHTML);
    });
    return marker;
}

window.onload = init;
```

Listing 7-2. *cities_within_bounds Action (Add to app/controllers/chap_seven_controller.rb)*

```
  def cities_within_bounds
    ne = params[:ne].split(',').collect{|e|e.to_f}
    sw = params[:sw].split(',').collect{|e|e.to_f}

    # if the NE longitude is less than the SW longitude,
    # it means we are split over the meridian.
```

```
    if ne[1] > sw[1]
      conditions = 'lng > ? AND lng < ? AND lat <= ? AND lat >= ?'
    else
      conditions = '(lng >= ? OR lng < ?) AND lat <= ? AND lat >= ?'
    end

    cities=CapitalCity.find :all,
      :conditions => [conditions,sw[1],ne[1],ne[0],sw[0]]

    render :text=>cities.collect{|c|c.attributes}.to_json
  end
```

To get this example working in your environment, take the following steps:

1. Create a controller for this chapter. From the command line, execute: `ruby script/generate controller ChapSeven`.
2. Replace your `public/application.js` file with the JavaScript code in Listing 7-1.
3. Add the `cities_within_bounds` action in Listing 7-2 to your `app/controllers/chap_seven_controller.rb` controller file.
4. Create a new file, `app/views/chap_seven/map.rhtml`, and place the code from Listing 7-3 into it. This creates the map view we'll be using throughout this chapter. Don't forget to insert your own Google Maps API key in to the `map.rhtml` file.

Listing 7-3. *app/views/chap_seven/map.rhtml (New File)*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"➥
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <script src="http://maps.google.com/maps?file=api&amp;v=2&amp;key=YOUR_KEY"➥
type="text/javascript"></script>
  <%=javascript_include_tag 'application'%>
</head>
<body>
  <div id="map" style="width:100%;height:700px"></div>
</body>
</html>
```

The server-side boundary approach has two key parts. The first is the request to the server in Listing 7-1, which includes the bounds of the map—the southwest and northeast corners—as `url` parameters in the Ajax call:

```
//create the boundary for the data
var bounds = map.getBounds();
var southWest = bounds.getSouthWest();
var northEast = bounds.getNorthEast();
var url = '/chap_seven/cities_within_bounds?ne=' +
      northEast.toUrlValue() +
      '&sw=' + southWest.toUrlValue();
```

The second is the `cities_within_bounds` action in Listing 7-2, which limits the points to the boundary defined by the southwest and northeast corners:

```
if ne[1] > sw[1]
  conditions = 'lng > ? AND lng < ? AND lat <= ? AND lat >= ?'
else
  conditions = '(lng >= ? OR lng < ?) AND lat <= ? AND lat >= ?'
end

cities=CapitalCity.find :all,
  :conditions => [conditions,sw[1],ne[1],ne[0],sw[0]]
```

■ **Caution** You may have noticed that the query conditions are wrapped in an `if` statement. This is due to the meridian in the Mercator projection of the map. The map is displayed using a Mercator projection where the meridian of the earth is at the left and right edges. When you slide to the left or right, the map will wrap as you move past the meridian at +/– 180 degrees. In that case, the bounds are partially split across the left and right edges of the map, and the northeast corner is actually positioned at a point that is greater than 180 degrees. The Google Maps API automatically adjusts the longitude values to fit between –180 and + 180 degrees, so you need to request two portions of the map from your database covering the left and right sides.

When you move the map around or change the zoom level, a new request is created by the `moveend` and `zoomend` events in Listing 7-1. The request to the server retrieves a new JSON object, which is then processed by the JavaScript to create the necessary markers.

As you would expect, there are both pros and cons to using the boundary method. The advantages are as follows:

- This technique uses the standard existing Google Maps API methods to create the markers on the map.
- It doesn't drastically change your code from the simple examples presented earlier in the book.
- The server-side code imposes very little overhead.

The following are the boundary method's disadvantages:

- It works for only dispersed data or higher zoom levels.
- It may not work for lower zoom levels, as too many markers will be shown at once.
- The client's web browser makes a new request for markers after each map movement, which could increase server traffic.

Server-Side Common-Point Method

Unlike the server-side boundary method, the server-side common-point method relies on a *known point*—one around which you can centralize your data—and retrieves the maximum number of points relative to that known point. This method is useful for location-based

applications where you are asking your users to search for things relative to other things, or possibly even relative to themselves. It works for any zoom level and any data set, whether it's a few hundred points or thousands of points, but larger data sets may require more time to process the relative distance to each point.

For example, suppose you want to create a map of all the FCC towers relative to someone's position so he can determine which towers are within range of his location. Simply browsing the map using the server-side boundary method won't be useful because the data is fairly dense and you would need to maintain a very close zoom. What you really want is to find towers relative to the person's street address or geographic location. You could have him enter an address on your map, and then you could create the central point by geocoding the address using the methods you learned in Chapter 4.

If you choose to use this method, also be aware that user interface problems may arise if you don't design your interface correctly. The problem may not be obvious at first, but what happens when you slide the map away from the common central point? Using strictly this method means no additional markers are shown outside those closest to the common point. Your users could be dragging the map around looking for the other markers that they know are there but aren't shown due to the restrictions of the central point location, as you can see in Figure 7-3.

Figure 7-3. *A map missing the available data outside the viewable area*

Some maps we've seen use "closest to the center" of the map to filter points. This imposes the same ambiguity, as the map actually contains much more information, but it's simply ignored. When using the server-side common-point method, be sure to indicate to the users that the information on the map is filtered relative to the known point. That way, they are aware they must perform an additional search to retrieve more information.

Listings 7-4 and 7-5 show a working example of the common-point method (`http://book.earthcode.com/chap_seven/server_closest`). To provide a simpler example, we've made the map clickable. The latitude and longitude of the clicked point is sent back to the server as the known point. Then, using the FCC tower database, the map will plot the closest 20 towers to the click. You could easily modify the example to send an address in the request and use a server-side geocoding application to encode the address into latitude and longitude coordinates, or you could use the API's `GClientGeocoder` object to geocode an address.

Listing 7-4. *public/javascripts/application.js (Replaces Existing File)*

```
var map;
var centerLatitude = 42;
var centerLongitude = -72;
var startZoom = 10;

function init() {
    map = new GMap2(document.getElementById("map"));
    map.addControl(new GSmallMapControl());
    map.setCenter(new GLatLng(centerLatitude, centerLongitude), startZoom);

    //pass in an initial point for the center
    updateMarkers(new GLatLng(centerLatitude, centerLongitude));

    GEvent.addListener(map,'click',function(overlay,point) {
        //pass in the point for the center
        updateMarkers(point);
    });

}

function updateMarkers(point) {

    //remove the existing points
    map.clearOverlays();

    //create the boundary for the data to provide
    //initial filtering
    var bounds = map.getBounds();
    var southWest = bounds.getSouthWest();
    var northEast = bounds.getNorthEast();
    var url = '/chap_seven/nearby_towers?ne=' + northEast.toUrlValue() +➥
      '&sw=' + southWest.toUrlValue()+'&ll='+point.toUrlValue();

    //log the URL for testing
    GLog.writeUrl(url);

    //retrieve the points using Ajax
    var request = GXmlHttp.create();
    request.open('GET', url, true);
```

```
    request.onreadystatechange = function() {
         if (request.readyState == 4) {
              var data = request.responseText;
              var points = eval('('+data+')');

              //create each point from the list
              for (var i=0; i < points.length; i++) {
                var point = new GLatLng(points[i].latitude,points[i].longitude);
                  var marker = createMarker(point);
                  map.addOverlay(marker);
              }
         }
    }
    request.send(null);
}

function createMarker(point) {
    var marker = new GMarker(point);
    return marker;
}
window.onload = init;
```

Listing 7-5. *nearby_towers Action (Add to app/controllers/chap_seven_controller.rb)*

```
  def nearby_towers
    lat,lng=params[:ll].split(',').collect{|e|e.to_f}
    ne = params[:ne].split(',').collect{|e|e.to_f}
    sw = params[:sw].split(',').collect{|e|e.to_f}

    #convert to radians
    lat_radians=(lat / 180) * Math::PI
    lng_radians=(lng / 180) * Math::PI
    distance_sql=<<-SQL_END
    (acos(cos(#{lat_radians})*cos(#{lng_radians})*cos(radians(latitude))*➥
cos(radians(longitude)) + cos(#{lat_radians})*sin(#{lng_radians})*➥
cos(radians(latitude))*sin(radians(longitude))+➥
sin(#{lat_radians})*sin(radians(latitude))) * 3963)
    SQL_END

    towers = Tower.find(:all,
          :select=>"latitude, longitude, #{distance_sql} as distance",
          :conditions=>['longitude > ? AND longitude < ? AND latitude <= ? AND➥
                      latitude >= ?',sw[1],ne[1],ne[0],sw[0]],
          :order => 'distance asc',
          :limit => 20)

    render :text=>towers.collect{|c|c.attributes}.to_json
  end
```

Let's step through the `nearby_towers` action:

1. *Get the* `url` *parameters and ensure they are numeric*: The `nearby_towers` action is called with three `url` parameters: `ll` (short for *latitude/longitude*), which represents a known point; `ne`, which represents the northeast corner of the map; and `sw`, which represents the southwest corner of the map. All three parameters are comma-delimited latitude/longitude pairs. The code parses them to purely numeric values and places the values into local variables.

2. *Formulate a SQL clause for distance from the given point*: This is a key part of the code. The variable `distance_sql` is assigned a SQL string, which calculates the distance to the known point during the query. The `distance_sql` variable is used in the `:select` option to get an additional column representing distance in miles.

■ **Note** If you want the distance values to be calculated in kilometers instead of miles, replace the number 3,963 with 6,378.8. The "magic numbers" 3,963 and 6,378.8 represent the earth's radius in miles and kilometers, respectively.

---

3. *Find towers ordered by distance*: There are a couple of things to note here. First, we are using the `:select` option to get a subset of columns from the `towers` table. We are also putting `distance_sql` in the select clause and naming it *distance*. Secondly, we are using the `:conditions` option to restrict the query to towers within the visible bounds of the map. This isn't strictly necessary from a functional standpoint, but it will help the database process the query more quickly. Finally, the `:order` and `:limit` options together ensure that we will get only 20 towers and that those 20 towers will be the closest ones to the given point.

4. *Render the result to JSON*: The `nearby_towers` action is called via JavaScript, which is expecting a JSON data structure in return.

Once you have the files in place, browse to `http://localhost:3000/chap_seven/map` to see the results. If you want to see the raw JSON output by the `nearby_towers` action, point your browser to `http://localhost:3000/chap_seven/nearby_towers?ne=42.356514,-71.135101&sw=41.642131,-72.866821&ll=42,-72`.

The server-side common-point method offers the following advantages:

- It works at any zoom level.
- It works for any sized data, provided you add additional filtering.
- This method is great for relative location-based searches.

Its disadvantages are as follows:

- Each request must be calculated and can't be easily cached.
- Not all available data points appear on the map.
- It requires a relative location.
- Database queries may be slow for very large and dense data sets.

Server-Side Clustering

The server-side clustering solution involves using the server to further analyze your requests, and it works well for high-density data sets. In this case, the server analyzes the locations you've requested along with their proximity and then *clusters* markers to provide the maximum amount of information from the fewest number of markers. A cluster is just a normal `GMarker`, but it represents more than one marker within a close distance and therefore usually has a different icon.

If your data has a very high density and markers are often overlapping, you can reduce the number of markers on the map simply by combining near markers into one single cluster marker. When you zoom the map for a closer look, the cluster marker will expand into several individual markers, or more cluster markers, until the zoom is close enough that no clusters are needed. For data sets of around 1,000 points, clustering can be accomplished through JavaScript on the client side, which we'll discuss in the "Client-Side Clustering" section later in this chapter. Here, you'll see how to cluster data on the server side when you have hundreds of thousands of points.

To initially filter your data for the request, you can use either the server-side boundary method or the server-side closest-to-common-point method. For this example, we've chosen to request all the points within the viewable area of the map (the boundary method), and then we've applied clustering to the remaining points, as shown in Figure 7-4.

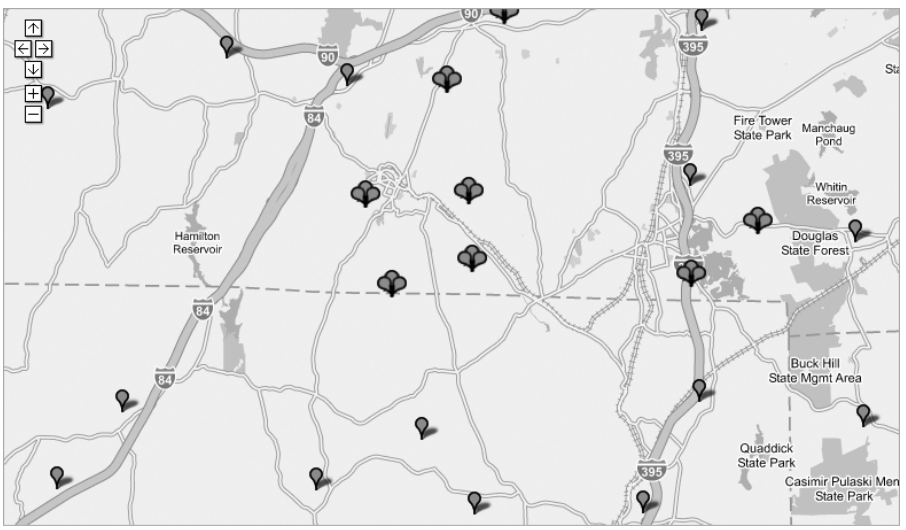


Figure 7-4. *A map with clustered and single points*

Combining clustering with either of the previous two methods can overcome some of their limitations. The drawback of the server-side boundary method is its limitation to a "closer to earth" zoom level. Zooming out means that there are too many points to display at one time on the map. By clustering the points, you can zoom out and still view the map within the marker limit, but some markers will be combined until you zoom in closer.

To cluster data into common groups, you need to determine which points lay relatively close to each other and then figure out how much clustering to apply to achieve the correct number of points. There are a variety of ways you can go about this, some simple and others much more complex. For the example here, we've chosen a simple method that we like to call the "*grid*" method.

To cluster using a grid, you take the outer boundary of the data set (for example, the *viewport*), divide the area into equally sized grid cells, and then allocate each of your points to a cell. The size of the grid cells will determine how detailed the map data is. If you use a grid cell that is 100 pixels wide, all markers within the 100-by-100 block will be combined into one marker. Listing 7-6 uses an incremental grid size starting with one-thirtieth of the height and width of the map:

```
lng_span+=(ne[1]-sw[1])/30
lat_span+=(ne[0]-sw[0])/30
```

which increases if the total is still too large at the end of the loop:

```
break unless clustered.length > max_markers
```

By incrementing the size of the cell, you can achieve the best resolution of data for the number of points available. Figure 7-5 shows an example map with grid cells and map areas outlined.

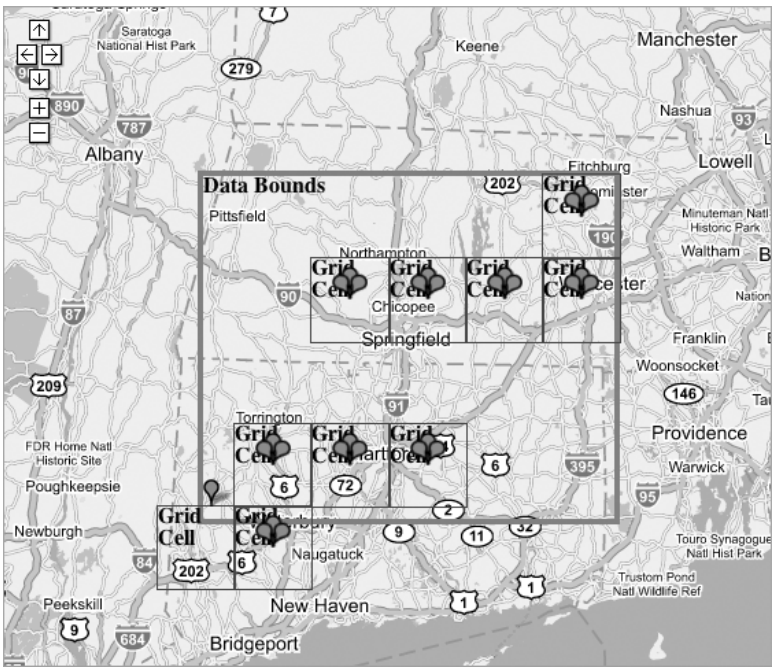


Figure 7-5. *A map showing the marked grid cells used for clustering*

Listings 7-6 and 7-7 (http://book.earthcode.com/chap_seven/server_cluster) are the JavaScript code and Rails action for the server-side clustering technique. Listing 7-6 should replace your existing public/javascripts/application.js, and Listing 7-7 should be added to your app/controllers/chap_seven.rb file.

Listing 7-6. *public/javascripts/application.js for Server-Side Clustering (Replaces Existing File)*

```
var map;
var centerLatitude = 42;
var centerLongitude = -72;
var startZoom = 10;

//create an icon for the clusters
var iconCluster = new GIcon();
iconCluster.image = "http://googlemapsbook.com/chapter7/icons/cluster.png";
iconCluster.shadow = "http://googlemapsbook.com/chapter7/icons/cluster_shadow.png";
iconCluster.iconSize = new GSize(26, 25);
iconCluster.shadowSize = new GSize(22, 20);
iconCluster.iconAnchor = new GPoint(13, 25);
iconCluster.infoWindowAnchor = new GPoint(13, 1);
iconCluster.infoShadowAnchor = new GPoint(26, 13);

//create an icon for the pins
var iconSingle = new GIcon();
iconSingle.image = "http://googlemapsbook.com/chapter7/icons/single.png";
iconSingle.shadow = "http://googlemapsbook.com/chapter7/icons/single_shadow.png";
iconSingle.iconSize = new GSize(12, 20);
iconSingle.shadowSize = new GSize(22, 20);
iconSingle.iconAnchor = new GPoint(6, 20);
iconSingle.infoWindowAnchor = new GPoint(6, 1);
iconSingle.infoShadowAnchor = new GPoint(13, 13);


function init() {
    map = new GMap2(document.getElementById("map"));
    map.addControl(new GSmallMapControl());
    map.setCenter(new GLatLng(centerLatitude, centerLongitude), startZoom);

    updateMarkers();

    GEvent.addListener(map,'zoomend',function() {
        updateMarkers();
    });
    GEvent.addListener(map,'moveend',function() {
        updateMarkers();
    });
}
```

```
function updateMarkers() {

    //remove the existing points
    map.clearOverlays();
    //create the boundary for the data to provide
    //initial filtering
    var bounds = map.getBounds();
    var southWest = bounds.getSouthWest();
    var northEast = bounds.getNorthEast();
    var url = '/chap_seven/clustered_towers?ne=' + northEast.toUrlValue() +➥
      '&sw=' + southWest.toUrlValue();

    //log the URL for testing
    GLog.writeUrl(url);

    //retrieve the points
    var request = GXmlHttp.create();
    request.open('GET', url, true);
    request.onreadystatechange = function() {
         if (request.readyState == 4) {
              var data = request.responseText;
              var points = eval('('+data+')');

              //create each point from the list
              for (var i=0; i < points.length; i++) {
                  var point = new GLatLng(points[i].latitude,points[i].longitude);
                  var marker = createMarker(point,points[i].type);
                  map.addOverlay(marker);
              }
         }
    }
    request.send(null);
}

function createMarker(point, type) {
     //create the marker with the appropriate icon
     if(type=='c') {
         var marker = new GMarker(point,iconCluster,true);
     } else {
         var marker = new GMarker(point,iconSingle,true);
     }
    return marker;
}

window.onload = init;
```

Listing 7-7. *clustered_towers Action (add to app/controllers/chap_seven_controller.rb)*

```
  def clustered_towers
    ne = params[:ne].split(',').collect{|e|e.to_f}
    sw = params[:sw].split(',').collect{|e|e.to_f}

    # get all the towers within the bounds. No need to worry about the
    # meridian here, since all towers are in the US
    towers=Tower.find :all,
      :select=>['latitude, longitude, structure_type, owner_name'],
      :conditions => ['longitude > ? AND longitude < ? AND latitude <= ? AND➥
latitude >= ?',sw[1],ne[1],ne[0],sw[0]]

    # limit to 30 markers
    max_markers=30
    lng_span=0
    lat_span=0
    clustered=Hash.new

    logger.debug("Starting clustering with #{towers.length} towers.")

    # we'll probably have to loop a few times to get
    # the number of clustered markers below the max_markers value
    loop do
      # Initially, each cell in the grid is 1/30th of the longitudinal span,
      # and 1/30th of the latitudinal span. With multiple iterations of the loop,
      # the spans get larger (and therefore clusters get more markers)
      lng_span+=(ne[1]-sw[1])/30
      lat_span+=(ne[0]-sw[0])/30

      # This is where we put the towers we've clustered into groups
      # the key of the hash is the coordinates of the grid cell,
      # and the value is an array of towers which have been assigned to the cell
      clustered=Hash.new

      # we're going to be iterating however many times we need to
      towers.each do |tower|
        # determine which grid square this marker should be in
        grid_y=((tower.latitude-sw[0])/lat_span).floor
        grid_x=((tower.longitude-sw[1])/lng_span).floor

        # create a new array if it doesn't exist
        key="#{grid_x}_#{grid_y}"
        (clustered[key]=Array.new) if !clustered.has_key?(key)
        clustered[key].push(tower)
      end # end of iterating through each tower
```

```
    logger.debug "Current clustering has #{clustered.size} elements."
    break unless clustered.length > max_markers
  end

  # At this point we've got max_markers or fewer points to render.
  # Now, let's go through and determine which cells have multiple markers
  # (which needs to be rendered as a cluster),
  # and which cells have a single marker
  result=Array.new
  clustered.each_value do |tower_array|
    # regardless of whether this is a cluster or an individual, set
    # the coordinates to the tower in the array
    marker = {:latitude => tower_array[0].latitude,
              :longitude => tower_array[0].longitude,
              :structure_type => tower_array[0].structure_type,
              :type => 'm'}

    # here's where we differentiate between clusters and individual markers
    marker[:type]='c' if tower_array.size > 1
    result.push(marker)
  end

  render :text => result.to_json
end
```

Once the files are in place, direct your browser to `http://localhost:3000/chap_seven/map` to see the results. You may want to examine the JSON generated by this action. This URL generates a JSON structure with nearly all the markers clustered: `http://localhost:3000/chap_seven/clustered_towers?ne=42.356514,-71.327362&sw=41.642131,-72.674561`. This URL generates JSON with just several towers clustered: `http://localhost:3000/chap_seven/clustered_towers?ne=42.083446,-71.682358&sw=41.904832,-72.017441`.

These are good starting points for your clustering code. Of course, there is room for improvement. For example, you could calculate an average position of the markers within one grid cell so that the cluster marker better represents the actual location of the points in that cell. You could also develop an algorithm that would allow you to cluster based on relative positions, so only dense groups would cluster rather than the entire page.

The advantages of the cluster method are that it isn't restricted to zoom levels and it works for any sized data set. Its disadvantage is that the data is clustered over possibly large areas, so you will still need to zoom in for more detail. An additional disadvantage is that dense data can take multiple iterations to reduce to an acceptable number of markers.

Custom Detail Overlay Method

So far, all the solutions we've presented use the `GMarker` to represent the data points on the map. With the release of Google Maps API version 2, Google has exposed additional classes in the API for building your own custom overlays.

An *overlay*, as we mentioned earlier, is anything that you add to the map, such as a `GMarker`, `GPolyline`, or an info window. In version 1 of the API, you were limited to the Google-provided

overlays. Now you can implement your own overlays using the `GOverlay` class. This opens up a realm of possibilities for creating overlays such as simple shapes or even your own info window object. Here we present the possibility of including a detail overlay for a specified area of the map.

The custom overlay you create can contain any information you want. For example, the Google Maps API documentation gives the example of a `Rectangle` overlay, as listed in Listing 7-8 (from `http://www.google.com/apis/maps/documentation/#Custom_Overlays`).

Listing 7-8. *Google's Example Rectangle Overlay*

```
// A Rectangle is a simple overlay that outlines a lat/lng bounds on the
// map. It has a border of the given weight and color and can optionally
// have a semi-transparent background color.
function Rectangle(bounds, opt_weight, opt_color) {
  this.bounds_ = bounds;
  this.weight_ = opt_weight || 2;
  this.color_ = opt_color || "#888888";
}
Rectangle.prototype = new GOverlay();

// Creates the DIV representing this rectangle.
Rectangle.prototype.initialize = function(map) {
  // Create the DIV representing our rectangle
  var div = document.createElement("div");
  div.style.border = this.weight_ + "px solid " + this.color_;
  div.style.position = "absolute";

  // Our rectangle is flat against the map, so we add our selves to the
  // MAP_PANE pane, which is at the same z-index as the map itself (i.e.,
  // below the marker shadows)
  map.getPane(G_MAP_MAP_PANE).appendChild(div);

  this.map_ = map;
  this.div_ = div;
}

// Remove the main DIV from the map pane
Rectangle.prototype.remove = function() {
  this.div_.parentNode.removeChild(this.div_);
}

// Copy our data to a new Rectangle
Rectangle.prototype.copy = function() {
  return new Rectangle(this.bounds_, this.weight_, this.color_,
                       this.backgroundColor_, this.opacity_);
}
```

```
// Redraw the rectangle based on the current projection and zoom level
Rectangle.prototype.redraw = function(force) {
  // We only need to redraw if the coordinate system has changed
  if (!force) return;

  // Calculate the DIV coordinates of two opposite corners of our bounds to
  // get the size and position of our rectangle
  var c1 = this.map_.fromLatLngToDivPixel(this.bounds_.getSouthWest());
  var c2 = this.map_.fromLatLngToDivPixel(this.bounds_.getNorthEast());

  // Now position our DIV based on the DIV coordinates of our bounds
  this.div_.style.width = Math.abs(c2.x - c1.x) + "px";
  this.div_.style.height = Math.abs(c2.y - c1.y) + "px";
  this.div_.style.left = (Math.min(c2.x, c1.x) - this.weight_) + "px";
  this.div_.style.top = (Math.min(c2.y, c1.y) - this.weight_) + "px";
}

function load() {
  if (GBrowserIsCompatible()) {
        var map = new GMap2(document.getElementById("map"));
        map.addControl(new GSmallMapControl());
        map.addControl(new GMapTypeControl());
        map.setCenter(new GLatLng(37.4419, -122.1419), 13);

        // Display a rectangle in the center of the map at about a quarter of
        // the size of the main map
        var bounds = map.getBounds();
        var southWest = bounds.getSouthWest();
        var northEast = bounds.getNorthEast();
        var lngDelta = (northEast.lng() - southWest.lng()) / 4;
        var latDelta = (northEast.lat() - southWest.lat()) / 4;
        var rectBounds = new GLatLngBounds(
        new GLatLng(southWest.lat() + latDelta,
            southWest.lng() + lngDelta),
        new GLatLng(northEast.lat() - latDelta,
            northEast.lng() - lngDelta));
        map.addOverlay(new Rectangle(rectBounds));
    }
}

window.onload = load;
```

The `Rectangle` overlay simply creates a `div` object on the map and applies a border to it. To create a detail overlay, you can use the `Rectangle` object in Listing 7-8, but add one additional property to the `div`: a background image. The background image can contain any information you want, from pictures and icons to lines and shapes, and can be created on the fly using a server-side script. The new custom detail overlay can then be placed on the map in the appropriate area on top of the existing Google Maps tiles.

Using an overlay is best for data sets that are high density but cover a relatively small portion of the map. If your data set contains hundreds of thousands of points, creating the overlay is going to take some time, and your application will still feel sluggish. If you have massive data sets spread across the world, you'll need to use custom tiles, which we'll discuss in the next section.

Suppose you want to mark all the FCC tower locations in Hawaii, as you did in Chapter 6. There are about 286 towers—too many for one map using just the `GMarker` object. Using a custom overlay, you can simply create a transparent GIF or PNG that covers all of Hawaii and mark each of the locations in whatever way you like. You can even add text, shapes, or photos. What you include in your image is up to you.

You can create a custom overlay programmatically on your server using RMagick, a powerful library for creating and manipulating graphics files with Ruby. RMagick provides a Ruby API to ImageMagick, which is a general-purpose (i.e., non-Ruby-specific) graphics manipulation library. You'll utilize ImageMagick more in Chapter 9.

To learn more about RMagick, visit its RubyForge home page at `http://rmagick.rubyforge.org/`. To learn more about ImageMagick, go to `http://www.imagemagick.org/`.

See the sidebar "Installing RMagick and Writing a Test Action" for help getting RMagick up and running.

INSTALLING RMAGICK AND WRITING A TEST ACTION

Before we create the custom overlay, let's install RMagick and write a simple test action to make sure we can use RMagick in Rails. Installation instructions for multiple operating systems are here: `http://rmagick.rubyforge.org/install-faq.html`. If you're installing on Windows, you can skip right to the `rmagick-win32` download at `http://rubyforge.org/projects/rmagick/` (under the "Latest File Releases" section), and follow the directions in the download. If you're running OSX, follow the instructions at `http://rmagick.rubyforge.org/install-osx.html`.

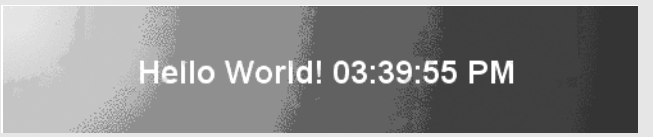
After you've installed RMagick, create a Rails action to generate a simple image. The following code shows an `image_test` action you can add to `app/controllers/chap_seven.rb`. This action generates white text on a blue gradient background:

```
def image_test
  fill = Magick::GradientFill.new(0, 0, 0, 0, '#eee', '#00f')
  canvas = Magick::Image.new(500, 100,fill)
  canvas.format = "GIF"

    text = Magick::Draw.new
    text.annotate(canvas, 0, 0, 0, 0, "Hello World! #{Time.now.strftime➥
('%I:%M:%S %p')}") {
      self.gravity = Magick::CenterGravity
      self.pointsize = 24
      self.fill = 'white'
      }

  send_data canvas.to_blob, :type => 'image/gif', :disposition => 'inline'
end
```

In addition to the `image_test` action, you must include the RMagick library in your controller. Include the following two bold lines at the top of your `app/controllers/chap_seven.rb` file:

```
require 'RMagick'
include Magick

class ChapSevenController < ApplicationController
...
```

Once you've added this code to `app/controllers/chap_seven.rb`, point your browser to `http://localhost:3000/chap_seven/image_test`. You should see an image similar to the following:



Congratulations! You have dynamically generated an image using RMagick and output it to your browser through a Rails action. If you refresh your browser, you will see that the image is regenerated each time, as reflected by the time embedded in the image.

Once you have RMagick installed, take a look at Listing 7-9, which shows the client-side JavaScript for the custom overlay method. You should replace your existing `public/javascripts/application.js` file with this code.

Listing 7-9. *public/javascripts/applicatoin.js for the Custom Overlay Method (Replaces Existing File)*

```
var map;
var centerLatitude = 19.9;
var centerLongitude = -156;
var startZoom = 7;

//create the Detail overlay object
function Detail(bounds, opt_weight, opt_color) {
    this.bounds_ = bounds;
    this.weight_ = opt_weight || 2;
    this.color_ = opt_color || "#000";
}
Detail.prototype = new GOverlay();

Detail.prototype.initialize = function(map) {
    //create the div representing the Detail
    var div = document.createElement("div");
    div.style.border = this.weight_ + "px dotted " + this.color_;
    div.style.position = "absolute";
```

```
    //the Detail is flat against the map, so we add it to the
    //MAP_PANE pane, which is at the same z-index as the map itself (i.e.,
    //below the marker shadows)
    map.getPane(G_MAP_MAP_PANE).appendChild(div);

    this.map_ = map;
    this.div_ = div;

    //load the background image
    this.loadBackground();
}

Detail.prototype.remove = function() {
    this.div_.parentNode.removeChild(this.div_);
}

Detail.prototype.copy = function() {
    return new Detail(this.bounds_, this.weight_, this.color_,
    this.backgroundColor_, this.opacity_);
}

Detail.prototype.redraw = function(force) {
    if (!force) return;

    this.bounds_ = this.map_.getBounds();

    var c1 = this.map_.fromLatLngToDivPixel(this.bounds_.getSouthWest());
    var c2 = this.map_.fromLatLngToDivPixel(this.bounds_.getNorthEast());

    this.div_.style.width = Math.abs(c2.x - c1.x) + "px";
    this.div_.style.height = Math.abs(c2.y - c1.y) + "px";
    this.div_.style.left = (Math.min(c2.x, c1.x) - this.weight_) + "px";
    this.div_.style.top = (Math.min(c2.y, c1.y) - this.weight_) + "px";

    //the position or zoom has changed so reload the background image
    this.loadBackground();
}

Detail.prototype.loadBackground = function() {

    //retrieve the bounds of the detail area
    var southWest = this.bounds_.getSouthWest();
    var northEast = this.bounds_.getNorthEast();

    //determine the pixel position of the corners
    var swPixels = this.map_.fromLatLngToDivPixel(this.bounds_.getSouthWest());
    var nePixels = this.map_.fromLatLngToDivPixel(this.bounds_.getNorthEast());
```

```
    //send the lat/lng as well as x/y and zoom to the server
    var image_url = '/chap_seven/get_overlay?ne=' + northEast.toUrlValue()
        + '&sw=' + southWest.toUrlValue()
        + '&nePixels=' + nePixels.x + ',' + nePixels.y
        + '&swPixels=' + swPixels.x + ',' + swPixels.y
        + '&zoom=' + this.map_.getZoom()
        + '';

    //log the URL for testing
    GLog.writeUrl(image_url);

    //set the background image of the div
    this.div_.style.background='transparent url('+image_url+')';
}


function init() {
    map = new GMap2(document.getElementById("map"));
    map.addControl(new GSmallMapControl());
    map.setCenter(new GLatLng(centerLatitude, centerLongitude), startZoom);

    var bounds = map.getBounds();

    map.addOverlay(new Detail(bounds));

}

window.onload = init;
```

■ **Tip** For examples of the mathematical formulas for different maps such as the Mercator projection maps, visit MathWorld at `http://mathworld.wolfram.com/MercatorProjection.html`.

Looking at Listing 7-9, you can see the `Rectangle` object renamed to `Detail` and the addition of a `loadBackground` method, which modifies the `background` style property of the `Detail` object. The following excerpt from Listing 7-9 shows the `loadBackground` function:

```
Detail.prototype.loadBackground = function() {
    //retrieve the bounds of the detail area
    var southWest = this.bounds_.getSouthWest();
    var northEast = this.bounds_.getNorthEast();

    //send the lat/lng as well as x/y and zoom to the server
    var image_url = '/chap_seven/get_overlay?ne=' + northEast.toUrlValue()
        + '&sw=' + southWest.toUrlValue()
        + '&nePixels=' + nePixels.x + ',' + nePixels.y
        + '&swPixels=' + swPixels.x + ',' + swPixels.y
```

```
        + '&zoom=' + this.map_.getZoom()
        + '';

    //log the URL for testing
    GLog.writeUrl(image_url);

    //set the background image of the div
    this.div_.style.background='transparent url('+image_url+')';
}
```

When loading your background image, you'll need to include several variables for your server-side action, including the northeast and southwest corners in latitude and longitude, as well as the northeast and southwest corners in pixel values. You also need to pass the current zoom level for the map. This will allow you to perform the necessary calculations on the server side and also allow you to modify your image, depending on how far your users have zoomed in on the map. The image is generated by the `get_overlay` action shown in Listing 7-10. Add this code to your `app/controllers/chap_seven.rb` controller to implement this example. For the example in Listing 7-10, we've chosen to create a GIF with a small circle marking each tower location within the northeast and southwest boundary.

Listing 7-10. *get_overlay Action (Add to app/controllers/chap_seven.rb)*

```
  def get_overlay
    ne = params[:ne].split(',').collect{|e|e.to_f}
    sw = params[:sw].split(',').collect{|e|e.to_f}
    nePx = params[:nePixels].split(',').collect{|e|e.to_i}
    swPx = params[:swPixels].split(',').collect{|e|e.to_i}

    width = (nePx[0]-swPx[0]).abs.to_i
    height = (nePx[1]-swPx[1]).abs.to_i
    zoom = (params[:zoom]).to_i

    towers=Tower.find :all,
      :select=>['latitude, longitude, structure_type, owner_name'],
      :conditions => ['longitude > ? AND longitude < ? AND latitude <= ? AND➥
latitude >= ?',sw[1],ne[1],ne[0],sw[0]]

    #calculate the Mercator coordinate position of the top
    #latitude and longitude and normalize from 0-1
    merc_top = 0.5-(Math::asinh(Math::tan((ne[0]/180.0)*Math::PI)) / Math::PI / 2)

    #calculate the scale and y position of the google map
    scale = (1<< zoom)*256
    y_top = merc_top * scale

    #calculate the pixels per degree of longitude
    lng_span=ne[1]-sw[1]
    pixels_per_deg_lng=(width/lng_span).abs
```

```
    #create the image
    canvas = Magick::Image.new(width, height) {self.background_color='transparent'}
    canvas.format = "GIF"

    gc = Magick::Draw.new

    gc.stroke('white')
    gc.fill('black')
    gc.stroke_width(1)

    towers.each do |tower|

      x = (((tower.longitude-sw[1]).abs)*pixels_per_deg_lng).ceil

      # calculate the Mercator coordinate position of this point's
      # latitude and longitude and normalize from 0-1
      y_merc = 0.5-(Math::asinh(Math::tan((tower.latitude/180.0)*Math::PI)) /➥
Math::PI / 2)
      #calculate the y position on the Google map
      y_map = y_merc * scale
      #calculate the y position in the overlay
      y = y_map - y_top

      gc.circle(x,y, x+(0.5*zoom).ceil,y+(0.5*zoom).ceil)
    end

    gc.draw(canvas)

    send_data canvas.to_blob, :type => 'image/gif', :disposition => 'inline'
  end
```

Looking back at the JavaScript in Listing 7-9 again, you'll notice that your background image for the overlay is based on the viewable area of the map. You can imagine, when you zoom in very close, the image covering all of Hawaii would be exponentially larger at each zoom increment. Limiting the image to cover only the viewable area decreases the number of points that need to be drawn and decreases the size of the image.

■ **Tip** Another advantage of the custom overlay method as well as the custom tile method, described next, is the ability to circumvent the same origin security policy built into most browsers. The policy doesn't apply to images, so your map can be hosted on one domain and you can request your background images or tiles from a different domain without any problems.

Once the overlay is loaded onto the map, you should have the towers for Hawaii marked as something like Figure 7-6. Point your browser to `http://localhost:3000/chap_seven/map` to see your custom overlay.

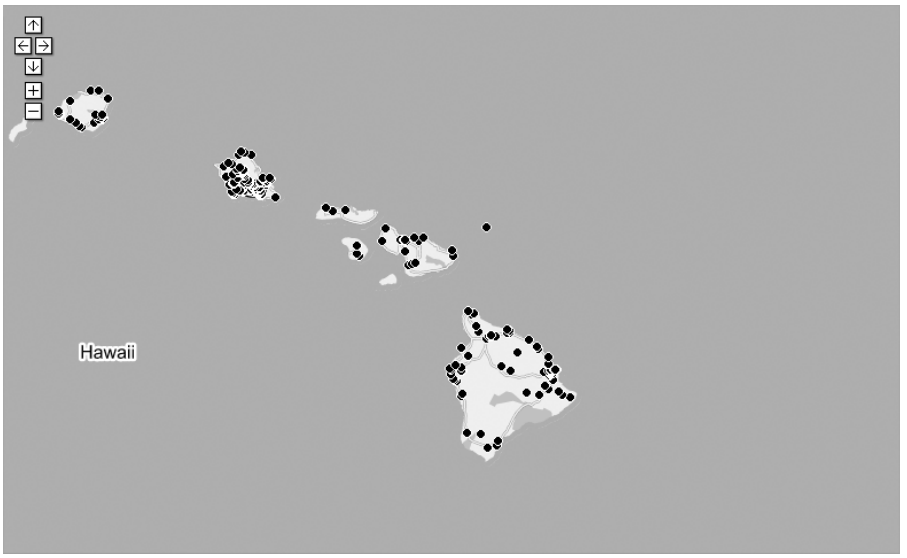


Figure 7-6. *A map showing the custom detail overlay for FCC towers in Hawaii*

If you need to debug the image generation action, you may want to view the generated image directly in your browser. You can use a URL such as `http://localhost:3000/chap_seven/get_overlay?ne=23.473324,-149.315186&sw=16.24632,-162.685547&nePixels=900,0&swPixels=0,700&zoom=7` to see the generated image in isolation.

The pros of using the custom overlay method are as follows:

- It overcomes API limitations on the number of markers and polylines.
- You can use the same method to display objects, shapes, photos, and more.
- It works for any size data set and at any zoom level.

The following are its disadvantages:

- It creates a new image after each map movement or zoom change.
- Extremely large data sets will be slow to render.

Custom Tile Method

The custom tile method is the most elegant solution to display the maximum amount of information on the map with the least overhead. You could use custom tiles to display a single point or millions of points.

To add your own custom tiles to the map, version 2 of the Google Maps API exposes the `GTile` and `GProjection` objects. This means you can now use the API to show your own tiles on the map. What's even better is that you can also layer transparent or translucent tiles on top of each other to create a multilayered map. By layering tiles on top of one another, you have no limit to what information you can display. For example, you could create tiles with your own driving directions, outline buildings and environmental features, or even display your information using an old antique map rather than Google's default or satellite map types.

FCC Tower Tiles

To demonstrate this method, let's create a map of all the available FCC towers in the United States. That's an excessively large amount of dense data (about 115,000 points, as mentioned earlier), and it covers a fairly large area of the earth. You could use the custom overlay method discussed in the previous section, but the map would be basically unusable as it continually redraws the image when looking at anything larger than a single city in a dense area. Your best option would be to create transparent tiles containing all your information and match them to Google's tiles so you can overlay them on top of each of the different map types. By slicing your data into smaller tiles, each image is relatively small (256 by 256 pixels) and both the client web browser and the server can cache them to reduce redundant processing. Figure 7-7 shows each of the tiles outlined on the sample Google map.

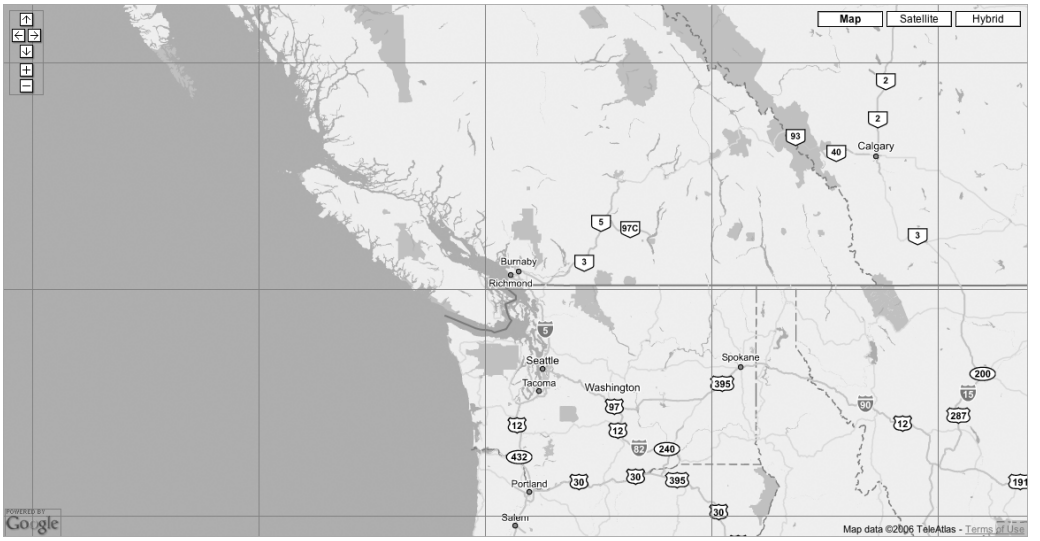


Figure 7-7. *Tiles outlined on a Google map*

To layer your data using the same tile structure as the Google Maps API, you'll need to create each of your tiles to match the existing Google tiles. Along with the sample code for the book, we've included a `GoogleMapsUtil` class in Listing 7-11, which has a variety of useful methods to help you create your tiles. The `get_tile` action (shown later in Listing 7-12) uses the methods of `GoogleMapsUtil` to calculate the various locations of each point on the tile. The calculations in the utility class are based on the Mercator projection, which we discuss further in Chapter 9 when we talk about types of map projections.

You should place the code in Listing 7-11 in a new file: `lib/google_maps_util.rb`.

Listing 7-11. *lib/google_maps_util.rb (New File)*

```
class GoogleMapsUtil
  # constants
  TILE_SIZE = 256
```

```
# Structures to represent points and boundaries
Point = Struct.new( "Point", :x, :y )
Boundary = Struct.new( "Boundary", :x, :y, :width, :height )


# calculate the pixel offset within a specific tile for
# the given latitude and longitude
def self.get_pixel_offset_in_tile(lat,lng,zoom)
  pixel_coords = GoogleMapsUtil.to_zoomed_pixel_coords(lat,lng,zoom)
  return Point.new(pixel_coords.x % GoogleMapsUtil::TILE_SIZE,
                   pixel_coords.y % GoogleMapsUtil::TILE_SIZE)
end

# convert from latitude and longitude to Mercator coordinates
def self.to_mercator_coords(lat,lng)
  if lng > 180
    lng -= 360
  end

  lng = lng / 360.0
  lat = Math::asinh(Math::tan((lat/180.0)*Math::PI))/Math::PI/2.0
  result = Point.new(lng,lat)
  return result
end

# normalize the mercator coordinates
def self.to_normalized_mercator_coords(point)
  point.x += 0.5
  point.y = ((point.y - 0.5).abs)
  return point
end

# translate to pixel coordinates within a tile
def self.to_zoomed_pixel_coords(lat,lng,zoom)
  normalized = GoogleMapsUtil.to_normalized_mercator_coords(
    GoogleMapsUtil.to_mercator_coords(lat,lng))

  scale = (1 << zoom) * GoogleMapsUtil::TILE_SIZE
```

```
    return Point.new((normalized.x * scale).to_i,
                     (normalized.y * scale).to_i)
  end

  # determine the geographical bounding box for the
  # specified tile index and zoom level
  def self.get_tile_rect(x,y,zoom)
    tiles_at_this_zoom = (1 << zoom)
    lng_width = 360.0 / tiles_at_this_zoom
    lng = -180 + (x * lng_width)

    lat_height_merc = 1.0 / tiles_at_this_zoom
    top_lat_merc = y * lat_height_merc
    bottom_lat_merc = top_lat_merc + lat_height_merc

    bottom_lat = (180 / Math::PI) * ((2*Math.atan(Math.exp(Math::PI *➥
(1 - (2 * bottom_lat_merc)))))-(Math::PI/2))
    top_lat = (180 / Math::PI) * ((2*Math.atan(Math.exp(Math::PI *➥
(1 - (2 * top_lat_merc)))))-(Math::PI/2))

    lat_height = top_lat - bottom_lat

    return Boundary.new(lng,bottom_lat,lng_width,lat_height)
  end
end
```

Using the `GoogleMapsUtil` class, you can determine what information you need to include in each tile. For example, in the client-side JavaScript for the custom tile method in Listing 7-12 (which you'll see later in this section), each tile request

```
var url="/chap_seven/get_tile?x="+tile.x+"&y="+tile.y+"&zoom="+zoom
```

contains three bits of information: an X position, a Y position, and the zoom level. These three bits of information can be used to calculate the latitude and longitude boundary of a specific Google tile using the `GoogleMapsUtil.getTileRect` method, as demonstrated in the custom tile action in Listing 7-13 (also coming up later in this section). The X and Y positions represent the tile number of the map relative to the top left corner, where positive X and Y are east and south, respectively, starting at 1 and increasing, as illustrated in Figure 7-8. You can also see that the first column in Figure 7-8 contains tile (7,1) because the map has wrapped beyond the meridian; so the first column is actually the rightmost edge of the map, and the second column is the leftmost edge.

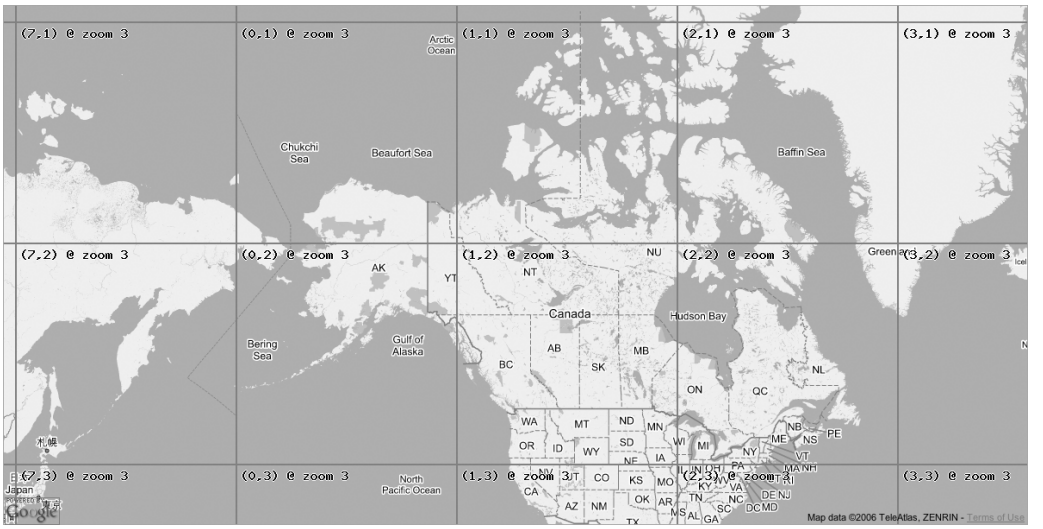


Figure 7-8. *Google tile numbering scheme*

The zoom level is also required so that the calculations can determine the latitude and longitude resolution of the current map. For now, play with the example in Listings 7-12 and 7-13 (`http://book.earthcode.com/chap_seven/server_custom_tiles`). In Chapter 9, you'll get into the math required to calculate the proper position of latitude and longitude on the Mercator projection, as well as a few other projections.

For the sample tiles, we've drawn a circle for each tower. You can see the results in Figure 7-9. Although you cannot see it in this gray-scale screenshot, we have made the color of each circle represent the height of the tower.

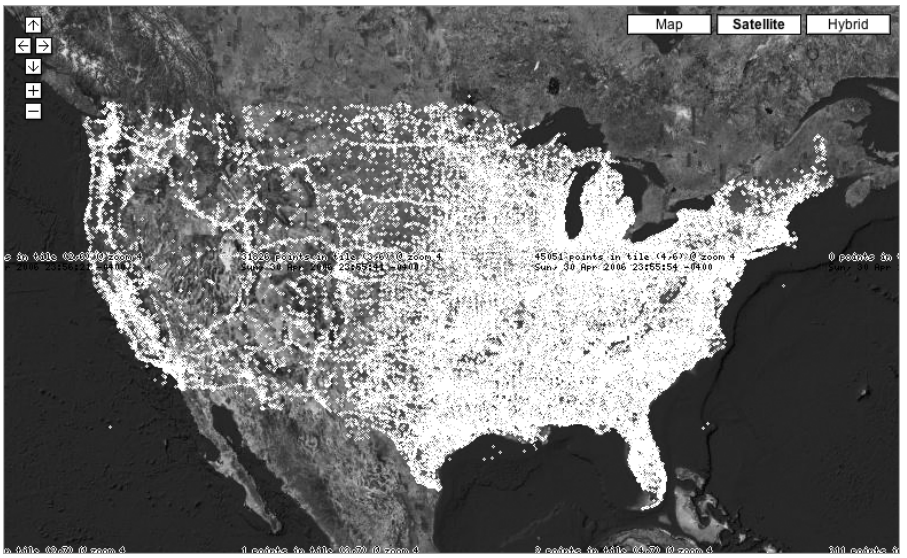


Figure 7-9. *The finalized custom tile map in satellite mode*

For testing purposes, each tile is also labeled with the date/time tile number and the number of points in that tile. Once drawn, the tiles are cached on the server side so when requested again, the tiles are automatically served up by the server.

Originally, when the tiles were created for zoom level 1, some took up to a minute to render on our development laptops. If the data on your map is continually changing, you should consider running a script to prerender the tiles at common zoom levels before exposing your map publicly. That way, your first visitors won't experience the lag associated with initial tile rendering. Note that you won't be able to prerender all tiles at every zoom level; doing so would take an exorbitant amount of disk space, as you'll see in Chapter 11. At the very least, you should prerender the tiles for your default view and zoom level.

Listing 7-12. *Client-Side JavaScript for the Custom Tile Method*

```
var map;
var centerLatitude = 40.598666;
var centerLongitude = -100.949219;
var startZoom = 1;

//create the tile layer object
var detailLayer = new GTileLayer(new GCopyrightCollection(''));

//method to retrieve the URL of the tile
detailLayer.getTileUrl = function(tile, zoom){
    //provide the x and y position as well as the zoom
    return url="/chap_seven/get_tile?x=" + tile.x +
                "&y="+tile.y+"&zoom="+zoom
};

detailLayer.isPng = function() {
    return true;
}

//add your tiles to the normal map projection
detailMapLayers = G_NORMAL_MAP.getTileLayers();
detailMapLayers.push(detailLayer);

//add your tiles to the satellite map projection
detailMapLayers = G_SATELLITE_MAP.getTileLayers();
detailMapLayers.push(detailLayer);

function init() {
    map = new GMap2(document.getElementById("map"));
    map.addControl(new GSmallMapControl());
    map.addControl(new GMapTypeControl());

    map.setCenter(new GLatLng(centerLatitude, centerLongitude), startZoom);
}

window.onload = init;
```

Listing 7-13. *get_tile Action (Add to app/controllers/chap_seven.rb)*

```
  def get_tile
    x,y,zoom = [params[:x].to_i,params[:y].to_i,params[:zoom].to_i]
    #make sure our tile store exists, create it if it doesn't
    directory='public/map_tiles'
    if ! File.exists? directory
      FileUtils.mkdir_p directory
    end

    #create a unique file name for this file
    filename = "tile-#{x}-#{y}-#{zoom}.png"
    full_path = File.join(directory, filename)

    File.delete(full_path) if File.exist?(full_path)

    # create the file if it doesn't already exist
    if !File.exist?(full_path)

      # create an array of marker size at each zoom level
      marker_sizes=[1,1,1,1,1,1,2,2,2,2,3,3,3,3,3,3,4,4,4,4,5,5,5,5,6,6]

      # get the lat/lng bounds of this tile from the helper method
      rect = GoogleMapsUtil.get_tile_rect(x,y,zoom)

      #init some vars
      swlat = rect.y
      swlng = rect.x
      nelat = swlat+rect.height
      nelng = swlng+rect.width

      # get the towers
      towers=Tower.find :all,
        :select=>['latitude, longitude, structure_type, height'],
        :conditions => ['longitude >= ? AND longitude <= ? AND latitude <= ? AND➥
latitude >= ?',swlng,nelng,nelat,swlat]

      canvas = Magick::Image.new(GoogleMapsUtil::TILE_SIZE,➥
GoogleMapsUtil::TILE_SIZE){
        self.background_color='transparent'}
      canvas.format="PNG"

      gc = Magick::Draw.new
```

```
      towers.each do |tower|
        point = GoogleMapsUtil.get_pixel_offset_in_tile(tower.latitude,➥
tower.longitude,zoom)

        # pick a color based on the tower's height
        case tower.height
          when 0...20
            color='darkred'
          when 21...40
            color='red'
          when 41...80
            color='darkgreen'
          when 81...120
            color='green'
          when 121...200
            color='darkblue'
          else
            color='blue'
        end
        gc.stroke('white')
        gc.fill(color)

        # get the size
        size=marker_sizes[zoom]
        if zoom < 2
          canvas.pixel_color point.x, point.y, color
        else
          gc.circle(point.x-size,point.y-size,point.x+size,point.y+size)
        end

      end

      gc.draw(canvas)

      text = Magick::Draw.new
      text.annotate(canvas, 0, 0, 0, 0, "#{towers.size} points in tile➥
 #{x},#{y} @ zoom #{zoom}.") {
        self.gravity = Magick::SouthGravity
        self.fill = 'black'
        self.text_antialias=true;
        }
      canvas.write(full_path)
    end

    send_file full_path, :type => 'image/png', :disposition => 'inline'

  end
```

The following are a few points of interest in the `get_tile` action:

- This action stores generated tiles in the `/public/map_tiles` directory. If the directory doesn't exist, the code creates it.
- As long as the requested tile already exists in `/public/map_tiles`, the code will serve up the already generated tile rather than create a new one. If you make a change in the tile generation code, you need to clear out the `map_tiles` directory to see the changes take effect.
- If you want to see a single tile generated in isolation, try this URL: `http://localhost:3000/chap_seven/get_tile?x=15&y=112&zoom=8`. This URL generates a tile that covers the Hawaiian island of Kauai.

What About Info Windows?

Using tiles to display your points on a map is relatively easy, and you can simulate most of the features of the `GMarker` object, with the exception of info windows. You can't attach an info window to the pretend markers in your tile, but you can fake it.

Back in Chapter 3, you created an info window when you clicked on the map by using `GMap2.openInfoWindow`. You could do the same here, and then use an Ajax request to ask for the content of the info window using something like this:

```
GEvent.addListener(map, "click", function(marker, point) {
    GDownloadUrl(
        "/chap_seven/your_action?"
        + "lat=" + point.lat()
        + "&lng=" + point.lng()
        + "&z=" + map.getZoom(),
        function(data, responseCode) {
            map.openInfoWindow(point,document.createTextNode(data));
        });
 });
```

The trick is figuring out what is actually clicked. When your users click your map, you'll need to send the location's latitude and longitude back to the server and have it determine what information is relative to that point. If something is clicked, you can then send the appropriate information back across the Ajax request and create an info window directly on the map. From the client's point of view, it will look identical to an info window attached to a marker, except that it will be slightly slower to appear, as your server needs to process the request to see what was clicked.

Optimizing the Client-Side User Experience

If your data set is too big to display all the markers at once, but still reasonably sized, you don't necessarily need to make new requests to retrieve your information. You can achieve good results using solutions similar to those we've outlined for the server side, while storing the data set in the browser's memory using a JavaScript object. This way, you can achieve the same effect but not require an excessive number of requests to the server.

The three client-side methods we will discuss are pretty much the same as the corresponding server-side methods, except that the processing is all done on the client side using the methods of the API rather than calculating everything on the server side:

- Client-side boundary method
- Client-side closest-to-a-common-point method
- Client-side clustering

After we look at these solutions using client-side JavaScript and data objects, we'll recommend a couple other optimizations to improve your users' experience.

Client-Side Boundary Method

With the server-side boundary method, you used conditions in your `find` call to restrict the database query to points inside the boundary of the map. The Google Maps API provides a different approach; you can use the `contains()` method of the `GLatLngBounds` object to ask the API if your `GLatLng` point is within the specified boundary. The `contains()` method returns `true` if the supplied point is within the geographical coordinates defined by the rectangular boundary.

To get these examples working, you need to add a `map` action to your `app/controllers/chap_seven.rb` controller file. The map action needs just a few lines of code, as shown in Listing 7-14.

Listing 7-14. *map Action (Add to app/controllers/chap_seven.rb)*

```
def map
  @towers = Tower.find(:all,
          :select=>"latitude, longitude",
          :conditions=>['longitude > ? AND longitude < ? AND latitude <= ? AND➥
latitude >= ?',-73.3996,-71.7517,42.5894,41.570251])
end
```

You also need to modify your `app/views/chap_seven/map.rhtml` file to output the list of towers in JSON format for the JavaScript code to utilize. Listing 7-15 shows the highlighted lines you should add to `map.rhtml`.

Listing 7-15. *Add Bold Lines to app/views/chap_seven/map.rhtml*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"➥
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <script src="http://maps.google.com/maps?file=api&amp;v=2&amp;key=YOUR_API_KEY"➥
 type="text/javascript"></script>
  <%=javascript_include_tag 'application'%>
  <script type="text/javascript">
    var points=<%=@towers.collect{|c|c.attributes}.to_json %>;
  </script>
</head>
```

```
<body>
  <div id="map" style="width:100%;height:700px"></div>
</body>
</html>
```

Listing 7-16 (http://book.earthcode.com/chap_seven/client_bounds) shows the working example of the boundary method implemented in JavaScript.

Listing 7-16. *JavaScript for the Client-Side Boundary Method*

```
var map;
var centerLatitude = 42.326062;
var centerLongitude = -72.290039;
var startZoom = 11;

function init() {
    map = new GMap2(document.getElementById('map'));
    map.addControl(new GSmallMapControl());
    map.setCenter(new GLatLng(centerLatitude, centerLongitude), startZoom);

    updateMarkers();

    GEvent.addListener(map,'zoomend',function() {
        updateMarkers();
    });
    GEvent.addListener(map,'moveend',function() {
        updateMarkers();
    });
}

function updateMarkers() {
    map.clearOverlays();
    var mapBounds = map.getBounds();

    //loop through each of the points from the global points object
    for (k=0;k<points.length;k++) {
        var latlng = new GLatLng(points[k].latitude,points[k].longitude);
        if(!mapBounds.contains(latlng)) continue;
        var marker = createMarker(latlng);
        map.addOverlay(marker);
    }
}

function createMarker(point) {
    var marker = new GMarker(point);
    return marker;
}

window.onload = init;
```

When you move or zoom the map, the `updateMarkers()` function loops through a `points` object to create the necessary markers for the boundary of the viewable area. The `points` JSON object resembles the object discussed in the "Streamlining Server-Client Communications" section earlier in this chapter:

```
var points=[{"latitude": 42.5399, "longitude": -73.381},
            {"latitude": 42.1909, "longitude": -73.3718},
            {"latitude": 41.6058, "longitude": -73.3675},
            {"latitude": 42.1282, "longitude": -73.3663},
            {"latitude": 41.9094, "longitude": -73.366},
            ... etc ...
           ];
```

The points object was set up inline in the page in the `map.rhtml` page. Now, rather than create a new request to the server, the `points` object contains all the points, so you only need to loop through `points` and determine whether the current point is within the current boundary. Listing 7-16 uses the current boundary of the map from `map.getBounds()`.

Client-Side Closest-to-a-Common-Point Method

If you don't have too many points, you can use the Google Maps API to implement the "closest to common point" approach. With a known latitude and longitude point, you can calculate the distance from the known point to any other point using the `distanceFrom()` method of the `GLatLng` class as follows:

```
var here = new GLatLng(lat,lng);
var distanceFromThereToHere = here.distanceFrom(there);
```

The `distanceFrom()` method returns the distance between the two points in meters, but remember that the Google Maps API assumes the earth is a sphere, even though the earth is slightly elliptical, so the accuracy of the distance may be off by as much as 0.3%, depending where the two points are on the globe.

In Listing 7-17 (`http://book.earthcode.com/chap_seven/client_closest`), you can see the client-side JavaScript is functionally very similar to the server-side closest-point code in Listing 7-5 (the `nearby_towers` action). The main difference (besides not sending a request to the server) is the use of `point.distanceFrom()` rather than calculating the closest points within the database query. Also for the example, the boundary of the data is outlined using the `Rectangle` object, similar to the one discussed earlier in the "Custom Detail Overlay Method" section.

Listing 7-17. *JavaScript for the Client-Side Closest-to-Common-Point Method*

```
var map;
var centerLatitude = 41.8;
var centerLongitude = -72.3;
var startZoom = 8;

function init() {
    map = new GMap2(document.getElementById("map"));
    map.addControl(new GSmallMapControl());
    map.setCenter(new GLatLng(centerLatitude, centerLongitude), startZoom);
```

```
        //pass in an initial point for the center
        updateMarkers(new GLatLng(centerLatitude, centerLongitude));

        GEvent.addListener(map,'click',function(overlay,point) {
            //pass in the point for the center
            updateMarkers(point);
        });

    }

    function updateMarkers(relativeTo) {

        //remove the existing points
        map.clearOverlays();

        //mark the outer boundary of the data from the points object
        var allsw = new GLatLng(41.57025176609894, -73.39965820312499);
        var allne = new GLatLng(42.589488572714245, -71.751708984375);
        var allmapBounds = new GLatLngBounds(allsw,allne);
        map.addOverlay(new Rectangle(allmapBounds,4,"#F00"));

        var distanceList = [];
        var p = 0;
        //loop through points and get the distance to each point
        for (k=0;k<points.length;k++) {
            distanceList[p] = {};
            distanceList[p].glatlng = new➥
    GLatLng(points[k].latitude,points[k].longitude);
            distanceList[p].distance = distanceList[p].glatlng.distanceFrom(relativeTo);
            p++;
        }

        //sort based on the distance
        distanceList.sort(function (a,b) {
            if(a.distance > b.distance) return 1
            if(a.distance < b.distance) return -1
            return 0
        });

        //create the first 50 markers
        for (i=0 ; i<50 ; i++) {
            var marker = createMarker(distanceList[i].glatlng);
            map.addOverlay(marker);
            if(++i > 50) break;
        }
    }
```

```
function createMarker(point) {
    var marker = new GMarker(point);
    return marker;
}

window.onload = init;


/*
* Rectangle overlay for testing to mark boundaries
*/
function Rectangle(bounds, opt_weight, opt_color) {
  this.bounds_ = bounds;
  this.weight_ = opt_weight || 1;
  this.color_ = opt_color || "#888888";
}
Rectangle.prototype = new GOverlay();

Rectangle.prototype.initialize = function(map) {
  var div = document.createElement("div");
  div.innerHTML = '<strong>Click inside area</strong>';
  div.style.border = this.weight_ + "px solid " + this.color_;
  div.style.position = "absolute";
  map.getPane(G_MAP_MAP_PANE).appendChild(div);
  this.map_ = map;
  this.div_ = div;
}
Rectangle.prototype.remove = function() {
  this.div_.parentNode.removeChild(this.div_);
}
Rectangle.prototype.copy = function() {
  return new Rectangle(
    this.bounds_,
    this.weight_,
    this.color_,
    this.backgroundColor_,
    this.opacity_
  );
}
Rectangle.prototype.redraw = function(force) {
  if (!force) return;
  var c1 = this.map_.fromLatLngToDivPixel(this.bounds_.getSouthWest());
  var c2 = this.map_.fromLatLngToDivPixel(this.bounds_.getNorthEast());
  this.div_.style.width = Math.abs(c2.x - c1.x) + "px";
  this.div_.style.height = Math.abs(c2.y - c1.y) + "px";
  this.div_.style.left = (Math.min(c2.x, c1.x) - this.weight_) + "px";
  this.div_.style.top = (Math.min(c2.y, c1.y) - this.weight_) + "px";
}
```

Client-Side Clustering

If your data is dense, you may still want to cluster points when there are overlapping points in proximity. As with the server-side clustering method, there are a variety of ways you can calculate which points to group. In Listing 7-18 (`http://book.earthcode.com/chap_seven/client_cluster`), we use a grid method similar to the one we used with the server-side clustering example. The biggest difference here is your grid cells will be larger and not as fine-grained, so you don't slow down the JavaScript on slower computers.

Listing 7-18. *JavaScript for Client-Side Clustering*

```
var map;
var centerLatitude = 42.326062;
var centerLongitude = -72.290039;
var startZoom = 8;

//create an icon for the clusters
var iconCluster = new GIcon();
iconCluster.image = "http://googlemapsbook.com/chapter7/icons/cluster.png";
iconCluster.shadow = "http://googlemapsbook.com/chapter7/icons/cluster_shadow.png";
iconCluster.iconSize = new GSize(26, 25);
iconCluster.shadowSize = new GSize(22, 20);
iconCluster.iconAnchor = new GPoint(13, 25);
iconCluster.infoWindowAnchor = new GPoint(13, 1);
iconCluster.infoShadowAnchor = new GPoint(26, 13);

//create an icon for the pins
var iconSingle = new GIcon();
iconSingle.image = "http://googlemapsbook.com/chapter7/icons/single.png";
iconSingle.shadow = "http://googlemapsbook.com/chapter7/icons/single_shadow.png";
iconSingle.iconSize = new GSize(12, 20);
iconSingle.shadowSize = new GSize(22, 20);
iconSingle.iconAnchor = new GPoint(6, 20);
iconSingle.infoWindowAnchor = new GPoint(6, 1);
iconSingle.infoShadowAnchor = new GPoint(13, 13);


function init() {
    map = new GMap2(document.getElementById("map"));
    map.addControl(new GSmallMapControl());
    map.setCenter(new GLatLng(centerLatitude, centerLongitude), startZoom);

    updateMarkers();

    GEvent.addListener(map,'zoomend',function() {
        updateMarkers();
    });

    GEvent.addListener(map,'moveend',function() {
```

```
        updateMarkers();
    });

}

function updateMarkers() {

    //remove the existing points
    map.clearOverlays();

    //mark the boundary of the data
    var allsw = new GLatLng(41.57025176609894, -73.39965820312499);
    var allne = new GLatLng(42.589488572714245, -71.751708984375);
    var allmapBounds = new GLatLngBounds(allsw,allne);
    map.addOverlay(
        new Rectangle(
            allmapBounds,
            4,
            '#F00',
            '<strong>Data Bounds, Zoom in for detail.</strong>'
        )
    );


    //get the bounds of the viewable area
    var mapBounds = map.getBounds();
    var sw = mapBounds.getSouthWest();
    var ne = mapBounds.getNorthEast();
    var size = mapBounds.toSpan(); //returns GLatLng

    //make a grid that's 10x10 in the viewable area
    var gridSize = 10;
    var gridCellSizeLat = size.lat()/gridSize;
    var gridCellSizeLng = size.lng()/gridSize;
    var gridCells = [];

    //loop through the points and assign each one to a grid cell
    for (k=0;k<points.length;k++) {
        var latlng = new GLatLng(points[k].latitude,points[k].longitude);

        //check if it is in the viewable area,
        //it may not be when zoomed in close
        if(!mapBounds.contains(latlng)) continue;

        //find grid cell it is in:
        var testBounds = new GLatLngBounds(sw,latlng);
        var testSize = testBounds.toSpan();
        var i = Math.ceil(testSize.lat()/gridCellSizeLat);
```

```
            var j = Math.ceil(testSize.lng()/gridCellSizeLng);
            var cell = i+j;

            if( typeof gridCells[cell] == 'undefined') {
                //add it to the grid cell array
                var cellSW = new GLatLng(
                    sw.lat()+((i-1)*gridCellSizeLat),
                    sw.lng()+((j-1)*gridCellSizeLng)
                );
                var cellNE = new GLatLng(
                    cellSW.lat()+gridCellSizeLat,
                   cellSW.lng()+gridCellSizeLng
                );
                gridCells[cell] = {
                    GLatLngBounds : new GLatLngBounds(cellSW,cellNE),
                    cluster : false,
                    markers:[],
                    length:0
                };

                //mark cell for testing
                map.addOverlay(
                    new Rectangle(
                        gridCells[cell].GLatLngBounds,
                        1,
                        '#00F',
                        '<strong>Grid Cell</strong>'
                    )
                );

            }

            gridCells[cell].length++;

            //already in cluster mode
            if(gridCells[cell].cluster) continue;

            //only cluster if it has more than 2 points
            if(gridCells[cell].markers.length==3) {
                gridCells[cell].markers=null;
                gridCells[cell].cluster=true;
            } else {
                gridCells[cell].markers.push(latlng);
            }

        }

        for (k in gridCells) {
```

```
        if(gridCells[k].cluster == true) {
            //create a cluster marker in the center of the grid cell
            var span = gridCells[k].GLatLngBounds.toSpan();
            var sw = gridCells[k].GLatLngBounds.getSouthWest();
            var marker = createMarker(
                new GLatLng(sw.lat()+(span.lat()/2),
                sw.lng()+(span.lng()/2))
                ,'c'
            );
            map.addOverlay(marker);
        } else {
            //create the single markers
            for(i in gridCells[k].markers) {
                var marker = createMarker(gridCells[k].markers[i],'p');
                map.addOverlay(marker);
            }
        }
    }
}

function createMarker(point, type) {
     //create the marker with the appropriate icon
     if(type=='c') {
         var marker = new GMarker(point,iconCluster,true);
     } else {
         var marker = new GMarker(point,iconSingle,true);
     }
    return marker;
}

window.onload = init;


/*
* Rectangle overlay for development only to mark boundaries for testing...
*/
function Rectangle(bounds, opt_weight, opt_color, opt_html) {
  this.bounds_ = bounds; this.weight_ = opt_weight || 1;
  this.html_ = opt_html || ""; this.color_ = opt_color || "#888888";
}
Rectangle.prototype = new GOverlay();

Rectangle.prototype.initialize = function(map) {
  var div = document.createElement("div");
  div.innerHTML = this.html_;
  div.style.border = this.weight_ + "px solid " + this.color_;
  div.style.position = "absolute";
  map.getPane(G_MAP_MAP_PANE).appendChild(div);
```

```
  this.map_ = map;
  this.div_ = div;
}
Rectangle.prototype.remove = function() {
  this.div_.parentNode.removeChild(this.div_);
}
Rectangle.prototype.copy = function() {
  return new Rectangle(
    this.bounds_,
    this.weight_,
    this.color_,
    this.backgroundColor_,
    this.opacity_
  );
}
Rectangle.prototype.redraw = function(force) {
  if (!force) return;
  var c1 = this.map_.fromLatLngToDivPixel(this.bounds_.getSouthWest());
  var c2 = this.map_.fromLatLngToDivPixel(this.bounds_.getNorthEast());
  this.div_.style.width = Math.abs(c2.x - c1.x) + "px";
  this.div_.style.height = Math.abs(c2.y - c1.y) + "px";
  this.div_.style.left = (Math.min(c2.x, c1.x) - this.weight_) + "px";
  this.div_.style.top = (Math.min(c2.y, c1.y) - this.weight_) + "px";
}
```

If you modify the grid cells over several loops, the browser may assume that the script is taking too long and display a warning, as shown in Figure 7-10.

Figure 7-10. *A JavaScript warning in Firefox indicating the script is taking too long to execute*

Further Client-Side Optimizations

Once you have your server and JavaScript optimized for your data set, you may also want to consider some additional niceties.

Removing Load Flashing

With the examples we've presented so far, you may have noticed that your maps "flash" between redraws and requests. This occurs because the JavaScript removes all the points and then draws

them all again. If you don't move the map a considerable distance, some points that are removed are then immediately replaced again. To avoid this, you can create a secondary JavaScript object to "remember" which points are currently on the map and remove only those that aren't in the new list. Using the same object, you can also add only those that aren't in the old list. Listing 7-19 (`http://book.earthcode.com/chap_seven/tracking_points`) shows the client-side boundary method from Listing 7-16 modified to keep track of points to remove the flashing between redraws.

Listing 7-19. *Modified Client-Side Boundary JavaScript That Remembers Which Markers Are on the Map*

```
var map;
var map;
var centerLatitude = 42.326062;
var centerLongitude = -72.290039;
var startZoom = 4;

var existingMarkers = {};

function init() {
    map = new GMap2(document.getElementById("map"));
    map.addControl(new GSmallMapControl());
    map.setCenter(new GLatLng(centerLatitude, centerLongitude), startZoom);

    updateMarkers();

    GEvent.addListener(map,'zoomend',function() {
        updateMarkers();
    });
    GEvent.addListener(map,'moveend',function() {
        updateMarkers();
    });
}

function updateMarkers() {
    //don't remove all the overlays!

    //map.clearOverlays();
    var mapBounds = map.getBounds();

    //loop through each of the points in memory and remove those that
    //aren't going to be shown
    for(k in existingMarkers) {
        if(!mapBounds.contains(existingMarkers[k].getPoint())) {
            map.removeOverlay(existingMarkers[k]);
            delete existingMarkers[k];
        }
    }
```

```
    //loop through each of the points from the global points object
    //and create markers that don't exist
    for (k=0; k<points.length; k++) {
         var latlng = new GLatLng(points[k].latitude,points[k].longitude);

        //skip it if the marker already exists
        //or is not in the viewable area
        if(!existingMarkers[k] && mapBounds.contains(latlng)) {
            existingMarkers[k] = createMarker(latlng);
            map.addOverlay(existingMarkers[k]);
        }
    }
}

function createMarker(point) {
    var marker = new GMarker(point);
    return marker;
}

window.onload = init;
```

You can apply the same fix for both server-side and client-side optimizations where the JavaScript is responsible for creating the markers.

Planning for the Next Move

If you want to be really nice and provide the ultimate user experience, you can put a little intelligence into your map and have it anticipate what the users are going to do next. From watching map users in test groups, it's our experience that most users "drag" the map in very small increments as they move around. The dragging movement of the map generally reveals only another 25% to 50% of that map in the direction opposite the drag.

Though you may assume your users will grab the map and drag in large sweeping motions (which they still could), smaller motions offer you an advantage. You can keep track of each movement and anticipate that the next movement will take the map in generally the same direction. If you know where the users are going to go, you can request the new points for that direction and have them already waiting before they get there.

Additionally, you could also extend the requested bounds beyond the edge of the viewport to include what's just outside the edge. By extending the boundary a bit outside the viewport, your users would think the map is loading faster, as markers are appearing quickly around the edge.

Summary

In this chapter, we presented a few optimization methods, for both your server and the browser, which allow your web application to run smoothly. By combining methods such as clustering and closest-to-point searches, you can further improve and create new optimization methods that will present your data in easy-to-understand and creative ways.

While working on your projects, be sure to choose the best method for the task at hand and don't base your decision on *coolness* alone. Creating your own tiles, as in the custom tile method described in this chapter, is pretty neat, but doesn't serve well for data that is generated from filtered searches, since each tile will always be different. Also, when using a feature such as clustering, make sure that your icons and user interface indicate this to the user.

Once you have your web application working, be sure to go over it again and look for places that could benefit from further optimization. Check again for areas where you could reduce the amount of data transferred between the client and the server, or check places where you're looping through large amounts of data and see if you can reduce it further. Just because your web application works doesn't mean it's working as well as it could. The better optimized your map, the happier your users will be and the better experience they'll have.

At the same time you're improving your web application and optimizing it to the best of your ability, Google will continue to develop its Maps API, adding improvements and new features. In the next chapter, you'll see some of the possible things Google *may* add—but no guarantees!

CHAPTER 8

What's Next for the Google Maps API?

As this book goes to press, the Google Maps API is still very much in development; its feature set continues to change and improve. As the API increases in popularity and new methods are added, it's often necessary to alter the way things work to enable new capabilities or provide more consistency throughout the API as a whole. Version 2, for example, splits the `GPoint` class into separate `GPoint` and `GLatLng` classes, each with enhanced capabilities corresponding to their respective roles in handling pixel coordinates and geographical locations. In reversing the zoom levels, which may have been an annoyance to developers, Google allows the maps to support as many detail levels as the satellite photography (or your custom overlay) warrants.

So far, we've shown you a lot of really neat techniques and tricks for getting data into your application and onto a map. In the following chapters, we expand on that and show you some powerful tools for making complex projects. But before we dive deeper into the API, we want to mention a few things you may want to keep a lookout for as the API continues to mature. None of these things are guarantees, but they're likely possibilities, given the demand and interest in them. In this chapter, we speculate on some of the things that Google *may* add to the API:

- Driving directions
- Integrated Google services
- KML data support
- More data layers
- A "map in a box" appliance solution
- Interface improvements

Driving Directions

If you follow the Google Maps discussion group at `http://groups.google.com/group/Google-Maps-API`, which we highly recommend you do, you'll notice a growing interest in the routing system built into `http://maps.google.com`, as shown in Figure 8-1.

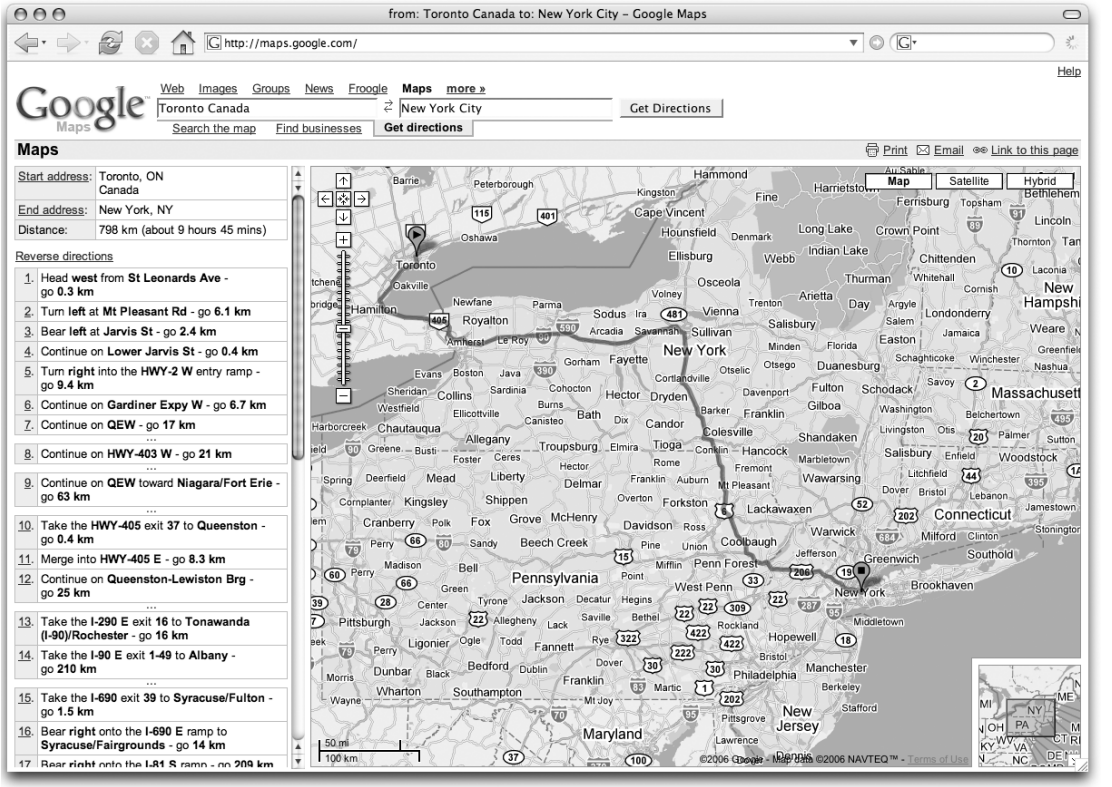


Figure 8-1. *Google Maps with a route from Toronto to New York*

Similar to the recently released geocoding service, Google could add an additional class that would allow you to retrieve the route information between arbitrary points on your map. This seems even more likely now that Google is also offering an Enterprise edition of the Maps API (`http://www.google.com/enterprise/maps/`) for use in closed corporate environments. Franchises and large chains of stores or restaurants could benefit from the inclusion of routing features to service their customers and delivery personnel.

Routing is an interesting can of worms, since it begins to expose more of Google's internal road database. But road information is not a secret, of course; if you want it, you can get it from freely available sources such as the U.S. Census Bureau's TIGER/Line files, as you will see in Chapter 11. The concern would be more with the immense computational power necessary to serve up complicated road queries in high volume, particularly to amateur API developers, who may not understand throttling or caching.

Integrated Google Services

As you've seen in Chapter 4, searching manually for data to plot and geocoding all the information yourself can be time-consuming and costly. However, vast stores of information are already available, hidden away in Google's search and service databases.

Google already offers its own business listing map web application at `http://maps.google.com`, where you can search for businesses based on their geographical locations, as shown in Figure 8-2.

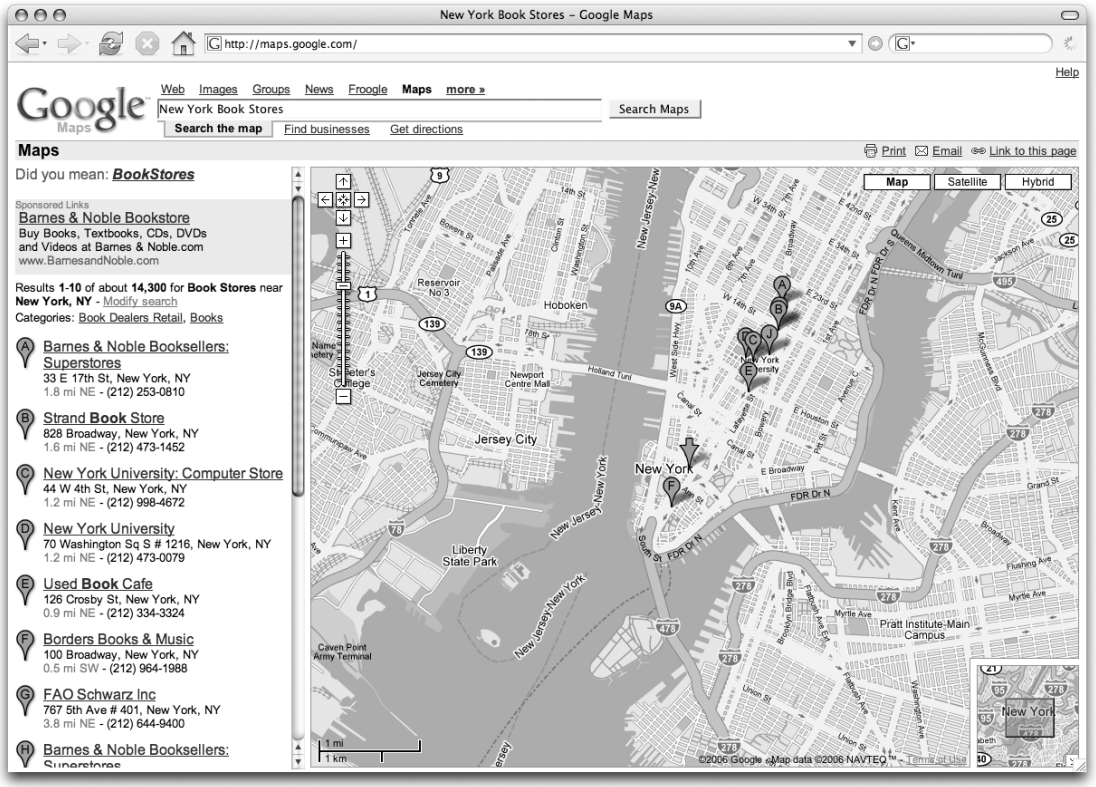


Figure 8-2. *Google Maps search for "New York bookstores"*

If Google chose to integrate its search database into the Google Maps API, Google's servers could provide you with ready-to-use mapping information based on search terms. This would relieve you of some parsing and geocoding tasks and eliminate the burden of collecting the information for your web application.

Imagine creating a map of bookstores in New York by asking the API for "bookstores in New York." The possibility of supplementing your map's proprietary data with Google's public data is certainly an intriguing one. As the owner of a chain of bookstores, you could not only help your customers locate your stores, but you could also offer added value by throwing up the results of a "Coffee shops within one mile of `StoreLatLng`" query.

Tip Though not built into the Google Maps API, Google's search database can be used now by combining some additional Google APIs such as the Google Ajax Search API and maps. For an example, check out the My Favorite Places page at `http://www.google.com/uds/samples/places.html`, where you can type in a request such as **New York bookstores** and get mapping information.

KML Data

As you saw in Chapter 1, the `http://maps.google.com` site lets you plot any arbitrary KML data directly on your map. In that chapter, we showed you a quick sample file that marked three popular destinations in downtown Toronto. Figure 8-3 shows a similar file, which drops an arbitrary point onto southeastern Ontario.

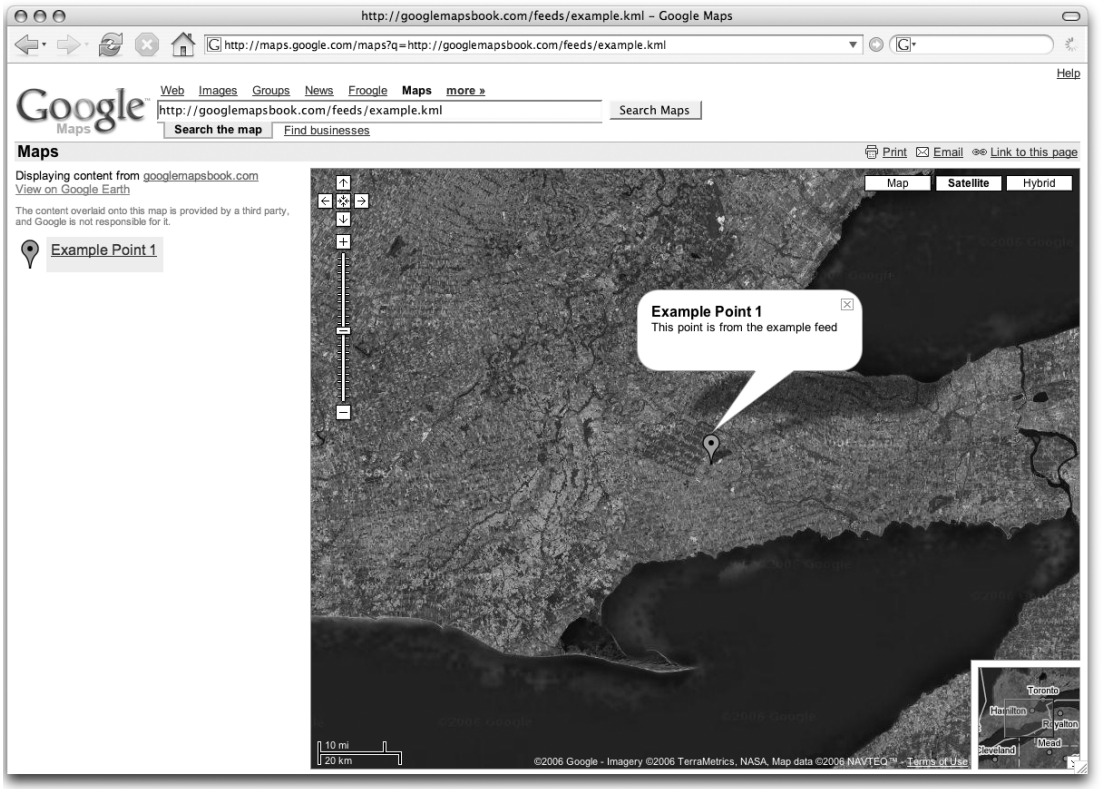


Figure 8-3. *Sample KML file in a map*

At the moment, using KML data is possible only with Google Maps itself, not directly from the API. But it certainly appears that Google has reason to expand interest in the KML data format. We expect future versions of the API to provide shortcut functions for loading and parsing this kind of information. You can do it yourself, of course, but to automate it would help bridge the gap between users of Google Maps and users of the Maps API.

More Data Layers

The satellite imagery included in the API has opened the whole world to people who may never even travel out of their hometowns. With a simple click and drag of the mouse, sites such as `http://googlesightseeing.com` (Figure 8-4) can take you anywhere on the planet, and in many cases give you a close enough look to make out cars and people.

Figure 8-4. *The Google Sightseeing home page*

So if Google can offer two layers of data (satellite and map), then why shouldn't we expect that it will begin to offer other complementary layers? The data for things such as elevation, weather trends, and population density are all available and would make excellent layers in the system. While this may tread on some of the maps we are building, it could also open up new opportunities, just as the satellite imagery did for sightseeing.

Also, Google Earth, Google's desktop mapping software, already allows you to incorporate Google SketchUp objects, so why not make these objects available to the Google Maps API, too?

Beyond the Enterprise

In building new relationships with enterprise providers, Google is edging into the corporate mapping space previously dominated by desktop products such as Microsoft MapPoint. When enterprise clients begin to require even greater performance and feature diversity, Google may provide a Google Maps Mini appliance similar to the Google Mini search appliance offered today (`http://www.google.com/enterprise/mini/`). A Mini appliance would provide the corporate world with a "map in a box" solution that could be highly customized and branded to offer features that support the needs of specific companies and markets.

Those of us using the free mapping API may also one day see integrated advertisements in our maps. The terms of service have always provided for the eventuality of Google adding means to make money from your map. Paying enterprise customers would certainly be exempt from any integrated advertising, which would offer the rest of us a compelling reason to upgrade to the enterprise subscription.

■ **Note** The API key signup page explicitly states that Google will give developers 90 days notice via the official Google Maps API blog (`http://googlemapsapi.blogspot.com`) before introducing advertising into third-party sites such as those you're building. If the prospect of advertising bothers you, we suggest that you follow this blog closely.

Interface Improvements

The current Google Maps interface is built entirely using XHTML, CSS, and JavaScript. It works extremely well, but is limited by the browser's ability to quickly scale images or move around large numbers of onscreen objects. Other mapping tools such as the Yahoo Maps API offer alternative Flash clients that can benefit from the performance optimizations of that system. Though Google doesn't offer a Flash-based API, others have attempted to incorporate the Google Maps API with Flash and created unique, highly interactive, and rich web applications. Figure 8-5 shows one example: the X-Men map at `http://xplanet.net`.

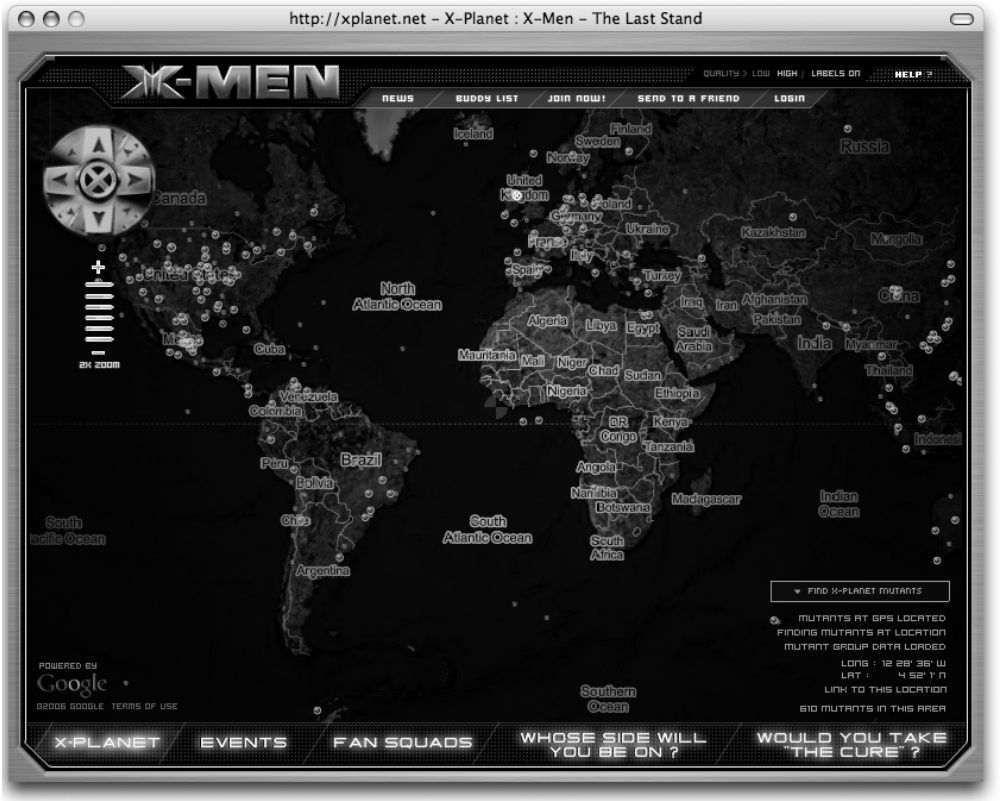


Figure 8-5. *The X-Men Flash-based Google map*

With the growing competition from Yahoo Maps and Windows Live Local (Microsoft's competing map service), Google may come to offer additional options such as a Flash API. Perhaps Google will develop an even more sophisticated interface based on Scalable Vector Graphics (SVG) or some other technology that can bring the browser experience closer to that of Google Earth.

A more incremental interface improvement (and one that seems likely to be incorporated soon) is the integration of the mouse scroll wheel into the Google Maps API. The Google Maps web site allows users to zoom using their mouse scroll wheels; it makes sense that Google would incorporate this feature into the API as well.

Summary

In this chapter, we speculated about what might be coming up in the Google API. Along with the new services, we can expect better tools. As with any web application, Google will be continually improving on the existing components of the Maps API. Tools like the geocoder will eventually expand to cover more countries and improve accuracy as more detailed information becomes available. Satellite imagery will increase in detail and will be updated continually with more and more recent images.

Now we are ready to move on to some more advanced mapping techniques. In the next part of the book, we'll cover a wide variety of complementary concepts for your mapping projects. Chapter 9 demonstrates how to make your own info windows and tool tips, as well as other overlay-related tricks. In Chapter 10, we cover some mathematics you may need in a professional map. Finally, in Chapter 11, we show you how to build your own geocoder from scratch, using a raw data set.

PART 3

Advanced Map Features and Methods

CHAPTER 9

Advanced Tips and Tricks

Beyond what you've seen so far, the Google Maps API has a number of features that are often overlooked. Here, you'll go through a variety of examples to learn how to use some of the more advanced features of the API, such as the ability to change map tiles and the possibility of creating your own overlay objects.

In this chapter, the examples demonstrate how to do the following:

- Create an overlay for markers that act as a tool tip
- Promote yourself with a custom icon control
- Add tabs to info windows
- Construct your own info window
- Create your own map tiles using the NASA Blue Marble images

Debugging Maps

Before diving into the examples, let's take a quick look at debugging within the Google Maps API. In the Google Maps API version 1, the debugger's best friend was `alert()`. With Google Maps API version 2, you now have access to the simple, yet very useful, `GLog` class. Now `GLog.write()` is the "new" `alert()`, but it creates a floating log window, as shown in Figure 9-1, to hold all your debugging messages.

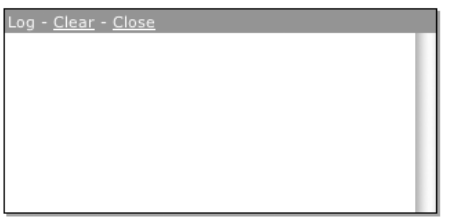


Figure 9-1. *Empty GLog window*

For example, if you're curious about what methods and properties a JavaScript object has, such as the `GMap2` object, try this:

```
var map = new GMap2(document.getElementById("map"));
for(i in map) { GLog.write(i); }
```

Voilà! The `GLog` window in Figure 9-2 now contains a scrolling list of all the methods and properties belonging to your `GMap2` object, and you didn't need to click OK in dozens of alert windows to get to it.

Figure 9-2. *GLog window listing methods and properties of the GMap2 object*

The `GLog.write()` method escapes any HTML and logs it to the window as source code. If you want to output *formatted* HTML, you can use the `GLog.writeHtml()` method. Similarly, to output a clickable link, just pass a URL into the `GLog.writeUrl()` method. The `writeUrl()` method is especially useful when creating your own map tiles, as you'll see in the "Implementing Your Own Map Type, Tiles, and Projection" section later in the chapter, where you can simply log the URL and click the link to go directly to an image for testing.

■ **Tip** `GLog` isn't bound to just map objects; it can be used throughout your web application to debug any JavaScript code you want. As long as the Google Maps API is included in your page, you can use `GLog` to help debug anything from Ajax requests to mouse events.

Interacting with the Map from the API

When building your web applications using Google Maps, you'll probably have more in your application than just the map. What's outside the map will vary depending on the purpose of your project and could include anything from graphical eye candy to interactive form elements. When these external elements interact with the map, especially when using the mouse, you may often find yourself struggling to locate the pixel position of the various map objects on your screen. You may also run into situations where you need to trigger events, even mouse-related events, without the cursor ever touching the element. In these situations, a few classes and methods may come in handy.

Helping You Find Your Place

More and more, your web applications will be interacting with users in detailed and intricate ways. Gone are the days of simple requests and responses, where the cursor was merely used to navigate from box to box on a single form. Today, your web application may rely on drag-and-drop, sliders, and other mouse movements to create a more desktoplike environment. To help you keep track of the position of objects on the map and on the screen, Google has provided coordinate transformation methods that allow you to convert a longitude and latitude into X and Y screen coordinates and vice versa.

To find the pixel coordinates of a location on the map relative to the map's `div` container, you can use the `GMap2.fromLatLngToDivPixel()` method. By converting the latitude and longitude into a pixel location, you can then use the pixel location to help position other elements of your web application relative to the map objects. Take a quick look at Listing 9-1, where the `mousemove` event is used to log the pixel location of the cursor on the map.

Listing 9-1. *Tracking the Mouse on the Map*

```
var map;
var centerLatitude = 43.49462;
var centerLongitude = -80.548239;
var startZoom = 3;

function init() {

    map = new GMap2(document.getElementById("map"));
    map.addControl(new GSmallMapControl());
    map.addControl(new GMapTypeControl());
    map.setCenter(new GLatLng(centerLatitude, centerLongitude), startZoom);

    GEvent.addListener(map,'mousemove',function(latlng) {
        var pixelLocation = map.fromLatLngToDivPixel(latlng);
        GLog.write('ll:' + latlng + ' at:' + pixelLocation);
    });
}
window.onload = init;
```

Moving around the map, the `GLog` window reveals the latitude and longitude location of the cursor, along with the pixel location relative to the top left corner of the map `div`, as shown in Figure 9-3.

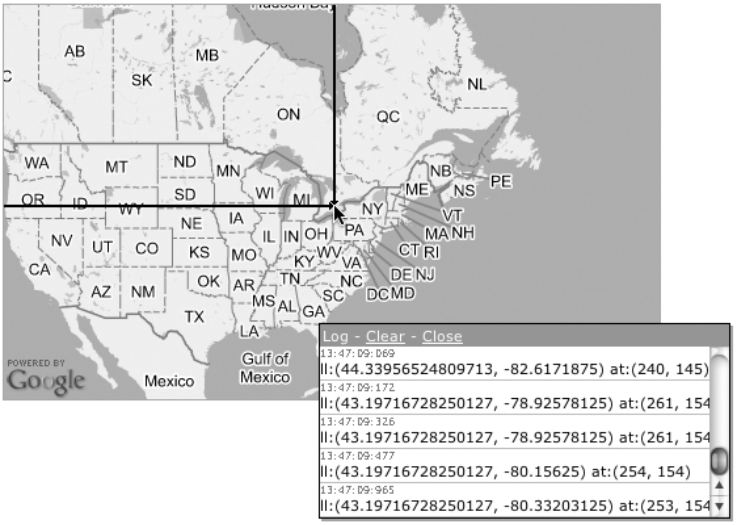


Figure 9-3. *Tracking the mouse movement relative to the map container*

Once you have the pixel location from `GMap2.fromLatLngToDivPixel()`, you can turn it into a location relative to the screen or window by applying additional calculations appropriate to the design and layout of your web application.

■ **Tip** For more information about JavaScript and using it to interact with your web page, pick up *DOM Scripting: Web Design with JavaScript and the Document Object Model* by Jeremy Keith (`http://www.friendsofed.com/book.html?isbn=1590595335`). It covers everything you need to know when using JavaScript to add dynamic enhancements to web pages and program Ajax-style applications.

Force Triggering Events with GEvent

The `GEvent` object (which we first utilized in Chapter 2) lets you run code when specific events are triggered on particular objects. You can attach events to markers, the map, DOM objects, info windows, overlays, and any other object on your map. In earlier chapters, you used the `click` event to create markers, and the `zoomend` event to load data from the server. These work great if your users are interacting with the map, but what happens if they're interacting with some *other* part of the web application and you want those objects to trigger these events? In those cases, you can use the `trigger()` method of the `GEvent` class to force the event to run.

For example, suppose you create an event that runs when the zoom level is changed on your map using the `zoomend` event, and it's logged to the `GLog` window:

```
GEvent.addListener(map,'zoomend',function(oldLevel, newLevel) {
    //some other code
    GLog.write('Zoom changed from ' + oldLevel + ' to ' + newLevel);
});
```

In this example, the event handler (the function that gets invoked when the event happens) is `function(oldLevel, newLevel){...}`. The event passes two arguments to the event handler, `oldLevel` and `newLevel`. If you adjust the zoom level of your map, you'll get a log entry that looks something like Figure 9-4.

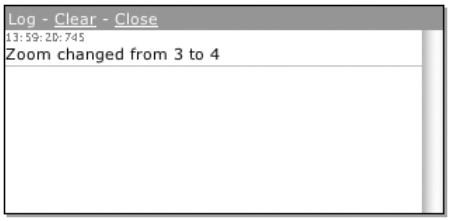


Figure 9-4. *GLog entry after changing zoom levels using the zoom control*

Figure 9-4 shows the message that is logged when the user changes the zoom level from 3 to 4. The event listener passes the event handler two arguments: `oldLevel`, with a value of `3`, and `newLevel`, with a value of `4`. From elsewhere in your web application, you can force the `zoomend` event to execute by calling this:

```
GEvent.trigger(map,'zoomend');
```

Executing this method will cause the `zoomend` event to run as normal. There is a problem here, however: there are no arguments provided to the event handler. Therefore, you'll see `undefined` values for both `oldLevel` and `newLevel` in the log, as shown in Figure 9-5.

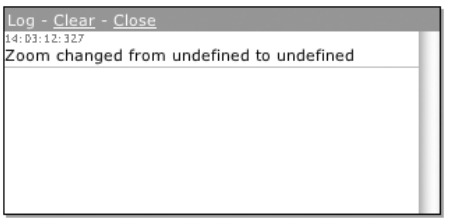


Figure 9-5. *GLog entries after triggering zoomend by calling GEvent.trigger(map, 'zoomend')*

The same applies for any event that passes arguments into its trigger function. If the API can't determine what to pass, you'll get an `undefined` value.

To overcome this problem, you can pass additional arguments after the `trigger()` `event` argument, and they'll be passed as the arguments to the event handler function. For example, calling

```
GEvent.trigger(map,'zoomend',3,5);
```

would pass `3` as the `oldLevel` and `5` as the `newLevel`. But unless you change the zoom level of the map some other way, the zoom level won't actually change, since you've manually forced the `zoomend` event without calling any of the zoom-related methods of the map.

Creating Your Own Events

Along with triggering the existing events from the API, `GEvent.trigger()` can also be used to trigger your own events. For example, you could create an `updateMessage` event to trigger a script to execute when a message box is updated, as follows:

```
var message = document.getElementById('messageBox');
GEvent.addDomListener(message,'updateMessage',function() {
    //whatever code you want
    if(message.innerHtml != '') alert('The system reported messages.');
});
```

Then, elsewhere in your application, you can update the message and trigger the `updateMessage` event using the `GEvent.trigger()`method:

```
var message = document.getElementById('messageBox');
if (error) {
    message.innerHtml = 'There was an error with the script.';
} else {
    message.innerHtml = '';
}
GEvent.trigger(message,'updateMessage');
```

Creating Map Objects with GOverlay

In Chapter 7, you saw how to use `GOverlay` to create an image that could hover over a location on a map to show more detail. In that instance, the overlay consisted of a simple HTML `div` element with a background image, similar to the `Rectangle` example in the Google Maps API documentation (`http://www.google.com/apis/maps/documentation/#Custom_Overlays`). Beyond just a simple `div`, the overlay can contain any HTML you want and therefore can include anything you could create in a web page. Even Google's info window is really just a fancy overlay, so you could create your own overlay with whatever features you want.

Caution Adding your own overlays will influence the limitations of the map the same way the markers do in Chapter 7. In fact, your overlays will probably be much more influential, as they will be more complicated and weighty than the simpler marker overlay.

Choosing the Pane for the Overlay

Before you create your overlay, you should familiarize yourself with the `GMapPane` constants. `GMapPane` is a group of constants that define the various layers of the Google map, as represented in Figure 9-6.

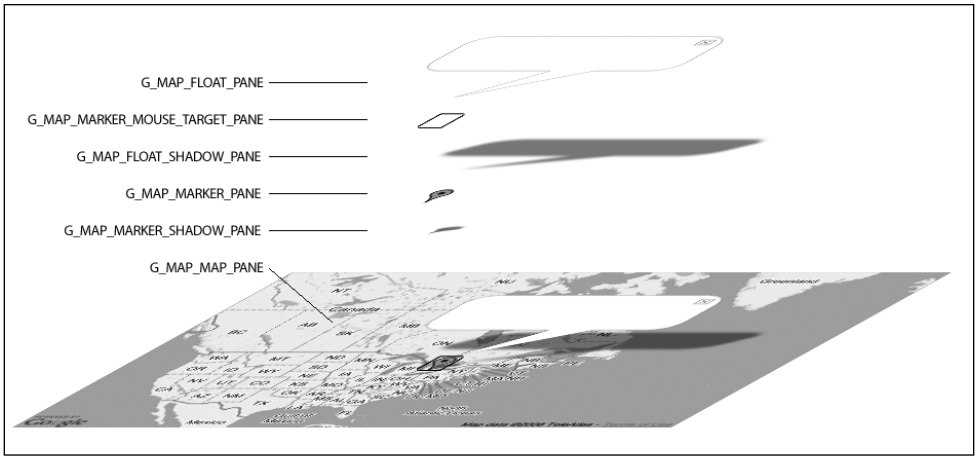


Figure 9-6. *GMapPane constants layering*

At the lowest level, flat against the map tiles, lies the `G_MAP_MAP_PANE`. This pane is used to hold objects that are directly on top of the map, such as polylines. Next up are the `G_MAP_MARKER_SHADOW_PANE` and `G_MAP_MARKER_PANE`. As the names suggest, they hold the shadows and icons for each of the `GMarker` objects on the map. The shadow and icon layers are separated, so the shadows don't fall on top of the icons when markers are clustered tightly together.

The next layer above that is the `G_MAP_FLOAT_SHADOW_PANE`, which is where the shadow of the info window will reside. This pane is above the markers, so the shadow of the info window will be cast over the markers on the map.

The next layer, `G_MAP_MARKER_MOUSE_TARGET_PANE`, is an ingenious trick. The mouse events for markers are not actually attached to the markers on the marker pane. An invisible object, hovering in the mouse target pane, captures the events, allowing clicks to be registered on the markers hidden in the shadow of the info window. Without this separate mouse target pane, clicks on the covered markers wouldn't register, as the info window's shadow would cover the markers, and in most browsers, only the top object can be clicked.

Finally, on top of everything else, is the `G_MAP_FLOAT_PANE`. The float pane is the topmost pane and is used to hold things such as the info window or any other overlays you want to appear on top.

When you create your overlay object, you need to decide which of the six panes is best suited. If your overlay has a shadow, like the custom info window presented later in Listing 9-5, you'll need to target two panes.

To retrieve and target the DOM object for each pane, you can use the `GMap2.getPane()` method. For example, to add a `div` tag to the float pane, you would do something similar to this:

```
div = document.createElement('div');
pane = map.getPane(G_MAP_FLOAT_PANE);
pane.appendChild(div);
```

Obviously, your code surrounding this would be a little more involved, but you get the idea.

Creating a Quick Tool Tip Overlay

For an easy `GOverlay` example, let's create an overlay for markers that acts as a tool tip, containing just a single line of text in a colored box, as shown in Figure 9-7.

Figure 9-7. *Tool tip overlay*

Listing 9-2 shows the code for the tool tip overlay.

Listing 9-2. *ToolTip Overlay Object*

```
//create the ToolTip overlay object
function ToolTip(marker,html,width) {
    this.html_ = html;
    this.width_ = (width ? width + 'px' : 'auto');
    this.marker_ = marker;
}

ToolTip.prototype = new GOverlay();

ToolTip.prototype.initialize = function(map) {
    var div = document.createElement("div");
    div.style.display = 'none';
    map.getPane(G_MAP_FLOAT_PANE).appendChild(div);
    this.map_ = map;
    this.container_ = div;
}

ToolTip.prototype.remove = function() {
    this.container_.parentNode.removeChild(this.container_);
}
```

```
ToolTip.prototype.copy = function() {
    return new ToolTip(this.html_);
}

ToolTip.prototype.redraw = function(force) {
    if (!force) return;
    var pixelLocation = this.map_.fromLatLngToDivPixel(this.marker_.getPoint());
    this.container_.innerHTML = this.html_;
    this.container_.style.position = 'absolute';
    this.container_.style.left = pixelLocation.x + "px";
    this.container_.style.top = pixelLocation.y + "px";
    this.container_.style.width = this.width_;
    this.container_.style.font = 'bold 10px/10px verdana, arial, sans';
    this.container_.style.border = '1px solid black';
    this.container_.style.background = 'yellow';
    this.container_.style.padding = '4px';

    //one line to desired width
    this.container_.style.whiteSpace = 'nowrap';
    if(this.width_ != 'auto') this.container_.style.overflow = 'hidden';

    this.container_.style.display = 'block';
}

GMarker.prototype.ToolTipInstance = null;
GMarker.prototype.openToolTip = function(content) {
    //don't show the tool tip if there is a custom info window
    if(this.ToolTipInstance == null) {
        this.ToolTipInstance = new ToolTip(this,content)
        map.addOverlay(this.ToolTipInstance);
    }
}
GMarker.prototype.closeToolTip = function() {
    if(this.ToolTipInstance != null) {
        map.removeOverlay(this.ToolTipInstance);
        this.ToolTipInstance = null;
    }
}
```

Now let's see how it works.

Creating the GOverlay Object

To create the `ToolTip` `GOverlay` in Listing 9-2, start by writing a function with the name you would like to use for your overlay and pass in any parameters you would like to include. For example, the arguments for the `ToolTip` overlay constructor in Listing 9-2 are the marker to attach the tool tip to and the HTML to display in the tool tip. For more control, there's also an optional `width` argument to force the tool tip to a certain size:

```
function ToolTip(marker,html,width) {
    this.html_ = html;
    this.width_ = (width ? width + 'px' : 'auto');
    this.marker_ = marker;
}
```

This function, `ToolTip`, will act as the constructor for your `ToolTip` class. Once finished, you would instantiate the object by creating a new instance of the `ToolTip` class:

```
var tip = new ToolTip(marker,'This is a marker');
```

When assigning properties to the class, such as `html`, it's always good to distinguish the internal properties using something like an underscore, such as `this.html_`. This makes it easy to recognize internal properties and also ensure that you don't accidentally overwrite a property of the `GOverlay` class if Google has used `html` as a property for the `GOverlay` class.

Next, instantiate the `GOverlay` as the prototype for your new `ToolTip` function:

```
ToolTip.prototype = new GOverlay();
```

Creating and Positioning the Container

For the guts of your `ToolTip` class, you need to prototype the four required methods listed in Table 9-1.

Table 9-1. *Abstract Methods of the GOverlay Object*

Method	Description
`initialize()`	Called by `GMap2.addOverlay()` when the overlay is added to the map
`redraw(force)`	Executed once when the object is initially created and then again whenever the map display changes; `force` will be `true` in the event the API recalculates the coordinates of the map
`remove()`	Called when `removeOverlay()` methods are used
`copy()`	Should return an uninitialized copy of the same object

First, start by prototyping the `initialize()` function:

```
ToolTip.prototype.initialize = function(map) {
    var div = document.createElement("div");
    div.style.display='none';
    map.getPane(G_MAP_FLOAT_PANE).appendChild(div);
    this.map_ = map;
    this.container_ = div;
}
```

The `initialize()` method is called by `GMap2.addOverlay()` when the overlay is initially added to the map. Use it to create the initial `div`, or other element, and to attach the `div` to the appropriate pane using `map.getPane()`. Also, you probably want to assign the `map` variable to an internal variable so you'll still have access to it from inside the other methods of the `ToolTip` object.

Next, prototype the `redraw()` method:

```
ToolTip.prototype.redraw = function(force) {
    if (!force) return;
    var pixelLocation = this.map_.fromLatLngToDivPixel(this.marker_.getPoint());
    this.container_.innerHTML = this.html_;
    this.container_.style.position='absolute';
    this.container_.style.left = pixelLocation.x + "px";
    this.container_.style.top = pixelLocation.y + "px";

    - cut -

    this.container_.style.display = 'block';
}
```

The `redraw()` method is executed once when the object is initially created and then again whenever the map display changes. The `force` flag will be `true` only in the event the API needs to recalculate the coordinates of the map, such as when the zoom level changes or the pixel offset of the map has changed. It's also `true` when the overlay is initially created so the object can be drawn. For your `ToolTip` object, the `redraw()` method should stylize the `container_ div` element and position it relative to the location of the marker. In the event that the `width` argument is provided, the `div` should also be defined accordingly, as it is in Listing 9-2.

Lastly, you should prototype the `copy()` and `remove()` methods:

```
ToolTip.prototype.remove = function() {
    this.container_.parentNode.removeChild(this.container_);
}

ToolTip.prototype.copy = function() {
    return new ToolTip(this.marker_,this.html_,this.width_);
}
```

The `copy()` method should return an uninitialized copy of the same object to the map. The `remove()` method should remove the existing object from the pane.

Using Your New Tool Tip Control

At the bottom of Listing 9-2 you'll also notice the addition of a few prototype methods on the `GMarker` class. These give you a nice API for your new `ToolTip` object by allowing you to call `GMarker.openToolTip('This is a marker')` to instantiate the tool tip; `GMarker.closeToolTip()` will close the tool tip.

Now you can create a marker and add a few event listeners, and you'll have a tool tip that shows on `mouseover`, similar to the one shown earlier in Figure 9-7:

```
var marker = new GMarker(new GLatLng(43, -80));

GEvent.addListener(marker,'mouseover',function() {
    marker.openToolTip('This is a GMarker!');
});
GEvent.addListener(marker,'mouseout',function() {
```

```
    marker.closeToolTip();
});
map.addOverlay(marker);
```

The `ToolTip` overlay is relatively simple but very useful. Later in the chapter (Listing 9-5), you'll revisit the `GOverlay` object when you create an overlay that's a little more complicated, to serve as your own customized info window.

Creating Custom Controls

Overlays are useful, but they generally apply to something on the map fixed to a latitude and longitude. When you drag the map, the overlays go with it. If you want to create a control or other object on the map that's fixed to a relative location within the map container, similar to the zoom control or the map type buttons, you'll need to implement a `GControl` interface.

Six controls are built into the Google Maps API, as you've seen throughout the book. The six controls are `GSmallMapControl`, `GLargeMapControl`, `GSmallZoomControl`, `GMapTypeControl`, `GScaleControl`, and `GOverviewMapControl`. Depending on your application and features, you can enable or disable the controls so your users can have varying degrees of control over the map.

If these controls don't suit your needs, you can implement a custom control that replicates the functionality of one of Google's existing controls, or create something completely different. For example, the Google Maps API documentation at `http://www.google.ca/apis/maps/documentation/#Custom_Controls` provides an example of a textual zoom control. The Google `TextualZoomControl` creates the text-based Zoom In and Zoom Out buttons shown in Figure 9-8 and is an alternative to the `GSmallMapControl`.

Figure 9-8. *The Google textual zoom control adds Zoom In and Zoom Out buttons.*

As an example, we'll show you how to create a custom icon control. After all the hard work you've poured into your web application, it might be nice to promote yourself a little and put your company logo down in the corner next to Google's. After all, a little promotion never hurt anyone. Implementing the icon control in Figure 9-9 is relatively simple, as you can see in Listing 9-3, and it's a great example you can further expand on.

Figure 9-9. *A promotional map control, clickable to a supplied link*

Listing 9-3. *Promotional Icon PromoControl*

```
var PromoControl = function(url) {
    this.url_ = url;
};

PromoControl.prototype = new GControl(true);

PromoControl.prototype.initialize = function(map) {
    var container = document.createElement("div");
    container.innerHTML = '<img style="cursor:pointer"➥
src="http://googlemapsbook.com/PromoApress.png" border="0">';
    container.style.width='120px';
    container.style.height='32px';
    url = this.url_;
    GEvent.addDomListener(container, "click", function() {
        document.location = url;
    });
    map.getContainer().appendChild(container);
    return container;
};

PromoControl.prototype.getDefaultPosition = function() {
    return new GControlPosition(G_ANCHOR_BOTTOM_LEFT, new GSize(70, 0));
};
```

The following sections describe how Listing 9-3 works.

Creating the Control Object

To create your promo `GControl` object, start the same way you did with the `GOverlay` in the previous example. Create a function with the appropriate name, but use the prototype object to instantiate the `GControl` class:

```
var PromoControl = function(url) {
    this.url_ = url;
};
PromoControl.prototype = new GControl(true);
```

By passing in a `url` parameter, your `PromoControl` can be clickable to the supplied `url` and you can reuse the `PromoControl` for different URLs, depending on your various mapping applications.

Creating the Container

Next, there are only two methods you need to prototype. First is the `initialize()` method, which is similar to the `initialize()` method from the `GOverlay` example:

```
PromoControl.prototype.initialize = function(map) {
    var container = document.createElement("div");
    container.innerHTML = '<img src="http://googlemapsbook.com/PromoApress.png"➥
border="0">';
    container.style.width='120px';
    container.style.height='32px';
    url = this.url_;
    GEvent.addDomListener(container, "click", function() {
        document.location = url;
    });
    map.getContainer().appendChild(container);
    return container;
};
```

The difference is the `GOverlay.initialize()` method will be called by the `GMap2.addControl()` method when you add the control to your map. In the case of `GControl`, the container `div` for the control is attached to the map's container DOM object returned from the `GMap2.getContainer()` method. Also, you can add events such as the `click` event to the `container` using the `GEvent.addDomListener()` method. For more advanced controls, you can include any HTML you want and apply multiple events to the various parts of the control. For the `PromoControl`, you're simply including an image that links to the supplied URL, so one click event can be attached to the entire container.

Positioning the Container

Last, you need to position the `PromoControl` within the map container by returning a new instance of the `GControlPostion` class from the `getDefaultPosition` prototype:

```
PromoControl.prototype.getDefaultPosition = function() {
    return new GControlPosition(G_ANCHOR_BOTTOM_LEFT, new GSize(70, 0));
};
```

The `GControlPosition` represents the anchor point and offset where the control should reside. To anchor the control to the map container, you can use one of four constants:

- `G_ANCHOR_TOP_RIGHT` to anchor to the top right corner
- `G_ANCHOR_TOP_LEFT` to anchor to the top left corner
- `G_ANCHOR_BOTTOM_RIGHT` to anchor to the bottom right corner
- `G_ANCHOR_BOTTOM_LEFT` to anchor to the bottom left corner

Once anchored, you can then offset the control by the desired distance. For the `PromoControl`, anchoring to just `G_ANCHOR_BOTTOM_LEFT` would interfere with the Google logo, thus going against the terms and conditions of the API. To fix this, you offset your control using a new `GSize` object with an X offset of 70 pixels, the width of the Google logo.

■ **Caution** If you plan on using the `GScaleControl` as well, remember that it too will occupy the space next to the Google logo, so you'll need to adjust your `PromoControl` accordingly.

Using the Control

With your `PromoControl` finished, you can add it to your map using the same `GMap2.addControl()` method and a new instance of your `PromoControl`:

```
map.addControl(new PromoControl('http://googlemapsbook.com'));
```

You'll end up with your logo positioned neatly next to the Google logo, linked to wherever you like, as shown earlier in Figure 9-9.

Adding Tabs to Info Windows

If you're happy with the look of the Google info window, or you don't have the time or budget to create your own info window overlay, there are a few new features of the Google Maps API version 2 info window that you may find useful. With version 1 of the Google Maps API, the info window was just the stylized bubble with a close box, as shown in Figure 9-10. You could add tabs, but the limit was two tabs, and doing so required hacks and methods that were not "official" parts of the API.

Figure 9-10. *The version 1 info window*

Creating a Tabbed Info Window

With version 2 of the API, Google has added many tab-related features to its info windows. You can have multiple tabs on each info window, as shown in Figure 9-11, and you can change the tabs from within the API using various `GInfoWindow` methods, as shown in Listing 9-4.

Figure 9-11. *A tabbed info window*

Listing 9-4. *Info Window with Three Tabs*

```
map = new GMap2(document.getElementById("map"));
map.addControl(new GSmallMapControl());
map.addControl(new GMapTypeControl());
map.setCenter(new GLatLng(centerLatitude, centerLongitude), startZoom);

marker = new GMarker(new GLatLng(centerLatitude, centerLongitude));
map.addOverlay(marker);

var infoTabs = [
    new GInfoWindowTab("Tab A", "This is tab A content"),
    new GInfoWindowTab("Tab B", "This is tab B content"),
    new GInfoWindowTab("Tab C", "This is tab C content")
];

marker.openInfoWindowTabsHtml(infoTabs,{
    selectedTab:1,
    maxWidth:300
});

GEvent.addListener(marker,'click',function() {
    marker.openInfoWindowTabsHtml(infoTabs);
});
```

To create the info window with three tabs in Figure 9-11, you simply create an array of `GInfoWindowTab` objects:

```
var infoTabs = [
    new GInfoWindowTab("Tab A", "This is tab A content"),
    new GInfoWindowTab("Tab B", "This is tab B content"),
    new GInfoWindowTab("Tab C", "This is tab C content")
];
```

Then use `GMarker.openInfoWindowTabsHtml()` to create the window right away:

```
marker.openInfoWindowTabsHtml(infoTabs,{
    selectedTab:1,
    maxWidth:300
});
```

Or it can be used inside of an event:

```
GEvent.addListener(marker,'click',function() {
    marker.openInfoWindowTabsHtml(infoTabs);
});
```

Additionally, you can define optional parameters for the tabbed info window the same way you can define options using the `GMarker.openInfoWindow` methods.

Gathering Info Window Information and Changing Tabs

If other parts of your web application need to interact with the various tabs on your info window, things get a little trickier. When the tabbed info window is created, the API instantiates the object for you, so you don't actually have direct access to the info window object yet. As you saw in Chapter 3, there is only one instance of an info window on a map at a time, so you can use the `GMap2.getInfoWindow()` method to retrieve a handle for the current info window:

```
var windowHandle = map.getInfoWindow();
```

With the handle, you can then use any of the `GInfoWindow` methods to retrieve information or perform various operations, such as the following:

- Retrieve the latitude and longitude of the window anchor:

```
windowHandle.getPoint();
```

- Hide the window:

```
windowHandle.hide();
```

- Switch to another tab:

```
windowHandle.selectTab(2);
```

For a full list of the `GInfoWindow` methods, see the API in Appendix B.

Creating a Custom Info Window

If you follow the Google Maps discussion group (`http://groups.google.com/group/Google-Maps-API`), you'll notice daily posts regarding feature requests for the info window. Feature requests are great, but most people don't realize the info window isn't really anything special. It's just another `GOverlay` with a lot of extra features. With a little JavaScript and `GOverlay`, you can create your very own info window with whatever features you want to integrate. To get you started, we'll show you how to create the new info window in Figure 9-12, which occupies a little less screen real estate, but offers you a starting point to add on your own features.

Figure 9-12. *A custom info window*

To begin, you'll need to open up your favorite graphics program and create the frame for the window. If you just need a box, then it's not much more difficult then the `ToolTip` object you created earlier. For this example, we used the Adobe Photoshop PSD file you'll find with the code accompanying this book, as illustrated in Figure 9-13. Once you have your info window working, feel free to modify it any way you want. You can edit the PSD file or create one of your own. For now, create a folder called `littleWindow` in your working directory and copy the accompanying presliced PNG files from the `littleWindow` folder in the Chapter 9 source code.

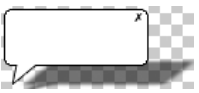


Figure 9-13. *The info window art file*

The finalized framework for the `LittleInfoWindow` overlay in Listing 9-5 is almost identical to the `ToolTip` overlay you created earlier in Listing 9-2, but the internals of each function are quite different.

Listing 9-5. *The LittleInfoWindow Object*

```
//create the LittleInfoWindow overlay onject
function LittleInfoWindow(marker,html,width) {
    this.html_ = html;
    this.width_ = ( width ? width + 'px' : 'auto');
    this.marker_ = marker;
}
```

```
//use the GOverlay class
LittleInfoWindow.prototype = new GOverlay();

//initialize the container and shadowContainer
LittleInfoWindow.prototype.initialize = function(map) {
    this.map_ = map;

    var container = document.createElement("div");
    container.style.display='none';
    map.getPane(G_MAP_FLOAT_PANE).appendChild(container);
    this.container_ = container;

    var shadowContainer = document.createElement("div");
    shadowContainer.style.display='none';
    map.getPane(G_MAP_FLOAT_SHADOW_PANE).appendChild(shadowContainer);
    this.shadowContainer_ = shadowContainer;
}

LittleInfoWindow.prototype.remove = function() {
    this.container_.parentNode.removeChild(this.container_);

    //don't forget to remove the shadow as well
    this.shadowContainer_.parentNode.removeChild(this.shadowContainer_);
}

LittleInfoWindow.prototype.copy = function() {
    return new LittleInfoWindow(this.marker_,this.html_,this.width_);
}

LittleInfoWindow.prototype.redraw = function(force) {
    if (!force) return;

    //get the content div
    var content = document.createElement("span");
    content.innerHTML = this.html_;
    content.style.font='10px verdana';
    content.style.margin='0';
    content.style.padding='0';
    content.style.border='0';
    content.style.display='inline';

    if(!this.width_ || this.width_=='auto' || this.width_ <= 0) {
        //the width is unknown so set a rough maximum and minimum
        content.style.minWidth = '10px';
        content.style.maxWidth = '500px';
        content.style.width = 'auto';
    } else {
        //the width was set when creating the window
```

```
    content.style.width= width + 'px';
}

//make it invisible for now
content.style.visibility='hidden';

//temporarily append the content to the map container
this.map_.getContainer().appendChild(content);

//retrieve the rendered width and height
var contentWidth = content.offsetWidth;
var contentHeight = content.offsetHeight;

//remove the content from the map
content.parentNode.removeChild(content);
content.style.visibility='visible';

//set the width and height to ensure they
//stay that size when drawn again
content.style.width=contentWidth+'px';
content.style.height=contentHeight+'px';

//set up the actual position relative to your images
content.style.position='absolute';
content.style.left='5px';
content.style.top='7px';
content.style.background='white';

//create the wrapper for the window
var wrapper = document.createElement("div");

//first append the content so the wrapper is above
wrapper.appendChild(content);

//create an object to reference each image
var wrapperParts = {
    tl:{l:0, t:0, w:5, h:7},
    t:{l:5, t:0, w:(contentWidth-6), h:7},
    tr:{l:(contentWidth-1), t:0, w:11, h:9},
    l:{l:0, t:7, w:5, h:contentHeight},
    r:{l:(contentWidth+5), t:9, w:5, h:(contentHeight-2)},
    bl:{l:0, t:(contentHeight+7), w:5, h:5},
    p:{l:5, t:(contentHeight+7), w:17, h:18},
    b:{l:22, t:(contentHeight+7), w:(contentWidth-17), h:5},
    br:{l:(contentWidth+5), t:(contentHeight+7), w:5, h:5}
}
```

```
//create the image DOM objects
for (i in wrapperParts) {
    var img = document.createElement('img');

    //load the image from your local image directory
    //based on the property name of the wrapperParts object
    img.src = 'littleWindow/' + i + '.png';

    //set the appropriate positioning attributes
    img.style.position='absolute';
    img.style.top=wrapperParts[i].t+'px';
    img.style.left=wrapperParts[i].l+'px';
    img.style.width=wrapperParts[i].w+'px';
    img.style.height=wrapperParts[i].h+'px';
    wrapper.appendChild(img);
    wrapperParts[i].img = img;
}

//add any event handlers like the close box
var marker = this.marker_;
GEvent.addDomListener(wrapperParts.tr.img, "click", function() {
        marker.closeLittleInfoWindow();
});

//get the X,Y pixel location of the marker
var pixelLocation = this.map_.fromLatLngToDivPixel(
    this.marker_.getPoint()
);

//position the container div for the window
this.container_.style.position='absolute';
this.container_.style.left = (pixelLocation.x-3) + "px";
this.container_.style.top = (pixelLocation.y
    - contentHeight
    - 25
    - this.marker_.getIcon().iconSize.height
) + "px";
this.container_.style.border = '0';
this.container_.style.margin = '0';
this.container_.style.padding = '0';
this.container_.style.display = 'block';

//append the styled info window to the container
this.container_.appendChild(wrapper);


//add a shadow
this.shadowContainer_.style.position='absolute';
```

```
    this.shadowContainer_.style.left = (pixelLocation.x+15) + "px";
    this.shadowContainer_.style.top = (pixelLocation.y
        - 10
        - this.marker_.getIcon().iconSize.height
    ) + "px";
    this.shadowContainer_.style.border = '1px solid black';
    this.shadowContainer_.style.margin = '0';
    this.shadowContainer_.style.padding = '0';
    this.shadowContainer_.style.display = 'block';

    var shadowParts = {
        sl:{l:0, t:0, w:35, h:26},
        s:{l:35, t:0, w:(contentWidth-40), h:26},
        sr:{l:(contentWidth-5), t:0, w:35, h:26}
    }

    for (i in shadowParts) {
        var img = document.createElement('img');
        img.src = 'littleWindow/' + i + '.png';
        img.style.position='absolute';
        img.style.top=shadowParts[i].t+'px';
        img.style.left=shadowParts[i].l+'px';
        img.style.width=shadowParts[i].w+'px';
        img.style.height=shadowParts[i].h+'px';
        this.shadowContainer_.appendChild(img);
    }

    //pan if necessary so it shows on the screen
    var mapNE = this.map_.fromLatLngToDivPixel(
        this.map_.getBounds().getNorthEast()
    );
    var panX=0;
    var panY=0;
    if(this.container_.offsetTop < mapNE.y) {
        //top of window is above the top edge of the map container
        panY = mapNE.y - this.container_.offsetTop;
    }
    if(this.container_.offsetLeft+contentWidth+10 > mapNE.x) {
        //right edge of window is outside the right edge of the map container
        panX = (this.container_.offsetLeft+contentWidth+10) - mapNE.x;
    }

    if(panX!=0 || panY!=0) {
        //pan the map
        this.map_.panBy(new GSize(-panX-10,panY+30));
    }
}
```

```
//add a new method to GMarker so you
//can use a similar API to the existing info window.
GMarker.prototype.LittleInfoWindowInstance = null;
GMarker.prototype.openLittleInfoWindow = function(content,width) {
    if(this.LittleInfoWindowInstance == null) {
        this.LittleInfoWindowInstance = new LittleInfoWindow(
            this,
            content,
            width
        );
        map.addOverlay(this.LittleInfoWindowInstance);
    }
}
GMarker.prototype.closeLittleInfoWindow = function() {
    if(this.LittleInfoWindowInstance != null) {
        map.removeOverlay(this.LittleInfoWindowInstance);
        this.LittleInfoWindowInstance = null;
    }
}
```

The following sections describe how this code works.

Creating the Overlay Object and Containers

Similar to the Google info window, your info window will require three inputs: a marker on which to anchor the window, the HTML content to display, and an optional width. When you extend this example for use in your own web application, you'll probably add more input parameters or additional methods. You could also add the various methods and properties of the existing `GInfoWindow` class so that your class provides the same API as Google's info window, with tabs and an assortment of options. To keep things simple in the example, we stick to the essentials.

Like the `ToolTip` object you created earlier, the `LittleInfoWindow` object in Listing 9-5 starts off the same way. The `LittleInfoWindow` function provides a construction using the `marker`, `html`, and `width` arguments, while the `GOverlay` is instantiated as the `prototype` object. The first big difference comes in the `initialize()` method where you create two containers. The first container, for the info window, is attached to the `G_MAP_FLOAT_PANE` pane:

```
var container = document.createElement("div");
container.style.display='none';
map.getPane(G_MAP_FLOAT_PANE).appendChild(container);
this.container_ = container;
```

And the second container, for the info window's shadow, is attached to the `G_MAP_FLOAT_SHADOW_PANE` pane:

```
var shadowContainer = document.createElement("div");
shadowContainer.style.display='none';
map.getPane(G_MAP_FLOAT_SHADOW_PANE).appendChild(shadowContainer);
this.shadowContainer_ = shadowContainer;
```

Tip A shadow isn't required for overlays, but it provides a nice finishing touch to the final map and makes your web application look much more polished and complete.

Next, the `remove()` and `copy()` methods are again identical in functionality to the `ToolTip` overlay, except the `remove()` method also removes the second `shadowContainer` along with the info window container.

Drawing a LittleInfoWindow

The most complicated part of creating an info window is properly positioning it on the screen with the `redraw()` method, and the problem occurs only when you want to position it *above* the existing marker or point.

When rendering HTML, the page is drawn on the screen from the top down and left to right. You can assign sizes and positions to `html` elements using CSS attributes, but in general, if there are no sizes or positions, things will start at the top and flow down. When you create the info window in the `redraw()` method, you'll take the HTML passed into the constructor, put it in a `content div`, and wrap it with the appropriate style. On an empty HTML page, you know the top left corner of the `content div` is at (0,0), but where is the bottom right corner? The bottom right corner is dependent on the content of the `div` and the general style of the `div` itself.

The ambiguity in the size of the `div` is compounded when you want to position the `div` on the map. The Google Maps API requires you to position the overlay using *absolute* positioning. To properly position the info window, so the arrow is pointing at the marker, you need to know the height of the info window, but as we said, the height varies based on the content. Luckily for you, browsers have a little-known feature that allows you to access the rendered position and size of elements on a web page.

Determining the Size of the Container

When creating the `redraw()` function, the first thing you'll do is put the HTML into a `content div` and apply the appropriate base styles to the `div`:

```
var content = document.createElement("div");
content.innerHTML = this.html_;
content.style.font='10px verdana';
content.style.margin='0';
content.style.padding='0';
content.style.border='0';
content.style.display='inline';

if(!this.width_ || this.width_=='auto' || this.width_ <= 0) {
    //the width is unknown so set a rough maximum and minimum
    content.style.minWidth = '10px';
    content.style.maxWidth = '500px';
    content.style.width = 'auto';
```

```
} else {
    //the width was set when creating the window
    content.style.width= width + 'px';
}

//make it invisible for now.
content.style.visibility='hidden';
```

The `display='inline'` and the last `style` attribute, `visibility='hidden'`, are important for the next step. To determine the `div`'s rendered position and size properties, you need to access hidden properties of the `div` elements. When rendered on the page, browsers attach `offset`*XXX* properties where the *XXX* is `Left`, `Right`, `Width`, or `Height`. These give you the position and size in pixels of the DOM element after it's rendered. For your info window, you're concerned with the `offsetWidth` and `offsetHeight`, as you'll need them to calculate the overall size of the window.

To access the offset variables, you'll first need to render the `content` `div` on the page. At this point in the overlay, the `content` DOM element exists only in the browser's memory and hasn't been "drawn" yet. To do so, append the `content` to the map container and retrieve the width and height before removing it again from the map container:

```
this.map_.getContainer().appendChild(content);
var contentWidth = content.offsetWidth;
var contentHeight = content.offsetHeight;
content.parentNode.removeChild(content);
content.style.visibility='visible';

//set the width and height to ensure they stay that size when drawn again.
content.style.width=contentWidth+'px';
content.style.height=contentHeight+'px'
```

The brief existence of the `content` `div` inside the map container allows the browser to set the offset properties so you can retrieve the `offsetWidth` and `offsetHeight`. As we mentioned, the `inline` display and the `hidden` visibility are important to retrieving the correct size. When the display is `inline`, the bounding `div` collapses to the size of the actual content, rather than expanding to a width of 100%, giving you an accurate width. Setting the visibility to `hidden` prevents the content from possibly flashing on the screen for a moment, but at the same time, preserves the size and shape of the `div`.

Building the Wrapper

Now that you have the size of the content box, the rest is pretty straightforward. First, style the `content` `div` accordingly and create another `div`, the `wrapper`, to contain the `content` and the additional images for the eye candy bubble wrapper from Figure 9-13:

```
content.style.position='absolute';
content.style.left='5px';
content.style.top='7px';
content.style.background='white';
var wrapper = document.createElement("div");
wrapper.appendChild(content);
```

To minimize the HTML required for the `LittleInfoWindow`, the images in the wrapper can be positioned using *absolute* positioning. The sample wrapper consists of nine separate images: four corners, four sides, and an additional protruding arm, as outlined in Figure 9-14 (along with the shadow and marker images). To give the new info window a similar feel to Google's info window, the upper right corner has also been styled with an X in the graphic to act as the close box.

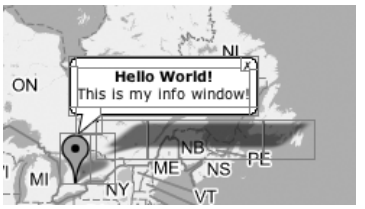


Figure 9-14. *Outlined images for the LittleInfoWindow wrapper*

To create the `wrapper` object in Listing 9-5, you could use the `innerHTML` property to add the images using regular HTML, but that wouldn't allow you to easily attach event listeners to the images. By creating each image as a DOM object

```
var wrapperParts = {
    tl:{l:0, t:0, w:5, h:7},
    t:{l:5, t:0, w:(contentWidth-6), h:7},
    - cut -
}

//create the images
for (i in wrapperParts) {
    var img = document.createElement('img');
    - cut -
    wrapper.appendChild(img);
    wrapperParts[i].img = img;
}
```

and using the `wrapper.appendChild()` method, you can then attach event listeners directly to image DOM elements, as when you want to add a click event to the close box:

```
var marker = this.marker_;
GEvent.addDomListener(wrapperParts.tr.img, "click", function() {
    marker.closeLittleInfoWindow();
});
```

Now all that's left to do with the `LittleInfoWindow` container is position it on the map and append the `wrapper`. The design of the `LittleInfoWindow` has the arm protruding in the lower left corner, so you'll want to position the top of the container so that the arm rests just above the marker. You can get the marker's position using the `GMap2.fromLatLngToDivPixel()` method you saw earlier in the chapter, and then use the calculated height of the `LittleInfoWindow` plus the height of the marker icon to determine the final resting position:

```
var pixelLocation = this.map_.fromLatLngToDivPixel(this.marker_.getPoint());
this.container_.style.position='absolute';
this.container_.style.left = (pixelLocation.x-3) + "px";
this.container_.style.top = ( pixelLocation.y
    - contentHeight
    - 25
    - this.marker_.getIcon().iconSize.height
) + "px";
this.container_.style.display = 'block';

this.container_.appendChild(wrapper);
```

Adding a Few Shades of Finesse

Your `LittleInfoWindow` should now be working, but a few tasks remain before we can call it complete. First, let's add a shadow to the window similar to the one on Google's info window. The shadow images are also supplied in the PSD files accompanying this book. The process for adding the shadow is similar to the wrapper you just created. We won't go through it again here, but you can take a look at the complete code in Listing 9-5 and see the example there. The shadow, in this case, expands only horizontally with the size of the wrapper, but you could easily add vertical expansion as well.

Listing 9-5 also includes some pan adjustments when your window initially opens. The nice thing about the Google info window is when it opens off-screen, the map pans until the window is visible onscreen. You can easily add this same functionality by comparing the upper right corner of your `LittleInfoWindow` with the top and right edges of the map container:

```
var mapNE = this.map_.fromLatLngToDivPixel(this.map_.getBounds().getNorthEast());
var panX=0;
var panY=0;
if(this.container_.offsetTop < mapNE.y) {
    panY = mapNE.y - this.container_.offsetTop;
}
if(this.container_.offsetLeft+contentWidth+10 > mapNE.x) {
    panX = (this.container_.offsetLeft+contentWidth+10) - mapNE.x;
}
if(panX!=0 || panY!=0) {this.map_.panBy(new GSize(-panX-10,panY+30)); }
```

Then, if necessary, you can pan the map, just as Google does, to show the open window. If you check out the online example at `http://book.earthcode.com/chap_nine/custom_info_window/`, you can see the pan in action by moving the marker to the top or right edge and then clicking it to open the `LittleInfoWindow`.

Using the LittleInfoWindow

The last and final addition for your `LittleInfoWindow` should be the creation of the appropriate methods on the `GMarker` class, in the same way you created methods for the `ToolTip` earlier. Again, by adding open and close methods to the `GMarker` class

```
GMarker.prototype.LittleInfoWindowInstance = null;
GMarker.prototype.openLittleInfoWindow = function(content,width) {
    if(this.LittleInfoWindowInstance == null) {
        this.LittleInfoWindowInstance = new LittleInfoWindow(this,content,width)
        map.addOverlay(this.LittleInfoWindowInstance);
    }
}
GMarker.prototype.closeLittleInfoWindow = function() {
    if(this.LittleInfoWindowInstance != null) {
        map.removeOverlay(this.LittleInfoWindowInstance);
        this.LittleInfoWindowInstance = null;
    }
}
```

you can access your custom info window with an API similar to the Google info window using something like this:

```
GEvent.addListener(marker,'click',function() {
    if(marker.LittleInfoWindowInstance) {
        marker.closeLittleInfoWindow();
    } else {
        marker.openLittleInfoWindow('<b>Hello World!</b>➥
<br/>This is my info window!');
    }
});
```

The difference from Google's info window is that the `LittleInfoWindowInstance` is attached to the `GMarker`, not the map, so you have the added advantage of opening more than one window at the same time. If you want to force only one window open at a time, you'll need to track the instance using the map object rather than the marker.

Implementing Your Own Map Type, Tiles, and Projection

By default, three types of maps are built into the Google Maps API:

Map (often referred to as *normal*): Shows the earth using outlines and colored objects, similar to a printed map you might purchase for driving directions

Satellite: Shows the map using satellite photos of the earth taken from space

Hybrid: Is a mixture of the satellite images overlaid with information from the normal map type

Each map is an instance of the GMapType class, and each has its own constant G_NORMAL_MAP, G_SATELLITE_MAP, and G_HYBRID_MAP, respectively. To quickly refer to all three, there is also the G_DEFAULT_MAP_TYPES constant, which is an array of the previous three constants combined.

In the example in this section, you'll create your own map using a new projection and the NASA Visible Earth images (http://visibleearth.nasa.gov). But first, you need to understand how the map type, projection, and tiles work together.

GMapType: Gluing It Together

Understanding the GMapType is key to understanding how the different classes interact to create a single map. Each instance of the GMapType class defines the draggable map you see on the screen. The map type tells the API what the upper and lower zoom levels are, which GTileLayer objects to include in the map, and which GProjection to use for latitude and longitude calculations. A typical GMapType object would look similar to this:

```
var MyMapType = new GMapType(
    [MyTileLayer1, MyTileLayer2],
    MyProjection,
    'My Map Type',{
        shortName:'Mine',
        tileSize:256,
        maxResolution:5,
        minResolution:0
});
```

MyTileLayer1 and MyTileLayer2 would be instances of the GTileLayer class, and MyProjection would be an instance of the GProjection class. The third parameter for GMapType is the label to show on the map type button in the upper right corner of the Google map. You'll also notice the fourth parameter is a JavaScript object implementing the properties of the GMapTypeOptions class listed in Table 9-2. In this case, the short name is Mine, the tile size is 256×256 pixels, and the zoom levels are restricted to 0 through 5.

Caution In your map type, all the tiles in each of the GTileLayer objects must be of equal size. You can't mix and match tile sizes within the same map type.

Table 9-2. *GMapTypeOptions Properties*

Property	Description
`shortName`	The short name returned from `GMapType.getName(true)` and used in the `GOverviewMapControl`. The default is the same as the name supplied in the `GMapType` arguments.
`urlArg`	The optional parameters for the URL of the map type; can be retrieved using `GMapType.getUrlArg()`.
`maxResolution`	The maximum zoom level of this map type.
`minResolution`	The minimum zoom level of this map type.
`tileSize`	The tile size. The default is `256`.
`textColor`	The text color returned by `GMapType.getTextColor()`. The default is `black`.
`linkColor`	The text link color returned by `GMapType.getLinkColor()`. The default is `#7777cc`.
`errorMessage`	An optional message returned by `GMapType.getErrorMessage()`.

The `GMapType` object directs tasks to various other classes in the API. For instance, when you need to know where a longitude or latitude point falls on the map, the map type asks the `GProjection` where the point should go. When you drag the map around, the `GTileLayer` receives requests from the map type to get more images for the new map tiles.

In the case where you don't really need a brand-new map type and just want to add a tile layer to an existing map (as with the custom tile method described in Chapter 7), you can simply reuse Google's existing projection and tiles, layering your own on top. Using Google's projection and tiles is easy. Creating your own `GProjection` and `GTileLayer` is where things get a bit tricky.

GProjection: Locating Where Things Are

The `GProjection` interface handles the math required to convert latitude and longitude into relative screen pixels and back again. It tells the map where `GLatLng(-80,43)` really is, and it tells your web application what latitude and longitude is at position `GPoint(64,34)`. Besides that, it's also responsible for the biggest untruth in the map.

You may not realize it, but when you look at a map—any map—it's stretching the truth. A map printed on a piece of paper or displayed on a screen is a two-dimensional representation of a three-dimensional object. People have long understood the earth is round, but a round object can't be represented accurately in a flat image without losing or skewing some of the information. To create the flat map, the round earth is *projected* onto the flat surface using some mathematical or statistical process, but as we said, projections do sometimes *stretch* the truth.

For example, take a look at Figure 9-15, where we've outlined the United States and Greenland. Greenland, on a round globe, covers about 836,000 square miles (2,166,000 square kilometers), and the United States covers about 3,539,000 square miles (9,166,000 square kilometers). That means Greenland is really about 20% of the area of the United States, but on the Google map (and many other maps), it looks as though you could fit two of the United States inside Greenland. It also looks as though Alaska, also outlined, is about half the area of the United States. This is because the Google API uses the *Mercator projection*, a mathematical method that vertically stretches distances by the same proportion as the horizontal distances so that shape and direction are preserved.

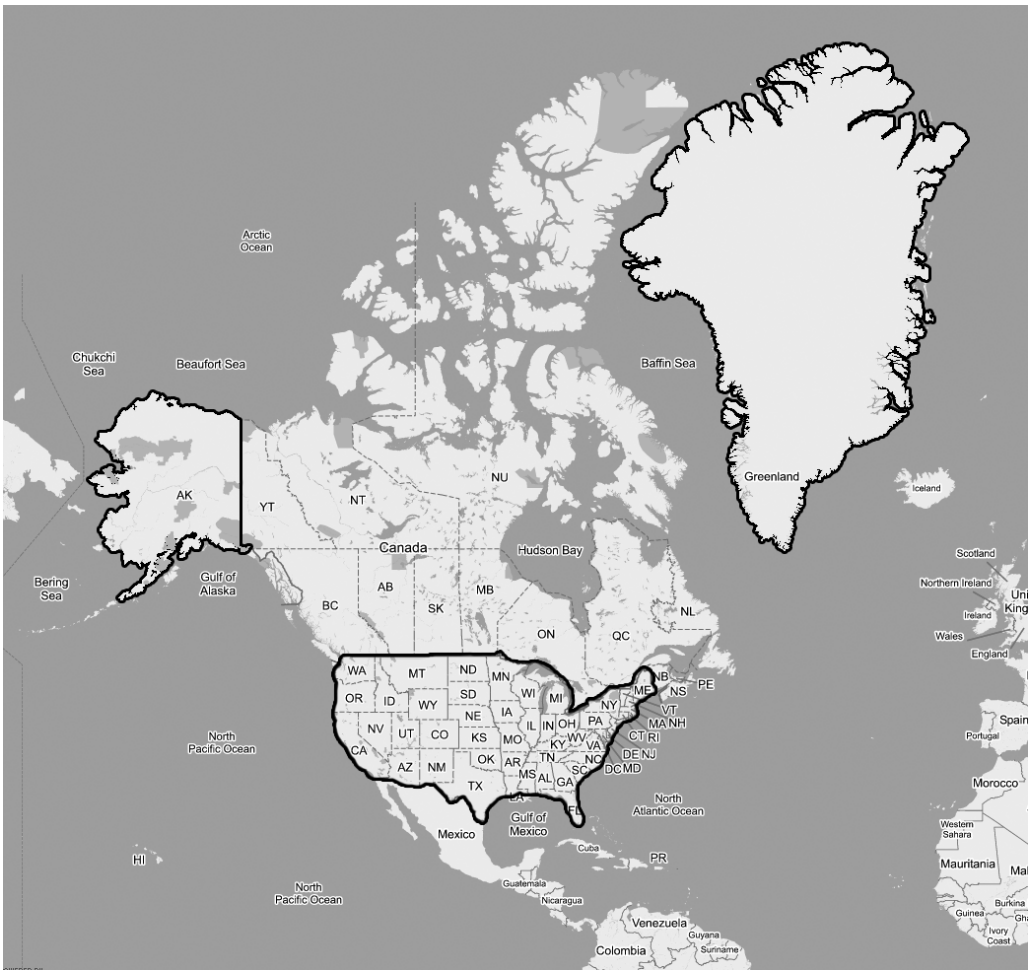


Figure 9-15. *Comparing the United States, Greenland, and Alaska on a Mercator projection*

Understanding Projection Types

Without going deep into mathematical theories and discussions, map projections can generally be divided into three categories—planar, conic, and cylindrical—but some projections, such as the Mollweide homolographic and the sinusoidal projection, are hybrids. Each category has dozens of different variations depending on the desired use and accuracy:

Planar: A planer map projection, often referred to as an *azimuthal projection*, is created by placing a flat plane tangent to the globe at one point and projecting the surface onto the plane from a single point source within the globe, as represented in Figure 9-16. Imagine an image on a wall, created by placing a light inside a glass globe. The resulting circular image would be a planar map representing the round glass globe. The positions of the latitude and longitude lines will vary depending on the position of the plane relative to the globe, and planar projections also vary depending on where the common point is within the globe. These projections are often used for maps of the polar regions.

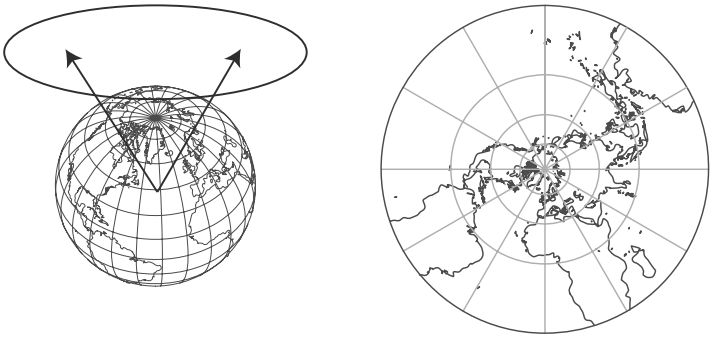


Figure 9-16. *Creating a planar projection*

Conic: Unlike the planer projection, the conic projection uses a cone placed on the globe like an ice cream cone, tangent to some parallel, as shown in Figure 9-17. Then like the planar projection, the globe is projected into the cone using the center of the globe as the common point. The cone can then be cut along one of the meridians and placed flat. Latitude lines are represented by straight lines converging at the center; longitude lines are represented by arcs with the apex of the cone at their center. Conic projections vary depending on the position of the cone and the size of the cone.

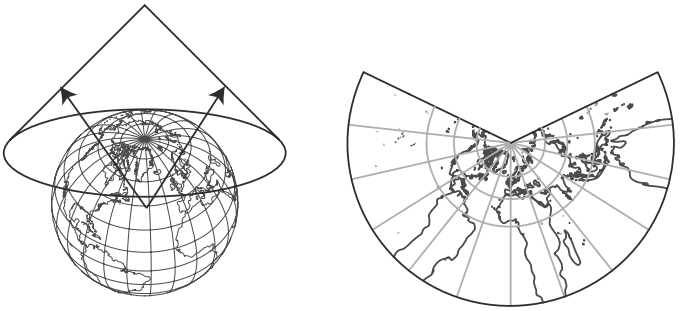


Figure 9-17. *Creating a conic projection*

Cylindrical: Cylindrical projections are similar to both the other two types of projections; however, the plane is wrapped around the globe like a cylinder, tangent to the equator, as illustrated in Figure 9-18. The globe is then projected onto the cylinder from a central point within the globe, or along a central line running from pole to pole. The resulting map has equidistant parallel longitude lines and parallel latitude lines that increase in distance as you move farther from the equator. The difficulty with cylindrical projections is that the poles of the earth can't be represented accurately.

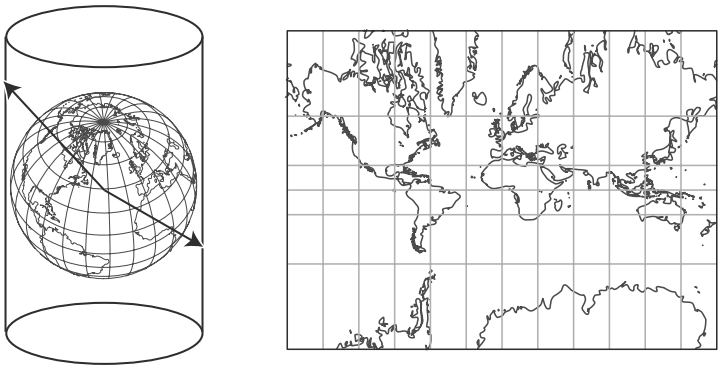


Figure 9-18. *Creating a cylindrical projection*

The Mercator projection used by the Google Maps API is a cylindrical projection; however, the latitude lines are mathematically adjusted using one of the following equations where Δ represents the longitude and Φ represents the latitude:

$$x = \lambda - \lambda_0$$

$$y = \ln\left[\tan\left(\frac{1}{4}\pi + \frac{1}{2}\varphi\right)\right]$$

$$= \frac{1}{2}\ln\left(\frac{1+\sin\phi}{1-\sin\phi}\right)$$

$$= \sinh^{-1}(\tan\phi)$$

$$= \tanh^{-1}(\sin\varphi)$$

$$= \ln(\tan\varphi + \sec\varphi)$$

The equations preserve more realistic shapes, as shown in Figure 9-19.

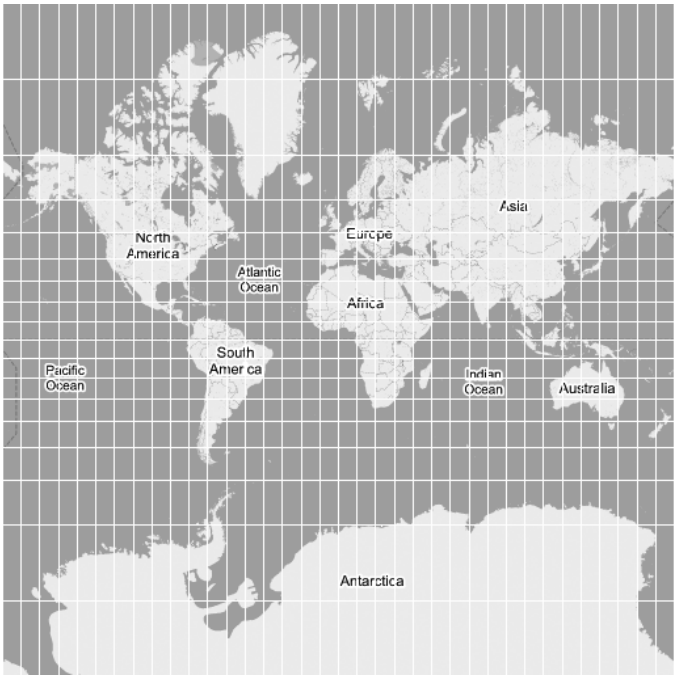


Figure 9-19. *Latitude and longitude lines of the Google Maps API's Mercator projection*

The downside with Mercator projections, as you can see in Figure 9-15, is that areas farther away from the equator are greatly exaggerated and the poles themselves can't be shown.

Using a Different Projection

By default, all of the maps in the API use the built-in `GMercatorProjection` class. The `GMercatorProjection` is an implementation of the `GProjection` interface using the Mercator projection. If your custom map image is using the Mercator projection, you don't have to worry about implementing your own `GProjection` interface, and you can just reference the `GMercatorProjection` class. If you would like to use a projection other than the Mercator projection, you need to create a new class for your projection and implement the methods listed in Table 9-3.

Table 9-3. *Methods Required to Implement a GProjection Class*

Method	Return Value	Description
`fromLatLngToPixel (latlng,zoom)`	`GPoint`	Given a latitude, longitude, and zoom provided in the arguments, this method returns the X and Y pixel coordinates of the location relative to the bounding `div` of the map.
`fromPixelToLatLng (point,zoom,unbounded)`	`GLatLng`	Given the pixel coordinates and the zoom, this method returns the geographical latitude and longitude of the location. If the `unbounded` flag is `true`, the geographical longitude should *not* wrap when beyond -180 or 180 degrees.
`tileCheckRange (tile,zoom,tilesize)`	`Boolean`	This method returns `true` if the `tile` parameter is within a valid range (`tile,zoom,tilesize`) for the known map type. If `false` is returned, the map will display an empty tile. In the case where you want the map to wrap horizontally, you may need to modify the tile index to point to the index of an existing tile.
`getWrapWidth(zoom)`	`Integer`	Given the zoom level, this method returns the pixel width of the entire map at the given zoom. The API uses this value to indicate when the map should repeat itself. By default, `getWrapWidth()` returns `Infinity`, and the map does not wrap.

Listing 9-6 shows a generic implementation of an equidistant cylindrical projection, which you'll use in the "The Blue Marble Map: Putting it All Together" section later in the chapter to create a map using the NASA Visible Earth images as tiles.

Listing 9-6. *Equidistant Cylindrical Projection*

```
EquidistantCylindricalProjection = new GProjection();

EquidistantCylindricalProjection.mapResolutions = [256,512,1024]

EquidistantCylindricalProjection.fromLatLngToPixel = function(latlng,zoom) {
    var lng = parseInt(Math.floor((this.mapResolutions[zoom] / 360) *➥
(latlng.lng() + 180)));
    var lat = parseInt(Math.floor(Math.abs((this.mapResolutions[zoom] / 2 / 180) *➥
(latlng.lat()-90))));
    var point = new GPoint(lng,lat);
    return point;
}

EquidistantCylindricalProjection.fromPixelToLatLng =➥
function(pixel,zoom,unbounded) {
    var lat = 90-(pixel.y / (this.mapResolutions[zoom] / 2 / 180));
    var lng = (pixel.x / (this.mapResolutions[zoom] / 360)) - 180;
    var latlng = new GLatLng(lat,lng);
    return latlng;

}
```

```
EquidistantCylindricalProjection.tileCheckRange = function(tile,zoom,tileSize){
    var rez = this.mapResolutions[zoom];
    //check if it is outside the latitude range
    //the height for the Blue Marble maps are always 1/2 the width
    if(tile.y < 0 || tile.y * tileSize >= rez / 2){ return false; }

    //check if it is outside the longitude range and if so, wrap the map
    //by adjusting tile x
    if(tile.x < 0 || tile.x * tileSize >= rez){
        var e = Math.floor( rez / tileSize );
        tile.x = tile.x % e;
        if(tile.x < 0){ tile.x += e; }
    }
    return true;
}

EquidistantCylindricalProjection.getWrapWidth = function(zoom){
    return this.mapResolutions[zoom];
}
```

The equidistant cylindrical projection is created by plotting the latitude and longitude values from the globe in a 1:1 ratio on a plane, as shown in Figure 9-20. This creates a map whose width, unlike Google's Mercator projection, is always twice its height, while latitude and longitude lines are all at equal distances. If you compare your final map with the Google map, your equidistant cylindrical map will actually be half the height and thus half the number of overall tiles per zoom level.

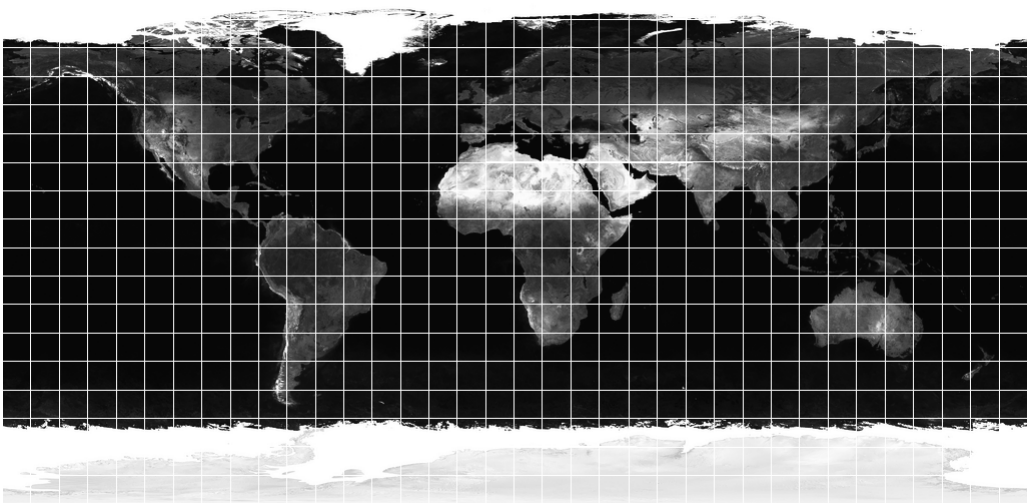


Figure 9-20. *Equidistant cylindrical projection*

You'll also notice the projection in Listing 9-6 has an additional property, `EquidistantCylindricalProjection.mapResolutions`, to hold the overall width of the map at each zoom level.

Caution Your implementation of the `GProjection` interface is dependent on the resolution of the map image you plan to use. If you want to reuse the `GMercatorProjection`, your map images must match the sizes discussed in the next section.

GTileLayer: Viewing Images

By now, you've probably already figured out that the map image, regardless of the type, is actually composed of smaller square images referred to as *tiles*. In the Google Maps API, each of these tiles is 256 × 256 pixels, and at the lowest zoom level (0), the entire earth is represented in one 256 × 256 tile, as shown in Figure 9-21. Some maps, such as the hybrid map in the API, use more than one layer of tiles at a time. In Chapter 7, you saw how you could use a tile layer to map large data sets, and in that instance, you added a tile layer to an existing Google map.

Figure 9-21. *The entire earth at zoom level 0 using one 256 × 256 tile*

Understanding Tiles

When creating your map with custom tiles, it's important to consider resources and storage required for the tiles. The number of tiles on a map is directly related to the zoom level of the map by

```
Number of tiles = (2 ^ zoom) ^ 2
```

This means at zoom level 0, there is one tile, and at zoom level 17, there are 17,179,869,184 billion tiles, not to mention the accumulated total for all the zoom levels combined. Table 9-4 shows the breakdown of number of tiles, map size, and rough storage requirements for each of the zoom levels 0 through 17.

Table 9-4. *Tile Size and Storage Requirements for Each Zoom Level of the Google Mercator Projection*

Zoom	Tile Dimensions	Pixel Dimensions	Number of Tiles	Disk Space Required*
0	1 × 1	256 × 256	1	10.209KB
1	2 × 2	512 × 512	4	40.839KB
2	4 × 4	1024 × 1024	16	163.359KB
3	8 × 8	2048 × 2048	64	653.437KB
4	16 × 16	4096 × 4096	256	2.552MB
5	32 × 32	8192 × 8192	1024	10.209MB
6	64 × 64	16384 × 16384	4096	40.839MB
7	128 × 128	32768 × 32768	16384	163.359MB
8	256 × 256	65536 × 65536	65536	653.437MB
9	512 × 512	131072 × 131072	262144	2.552GB
10	1024 × 1024	262144 × 262144	1048576	10.209GB
11	2048 × 2048	524288 × 524288	4194304	40.839GB
12	4096 × 4096	1048576 × 1048576	16777216	163.359GB
13	8192 × 8192	2097152 × 2097152	67108864	653.437GB
14	16384 × 16384	4194304 × 4194304	268435456	2.552TB
15	32768 × 32768	8388608 × 8388608	1073741824	10.209TB
16	65536 × 65536	16777216 × 16777216	4294967296	40.839TB
17	131072 × 131072	33554432 × 33554432	17179869184	163.359TB
Total			2906492245	217.812TB

**Based on an average file size of 10,455 bytes per tile*

Looking at Table 9-4, you quickly realize that it may not be feasible to create a large map at very high resolutions unless you have a fairly large storage facility and a lot of bandwidth to spare. Also, remember that Table 9-4 represents *one* map type and *one* tile layer. If you have a smaller map with multiple tile layers or various map images, you may also run into storage problems.

Creating Your GTileLayer

To create a tile layer for your map, you can follow the same process outlined in Chapter 7 and create a new `GTileLayer` object with the methods listed in Table 9-5.

Table 9-5. *Methods Required for a GTileLayer*

Method	Return Value	Description
`getTileUrl(tile,zoom)`	`String`	Returns the URL for the tile image. The URL can point to any domain, as image files are not required to be on the same domain as the page is.
`isPng()`	`Boolean`	Returns `true` if the tiles are in PNG format and could be transparent. You can still use transparent GIFs if this returns `false`, but if you use transparent PNGs this should be `true` so the API knows to fix cross-browser issues with transparent or translucent PNG files.
`getOpacity()`	`Float`	Returns the opacity to apply to the tiles: `1` is opaque and `0` is transparent. Remember that when dealing with translucent layers, performance may be degraded.

■ **Caution** Two additional methods for the `GTileLayer` class are `minResolution()` and `maxResolution()`. These return the minimum and maximum zoom levels for the tile layer. At the time of publishing, if you try to override them, the map behaves erratically. So you should leave them out of your custom tile layer and use the second and third arguments for the `GTileLayer` class to assign the maximum and minimum zoom levels.

The URL in the `getTileUrl()` method can point to a server-side script that generates tiles on the fly, as in the method described in Chapter 7, or you may want to preslice your image and save each tile with an appropriate name. Regardless of which method you choose, a `GTileLayer` simply requests the tile at the appropriate index and doesn't care how you create the image.

As shown in Listing 9-7, there is very little code required for a tile layer. You simply define the URL for each tile and pass in the appropriate zoom level restrictions. If you want, when adding multiple layers of tiles, you can adjust the opacity of each layer as well by using the `getOpacity()` method.

Listing 9-7. *Creating a GTileLayer*

```
var myTiles = new GTileLayer(new GCopyrightCollection(),0,10);

myTiles.getTileUrl = function(tile,zoom){
    return 'http://example.com/tiles/' + zoom + '.' + tile.x + '.'➥
+ tile.y + '.png';
};

myTiles.isPng = function() { return true; }
myTiles.getOpacity = function() { return 1.0; }
```

The Blue Marble Map: Putting It All Together

Now that you have an idea of all the interrelating parts of the map type, projection, and tiles, you can put it all together to create your own map.

If you don't have any readily available satellite photos lying around, you'll probably need to turn to other sources for map imagery. Luckily, NASA can provide you with just what you're looking for. The Visible Earth project at `http://visibleearth.nasa.gov`, shown in Figure 9-22, has a variety of public domain images you can download and use in your projects. Some, like the monthly Blue Marble: Next Generation images at `http://earthobservatory.nasa.gov/Newsroom/BlueMarble/BlueMarble_monthlies.html`, are provided at a resolution of 500 meters per pixel, enough to make a Google map to zoom level 9. The images are provided free of charge. According to the terms of use at `http://visibleearth.nasa.gov/useterms.php`, the only thing you need to do in return is provide credits for the imagery to NASA and the Visible Earth team, with a link back to `http://visibleearth.nasa.gov/`.

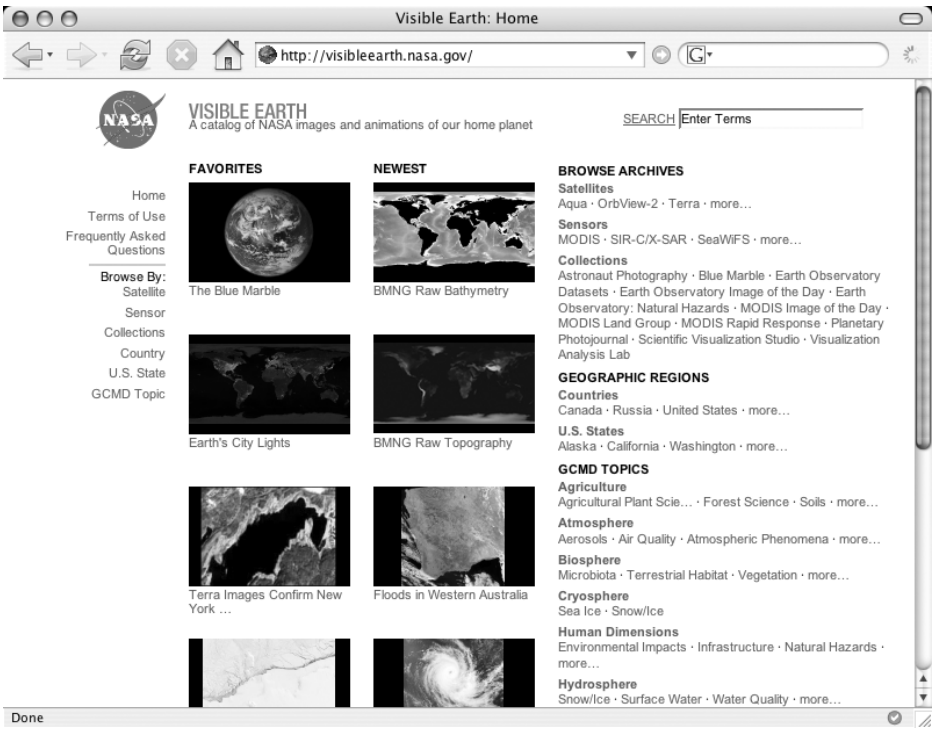


Figure 9-22. *The NASA Visible Earth web site*

Conveniently, the Blue Marble monthly images and various others are created using the equidistant cylindrical projection you saw earlier (in Listing 9-6). There are other projections and image types, such as one "side" of the earth as a circle, but for this example, you'll be working with the equidistant cylindrical images. If you would like to see working examples of other projections, check out the web site for this book at `http://googlemapsbook.com`.

The Images

For the final map, you'll need to create tiles for three maps, each with about five zoom levels. You'll be using these three images:

- *Earth's City Lights*:
  `http://veimages.gsfc.nasa.gov/1438/land_ocean_ice_lights_2048.tif`
- *The Blue Marble: Land Surface, Ocean Color, and Sea Ice*:
  `http://veimages.gsfc.nasa.gov/2430/land_ocean_ice_8192.tif`
- *The Blue Marble: Land Surface, Ocean Color, and Sea Ice and Clouds*:
  `http://veimages.gsfc.nasa.gov/2431/land_ocean_ice_cloud_8192.tif`

The first image, Earth's City Lights, is only 2048×1024 pixels. The other images are 8192×4096 pixels. By referencing Table 9-4 earlier in the chapter, you can see the two images at 8192 pixels fit nicely into zoom level 5, whereas the City Lights image at 2048 pixels will only go to a maximum of zoom level 3. You could probably increase the dimension by one zoom level using an image-editing program, but these three images will suffice for the example.

The Blue Marble Cylindrical Projection

As we mentioned, the three Blue Marble images you're using for the example were created using the equidistant cylindrical projection you saw earlier in Listing 9-6. The only modifications you need to make are to add the appropriate map resolutions to account for zoom levels 0 through 5 and rename the projection to `BlueMarbleProjection` so you can easily distinguish it from other projections you might make. Listing 9-8 shows the projection for this example.

■ **Tip** If you decide to integrate these or other images into your own maps, you could create a generic projection with a `setZoomResolution()` method that could add the various map resolutions appropriate for the given application. That way, you could easily reuse your projection without restricting it to specific zoom levels or map resolutions.

Listing 9-8. *BlueMarbleProjection for Your Custom Map Images*

```
BlueMarbleProjection = new GProjection();

BlueMarbleProjection.mapResolutions = [256,512,1024,2048,4096,8192]

BlueMarbleProjection.fromLatLngToPixel = function(latlng,zoom) {
    var lng = parseInt(Math.floor((this.mapResolutions[zoom] / 360) *➥
(latlng.lng() + 180)));
    var lat = parseInt(Math.floor(Math.abs((this.mapResolutions[zoom] / 2 / 180) *➥
(latlng.lat() - 90))));
    var point = new GPoint(lng,lat);
    return point;
}
```

```
BlueMarbleProjection.fromPixelToLatLng = function(pixel,zoom,unbounded) {
    var lat = 90 - (pixel.y / (this.mapResolutions[zoom] / 2 / 180));
    var lng = (pixel.x / (this.mapResolutions[zoom] / 360)) - 180;
    var latlng = new GLatLng(lat,lng);
    return latlng;
}

BlueMarbleProjection.tileCheckRange = function(tile,zoom,tileSize){
    var rez = this.mapResolutions[zoom];
    if(tile.y < 0 || tile.y * tileSize >= rez / 2){ return false; }
    if(tile.x < 0 || tile.x * tileSize >= rez){
        var e = Math.floor( rez / tileSize );
        tile.x = tile.x % e;
        if(tile.x < 0){ tile.x += e; }
    }
    return true;
}

BlueMarbleProjection.getWrapWidth = function(zoom){
    return this.mapResolutions[zoom];
}
```

The Blue Marble Tiles

The Google Maps API assumes a tile size of 256 × 256 pixels. Although you can change the tile size by using the `GMapType tileSize` option, the Blue Marble images divide nicely by 256, so there's no reason to change the default size for this example. Keeping the same tile size will also allow you to continue reusing most of the other examples in the book, without the need to modify code to accommodate a different tile size.

Slicing and Dicing

To serve up the tiled images for the three maps, you have a few options, including dynamically creating each tile on the fly, preslicing the images, and storing them all appropriately on the server, or a combination. Taking into consideration the storage requirements outlined in Table 9-4, and the processing power you'll need to continually slice the images on the fly, you'll probably opt to spend a little money on a hard drive, if necessary, and preslice your images. The three maps, sliced for each zoom level, will occupy only about 40MB of disk space, whereas slicing the images at each request will create a huge drain on resources and slow down the server.

To slice your images, you could use Adobe Photoshop's scripting capabilities and follow the instructions at `http://www.mapki.com/index.php?title=Automatic_Tile_Cutter`, or you could install some open source image-editing utilities, such as the ImageMagick `convert` utility.

Tip To install ImageMagick, visit `http://www.imagemagick.org/script/index.php`. You'll find installation instructions and binaries for both Unix and Windows systems. If you've never used ImageMagick before, we highly recommend you browse the manual to see all the great tools it offers. Also, check out the book *The Definitive Guide to ImageMagick* by Michael Still (`http://apress.com/book/bookDisplay.html?bID=10052`). If you're looking for some quick examples, check out `http://www.cit.gu.edu.au/~anthony/graphics/imagick6/`, where you'll find illustrated examples of how to use each of ImageMagick's commands. If you deal with dynamically generating images on a daily basis, you'll find ImageMagick an essential tool to add to your collection.

To tile your images with ImageMagick, first, if you haven't already done so, download the three images into a `tiles` directory, and then create subdirectories for each image's tiles. Your directory structure should look like this:

```
tiles/
    land_ocean_ice
    land_ocean_ice_8192.tif
    land_ocean_ice_cloud
    land_ocean_ice_cloud_8192.tif
    land_ocean_ice_lights
    land_ocean_ice_lights_2048.tif
```

Then it's as simple as running the following command to create each of the tiles for each of the images at each zoom level:

```
convert filename.tif -resize widthxheight -crop 256x256➥
directory/tile.zoomlevel.%d.png
```

For the resize width and height, refer back to Table 9-4.

For example, to create the tiles for the `land_ocean_ice_lights` image, you would execute the following four commands:

```
convert land_ocean_ice_lights_2048.tif -crop 256x256➥
land_ocean_ice_lights/tile.3.%d.png
convert land_ocean_ice_lights_2048.tif -resize 1024x512 -crop 256x256➥
land_ocean_ice_lights/tile.2.%d.png
convert land_ocean_ice_lights_2048.tif -resize 512x256 -crop 256x256➥
land_ocean_ice_lights/tile.1.%d.png
convert land_ocean_ice_lights_2048.tif -resize 256x256➥
land_ocean_ice_lights/tile.0.%d.png
```

Executing these four commands will create the tiles for each zoom level between 0 and 3. One tile for zoom level 0 will be created and named `tile.0.0.png`, while 32 tiles for zoom level 3 will be created and named `tile.3.0.png` through `tile.3.31.png`. The tiles you create with ImageMagick will be numbered 0 through X, starting with the top left corner and reading left to right, as illustrated in Figure 9-23. It's important that you remember this pattern when you create the `getUrl` method for the `GTileLayer`.

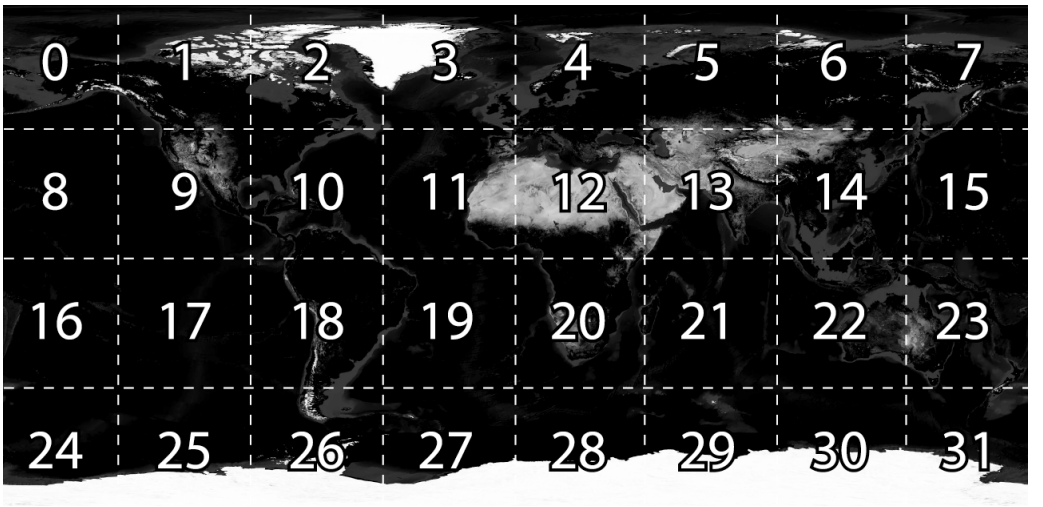


Figure 9-23. *Tile placement produced by ImageMagick*

For the other two images, `land_ocean_ice_8192.tif` and `land_ocean_ice_cloud_8192.tif`, you can follow the same process, but start at zoom level 5 and go to 0.

■ **Caution** If you are using an image that's excessively large, you may run into memory problems while running ImageMagick to create your tiles. ImageMagick tries to get as much main memory as possible when converting images so the conversion can run as fast as possible. To limit memory consumption and leave some for other processes, you can add `-limit memory 32 -limit map 32` to the command. This will force ImageMagick to use the disk cache rather than hog memory, but the processing time may be much slower.

Creating the GTileLayer Objects

For your Blue Marble map, you need to create three different `GTileLayer` objects, similar to the earlier generic object from Listing 9-7. For the Blue Marble tile layers, you'll need to change the generic `getUrl` method to account for the numbering scheme you used when you created the tiles, and you'll need to modify the URL to point to the actual location of your tiles for each of the three images:

```
myTiles.getTileUrl = function(tile,zoom){
    return 'http://example.com/tiles/' + zoom + '.' + tile.x + '.'➥
+ tile.y + '.png';
};
```

Each request to `getTileUrl` contains two arguments: the `tile` and the `zoom`. As shown in Figure 9-23, your images are numbered starting with 0 in the upper left corner and, at zoom level 3, 31 in the lower right corner. The corresponding `tile` argument for these two requests would have `tile.x=0` and `tile.y=0` for your tile number 0, and `tile.x=8` and `tile.y=4` for your tile number 31, as shown in Figure 9-24.

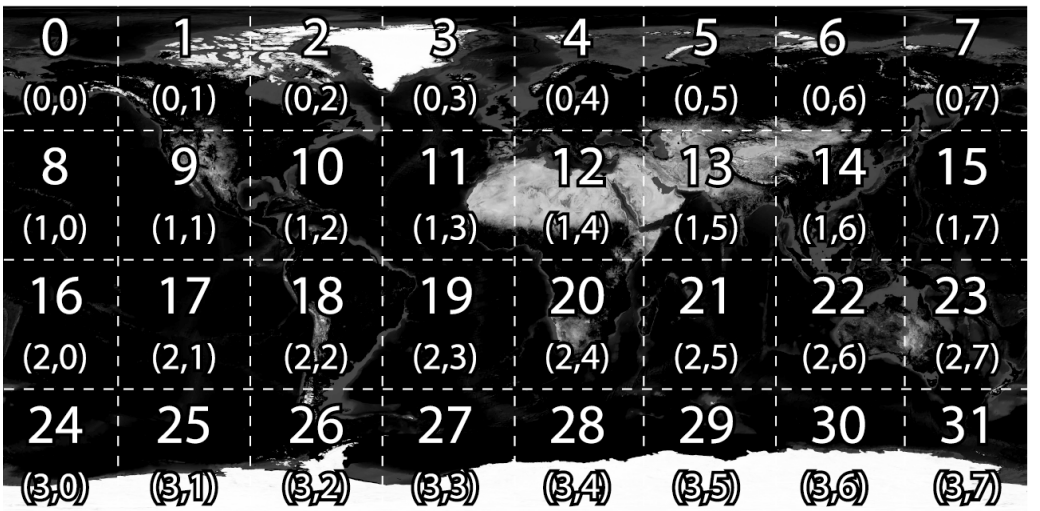


Figure 9-24. *Your tile numbering vs. Google's tile requests*

To convert the tile `x` and `y` values into your corresponding number scheme, you need to apply this simple formula:

```
x + y(2^zoom) = z
```

So the URL you return from `getTileUrl` should resemble this:

```
return 'tiles/image/tile.' + zoom + '.' + (tile.x + tile.y*Math.pow(2,zoom))➥
+ '.png';
```

where *image* is replaced by the name of each of the directories you created when making your tiles.

Along with your tiles, you may want to create one extra tile that shows when the map is at a zoom level that's too close for your tiles. For example, your three images don't all have the same resolution, so if you're looking at one map at zoom level 5 and then switch map types to the `land_ocean_ice_lights` image, it only goes up to zoom level 3, and the map will have nothing to display. Depending on your application, you could just display an image with a message indicating to the user "There are no tiles at this zoom level; zoom out for a broader look," or you could be a little more creative, like the creators of the moon map at `http://moon.google.com`. That map displays tiles of cheese when you zoom in too close, as shown in Figure 9-25.

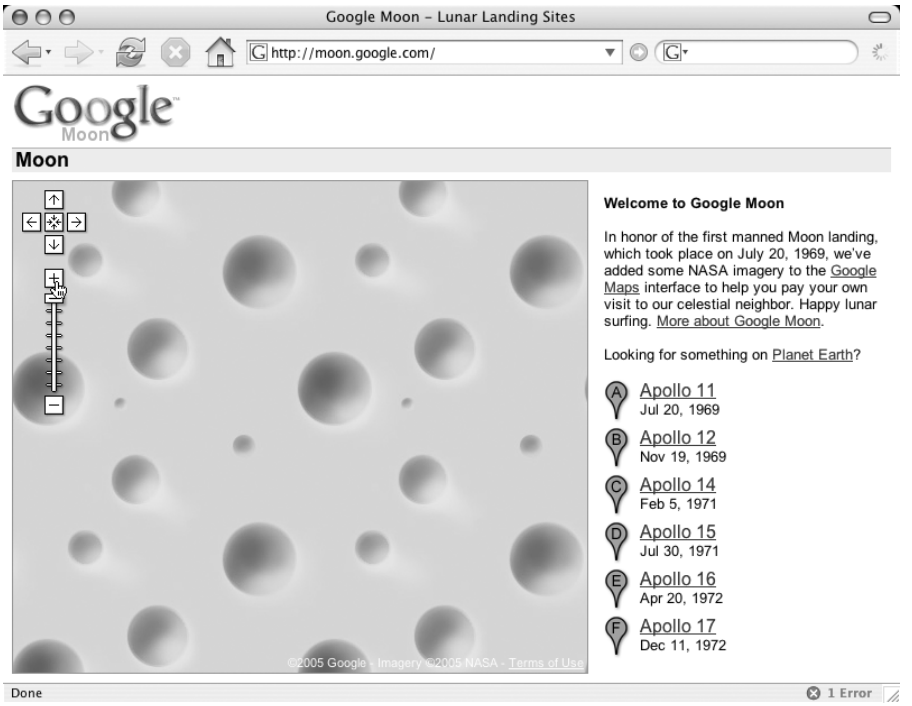


Figure 9-25. *Cheese on the moon when zoomed in too close at Google Moon*

To incorporate the "too close" image, simply check the zoom level before requesting the tiles for the `land_ocean_ice_lights` image and request the appropriate alternate tile:

```
if(zoom > 3) return 'tiles/no_tiles_at_zoom_level.png';
else return 'tiles/land_ocean_ice_lights/tile.' + zoom + '.' +
    (tile.x + tile.y *Math.pow(2,zoom)) + '.png';
```

Don't Forget the Copyright Credits

Remember that to use the images from the Visible Earth site, you must abide by the terms of use and give credit for the imagery to NASA and the Visible Earth team. To do so, you can easily add the appropriate copyright information to the tile layer using the `GCopyright` and `GCopyrightCollection` classes. If you use other images from different sources, you can add different or multiple copyrights to each tile layer. To do so, simply create a new `GCopyrightCollection` with an appropriate optional prefix:

```
copyrights = new GCopyrightCollection('Map Imagery:');
```

Then create a new GCopyright object, as per the arguments in Table 9-6, for the NASA Visible Earth team and add it to the copyright collection:

```
var visibleEarth = new GCopyright(
        'nasabluemarble',
        new GLatLngBounds(new GLatLng(-90,-180),new GLatLng(90,180)),
        0,
        '<a href="http://visibleearth.nasa.gov/">NASA Visible Earth</a>'
        copyrights.addCopyright(visibleEarth);
    );
```

Table 9-6. *GCopyright Input Arguments*

Argument	Type	Description
id	Integer	A unique identifier for this copyright information
minZoom	Integer	The lowest zoom level at which the copyright information applies
bounds	GLatLngBounds	The boundary of the map to which the copyright applies
text	String	The copyright message

Then when creating your GTileLayer objects for each image, pass copyrights into the tile layer as the first parameter to the GTileLayer class:

```
var BlueMarbleCloudyTiles = new GTileLayer(copyrights,0,5);
```

When your map loads, you should be able to see the credit in the copyright information in the lower right corner of the map, as shown in Figure 9-26.

Listing 9-9 includes the copyright credits plus the three completed tile layers, one for each image. The tile layers are named BlueMarbleTiles, BlueMarbleNightTiles, and BlueMarbleCloudyTiles, each representing one of the land_ocean_ice, land_ocean_ice_lights, and land_ocean_ice_cloud images, respectively. Also, when creating the tile layers, be sure to indicate the expected zoom levels using the second and third parameters to the GTileLayer class, so the API knows what zoom levels to expect.

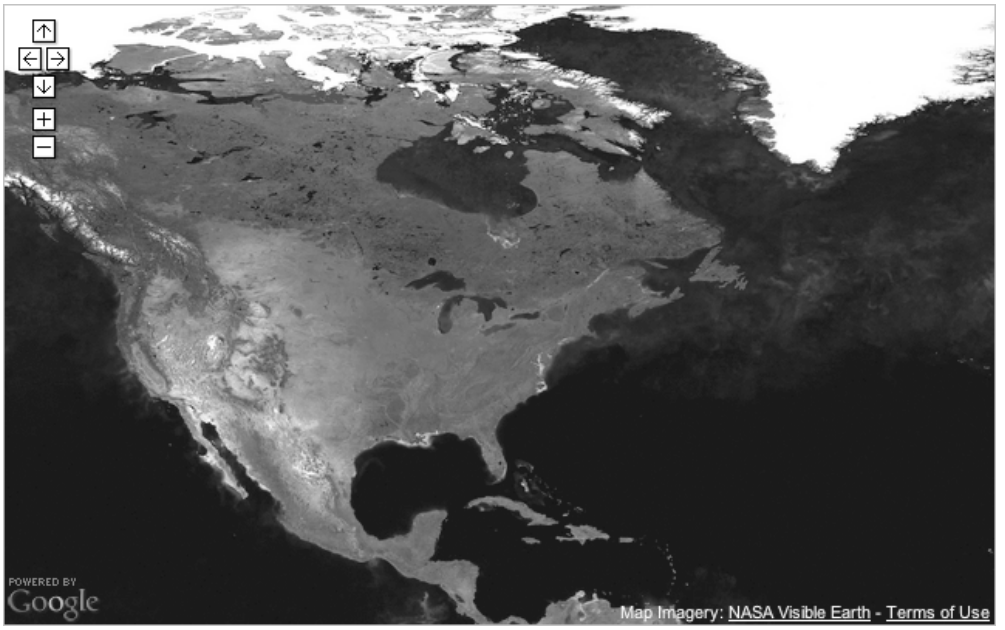


Figure 9-26. *Copyright information on the map (image courtesy of NASA Visible Earth)*

Listing 9-9. *Blue Marble Copyright Credits and Tile Layers*

```
copyrights = new GCopyrightCollection('Map Imagery:');
var visibleEarth = new GCopyright(
    'nasabluemarble',
    new GLatLngBounds(new GLatLng(-90,-180),new GLatLng(90,180)),
    0,
    '<a href="http://visibleearth.nasa.gov/">NASA Visible Earth</a>'
    copyrights.addCopyright(visibleEarth);
);

//tile layer for land_ocean_ice
var BlueMarbleTiles = new GTileLayer(copyrights,0,5);
BlueMarbleTiles.getTileUrl = function(tile,zoom){
    if(zoom > 5) return 'tiles/no_tiles_at_zoom_level.png';
    else return 'tiles/land_ocean_ice/tile.' + zoom + '.' +
        (tile.x + tile.y * Math.pow(2,zoom)) + '.png';
};
BlueMarbleTiles.isPng = function() { return true; }
BlueMarbleTiles.getOpacity = function() { return 1.0; }
```

```
//tile layer for land_ocean_ice_lights
var BlueMarbleNightTiles = new GTileLayer(copyrights,0,3);
BlueMarbleNightTiles.getTileUrl = function(tile,zoom){
    if(zoom > 3) return 'tiles/no_tiles_at_zoom_level.png';
    else return 'tiles/land_ocean_ice_lights/tile.' + zoom + '.' +
        (tile.x + tile.y * Math.pow(2,zoom)) + '.png';
};
BlueMarbleNightTiles.isPng = function() { return true; }
BlueMarbleNightTiles.getOpacity = function() { return 1.0; }

//tile layer for land_ocean_ice_cloud
var BlueMarbleCloudyTiles = new GTileLayer(copyrights,0,5);
BlueMarbleCloudyTiles.getTileUrl = function(tile,zoom){
    if(zoom > 5) return 'tiles/no_tiles_at_zoom_level.png';
    else return 'tiles/land_ocean_ice_cloud/tile.' + zoom + '.' +
        (tile.x + tile.y * Math.pow(2,zoom)) + '.png';
};
BlueMarbleCloudyTiles.isPng = function() { return true; }
BlueMarbleCloudyTiles.getOpacity = function() { return 1.0; }
```

The Blue Marble GMapType

Your last step is to assemble the `BlueMarbleProjection` and the three tile layers into their own map types. This is relatively straightforward, and you can follow the exact same process you used earlier in the chapter. Listing 9-10 contains the three map types: `BlueMarble` for the normal map, `BlueMarbleNight` for the city lights map, and `BlueMarbleCloudy` for the cloudy map.

Listing 9-10. *Blue Marble Map Types*

```
var BlueMarble = new GMapType(
    [BlueMarbleTiles],
    BlueMarbleProjection,
    'Blue Marble',
    {
        shortName:'BM',
        tileSize:256,
        maxResolution:5,
        minResolution:0
    }
);

var BlueMarbleNight = new GMapType(
    [BlueMarbleNightTiles],
    BlueMarbleProjection,
    'Blue Marble Night',
    {
        shortName:'BMN',
        tileSize:256,
```

```
        maxResolution:3,
        minResolution:0
    }
);

var BlueMarbleCloudy = new GMapType(
    [BlueMarbleCloudyTiles],
    BlueMarbleProjection,
    'Blue Marble Cloudy',
    {
        shortName:'BMC',
        tileSize:256,
        maxResolution:5,
        minResolution:0
    }
);
```

Using the New Blue Marble Maps

To use the new Blue Marble maps, you need to add them to your `GMap2` object using the `addMapType()` method:

```
map = new GMap2(document.getElementById("map"));
map.addMapType(BlueMarble);
map.addMapType(BlueMarbleNight);
map.addMapType(BlueMarbleCloudy);
```

After you add the new map type to the `GMap2` object, you'll see the new map type along with Google's map types, as shown in Figure 9-27.

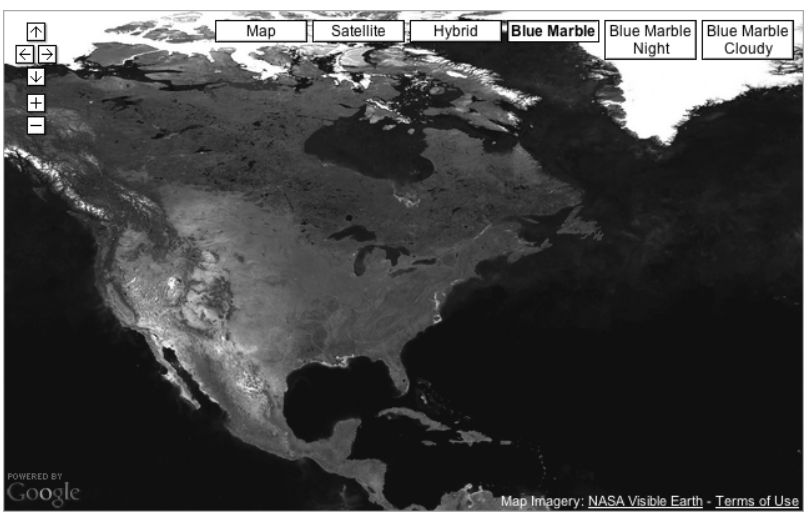


Figure 9-27. *The new map types on the map (image courtesy of NASA Visible Earth)*

If you want to show *only* the Blue Marble map types, just specify which map types to use when instantiating the `GMap2` object:

```
map = new GMap2(
    document.getElementById("map"),{
        mapTypes:[BlueMarble,BlueMarbleNight,BlueMarbleCloudy]
});
```

Now flipping back and forth between map types, you'll see the three different maps using the tiles you created. If you plot a point on the map, it will still appear in the correct location due to your new `BlueMarbleProjection`, as shown in Figure 9-28.

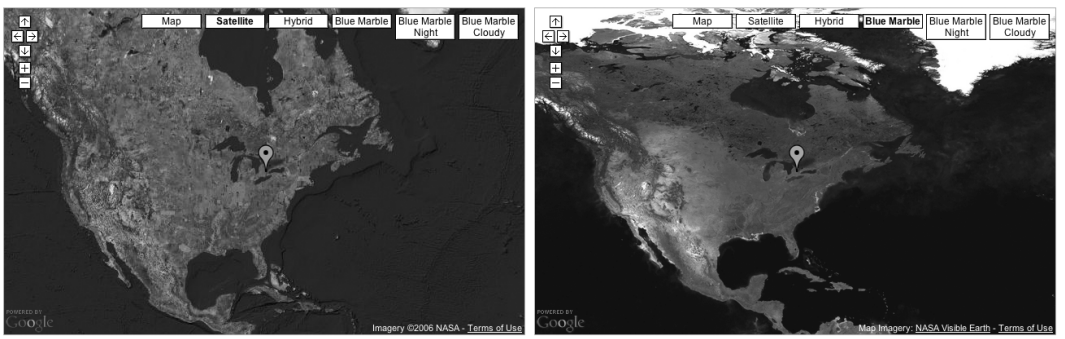


Figure 9-28. *Notice the difference in the locations plotted on a Google map (top) vs. the NASA Blue Marble map (bottom).*

Summary

In this chapter, you were introduced to some of the newer and more advanced features of the Google Maps API. Extending these examples further, you can create a wide variety of maps and controls that can do just about anything you want. For example, you could add a zoom control that works by clicking and dragging an area, or create fancy info windows incorporating QuickTime streams, Flash, or any other plug-ins. What you put into your own overlay objects is really up to you.

Using custom tiles, you could easily create your own map using an antique hand-drawn map from the early days of exploration, or you could use the API as a high-resolution image viewer by replacing the map tiles with tiles from your high-resolution images. You could even let people comment on parts of the image using the same techniques you saw in Chapter 3.

Whatever interesting things you decide to create, it's important to keep up to date with the API by checking the online reference at `http://www.google.com/apis/maps/documentation/reference.html`. Google is always updating, improving, and adding new features to the Google Maps API, so be sure to check back often. We also suggest that you join the Google Maps group at `http://groups.google.com/group/Google-Maps-API` and contribute any ideas you have to the Google Maps development team. Contributing back to the community will help it prosper,

and keeping up with the current topics and discussion will make you aware of all the latest additions. The group discussions also provide examples and neat ideas you might be able to use in your projects.

In the next chapter, you'll play with some other features of the API, such as using polylines, finding region perimeters, and calculating angles on the earth's surface.

CHAPTER 10

Lines, Lengths, and Areas

All of the projects we've presented have dealt with map markers as either individual entities or as related clusters. In this chapter, we'll demonstrate some of the other ways that groups of points may be interpreted and presented.

A group of points is just that: a group of points. But a string of points in sequence may represent a line or a path, which has the calculable property of *length*. Once the points form a closed loop, they may be treated as the outline of a region having *area*. Using the appropriate formulas, you can compute these distance and area values for your map projects.

The Google Maps 2.0 API includes some of this functionality, but remember that you may need to perform these kinds of calculations in your server-side scripts as well. With these mathematical tricks, as with any tools, it's good to at least have a vague understanding of their underlying principles, so you have the confidence to apply them correctly and trust their output.

In this chapter, you'll learn how to do the following:

- Compute the area and perimeter of an arbitrary region
- Calculate angles on the earth's surface
- Plot polygons in response to mouse clicks and allow draggable markers

Starting Flat

When you measure quantities such as length and area on a planet's surface, what you're really measuring are properties of three-dimensional figures. A region of any significant size plotted on the surface of the earth is not flat—it contains a bulge corresponding to the earth's curvature. This bulge *increases* the amount of area over what you might measure if you plotted a region of similar perimeter on a flat (planar) surface.

An important thing to realize, though, is that you can't just generalize that *plotting it on a sphere makes it bigger*, because the results actually depend on which method you use to translate the flat object to its spherical representation.

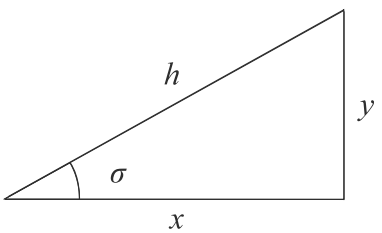
As an example, picture a gigantic circle drawn on the earth, big enough to encompass all of Australia. This circle has two key dimensions: radius and circumference. If you plot a flat circle with the same *circumference*, you'll find that its area is smaller than the one around Australia, since it doesn't have the bonus area from the earth's bulge. But if you plot a circle using the same *radius*, you'll find quite the opposite: the flat one has a larger surface area. Why?

Because forming the bulged circle is like taking a flat doily and rolling it into a cone. Even though the *surface* radius is the same, the cone has less surface area (since some of the doily folds over on itself).

Before we discuss how to compute these distance and area values, let's quickly review the classical Euclidean stuff that applies to flat shapes on a plane.

Lengths and Angles

A cornerstone of high-school geometry is the Pythagorean theorem. In a flat system, it allows you to quickly and accurately calculate the length of the diagonal on a right-angle triangle. In practical terms, this means that given any straight line drawn on a Cartesian (X and Y) coordinate system, you can independently measure the X and Y displacements from the start to the end of the line, and then use the theorem to get the length of the line itself. You can see in Figure 10-1 how this is applied.



a) $h = \sqrt{x^2 + y^2}$

b) $\sigma = \arctan(\frac{y}{x})$

Figure 10-1. *The Pythagorean theorem for length (a), arctangent for angle (b)*

Finding the length of a line is only half the story, though. To be able to fully describe a line, you need its length *and* its angle. And again, high school math has us covered. The arctangent (also *atan* or *arctan*) function takes the ratio of the Y and X displacements (the slope), and gives back an angle from horizontal (shown in Figure 10-1).

Most programming languages, however, go a step beyond, providing just basic arctangent and also providing an additional function, typically called `atan2()`. With `atan2()`, you pass in the Y and X displacements separately, and it will correctly compute the angle, in the range $-\pi$ to π. Plus, it will properly handle the vertical case. (Remember that a vertical line has undefined slope because its horizontal displacement is zero; anything divided by zero is *undefined*.)

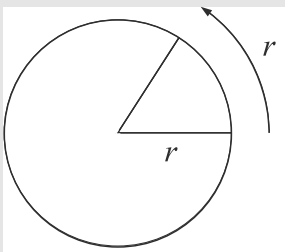
In JavaScript, this function takes the form of `Math.atan2()`.

RADIAN REFRESHER

You may be confused by some of the values that you get back from functions such as `Math.atan2()`. Keep in mind that JavaScript, like most programming languages, does all of its trigonometric operations using *radians*. Switching between radians and degrees is a straightforward operation. But radians are the favored unit for working with circles and other curves.

A radian is defined as one *radius length* around a circle's perimeter. Since the radius and circumference of a circle are directly proportional to each other, an angle measured in radians doesn't vary with the size of the circle.

Given a circle with radius r, we know from basic principles that its circumference is π times its diameter. So the circumference of that circle is 2πr. If we want to know what percentage of the perimeter a single radian represents, we can just divide one radian's distance (r) by the total distance (2πr). And from that, it's possible to see that a radian is a little less than one-sixth of a circle, as shown in the illustration.



$$\frac{r}{2\pi r} = \frac{x}{360^\circ}$$

$$x = \frac{360^\circ}{2\pi}$$

$$x \approx 57.3^\circ$$

Radians represent a ratio, and ratios have no units. When you write an angle in degrees, you must denote it with the small circle that represents degrees. After all, degrees are an arbitrary unit; the value 360 happens to work well for a circle simply because it divides cleanly in so many ways. Indeed, 180 degrees is equal to exactly π radians.

Of course, sometimes it will be important to make it perfectly clear that radians are the method of measurement. In that case, you can append "rad" to the value. But this is not a unit; it's simply an indication of what the number represents.

Here's a summary of the conversion calculations:

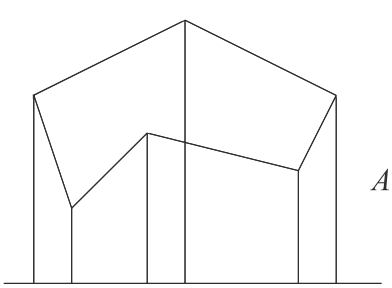
- To convert from radians to degrees, divide by π and multiply by 180.
- To convert from degrees to radians, multiply by π and divide by 180.

Areas

In computing the area of an arbitrary region, the human method would be to break it down into simple components, such as triangles, and then sum up the individual areas of these smaller pieces. A triangle's area is just half the base times the height, and solving for the height is possible given enough of the angles and side lengths.

Breaking down a complex shape can be a tricky task, however, even for a human. In order for a computer to be able to solve for the area of an arbitrary region, a systematic approach must be developed—one that a simpleminded JavaScript function can reliably apply in all situations. To derive such a method, we'll begin by representing each point around a figure's perimeter as a coordinate pair labeled x_1 and y_1, x_2 and y_2, and so on.

The initial step is to extend each vertex of the shape to the X-axis, and then picture each line segment as being part of a quadrilateral, involving two of the extension lines and a piece of the X-axis. You can see this developing in Figure 10-2.



$$A = \frac{1}{2}\sum_{i=0}^{N-1}(x_i y_{i+1} - x_{i+1} y_i)$$

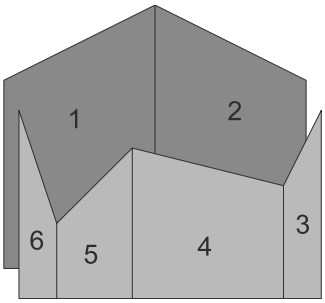


Figure 10-2. *Arbitrary concave polygon formula, showing the component quadrilaterals*

■ **Note** Concave and convex—which is which? *Convex* describes a shape where a straight line from any point inside the shape to any other point inside the shape will never leave the shape. *Concave*, on the other hand, means that there are areas where something has been "cut out" of the shape. These definitions apply equally to three-dimensional figures. A concave lens is one that narrows toward the middle, leaving a "cave" on either side of it.

From here, it's clear that if you take the areas of all the quadrilaterals on the *far* side of the shape and subtract the areas of those quadrilaterals on the *near* side, the area remaining is that of the shape itself.

To express this mathematically, we must use the summation operator Σ to add up the areas of the trapezoids, which are simply the average of their top and bottom lines (left and right, in our case), multiplied by the height:

$$A = \sum_{i=0}^{N-1} \frac{1}{2}(y_{i+1} + y_i)(x_{i+1} - x_i)$$

The business about adding the far-side area and subtracting the near-side area is actually one we get for free. Working under the assumption that the points are provided in clockwise order, subtracting the *x* values for the height ensures that the near and far regions have the opposite sign. The formula as given here assumes points provided in clockwise order. If you wish to accept them in either order, you can take the absolute value of the result.

This can be simplified if you bring the constant outside the summation, and expand the multiplication on the inside:

$$A = \frac{1}{2}\sum_{i=0}^{N-1} x_{i+1}y_{i+1} - x_i y_{i+1} + x_{i+1}y_i - x_i y_i$$

The observant will notice that once this summation is applied across a cyclical list of points, every $-x_{i+1}y_{i+1}$ term will *subtract out* the *xiyi* term in the following iteration of the sum. After this final simplification, you're left with a straightforward formula:

$$A = \frac{1}{2}\sum_{i=0}^{N-1}\left(x_{i+1}y_i - x_i y_{i+1}\right)$$

To implement this in JavaScript is a simple matter of a loop, as shown in Listing 10-1.

Listing 10-1. *Function for an Arbitrary Shape's Area, Given by a List of Coordinate Pairs*

```
var points = [
    {'x': 1, 'y': 4 }, {'x': 4, 'y': 6 }, {'x': 6, 'y': 5 },
    {'x': 5, 'y': 1 }, {'x': 3, 'y': 3 }, {'x': 2, 'y': 2 }
];

function calculateArea(points) {
    var count = points.length;
    var tally = 0;
    var i;

    // add the first point to the end of the array
    points[points.length] = points[0];

    for(i = 0; i < count; i++) {
        tally += points[i + 1].x * points[i].y
        tally -= points[i].x * points[i + 1].y
    }

    return tally * 0.5;
}
```

Caution The code in Listing 10-1 contains a "gotcha." JavaScript passes all nonprimitives by *reference*, which means the caller's copy of the `points` array will get back the duplicated version with the extra element tacked on the end. If this is important, you could call the array's `pop()` method to remove the final element.

You can see the Listing 10-1 code in action in Figure 10-3. Although it would work for highly localized regions, where the earth can be assumed flat, it's unsuitable as a general, global solution. You can see in the demo that we've used points plotted in pixel increments on your flat screen and then calculated the area inside those.

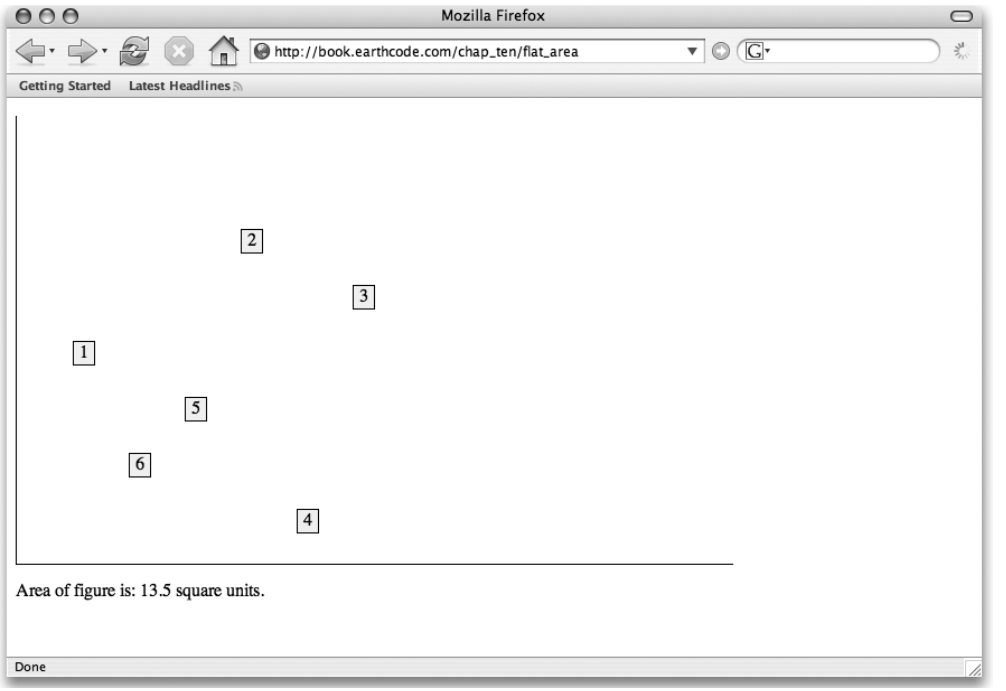


Figure 10-3. *Calculating with JavaScript the area encompassed by six flat points*

However, the formula is still important. For areas small enough to be approximated as flat, such a method is straightforward to apply and not difficult conceptually. It's helpful to see it in comparison to the spherical methods we'll develop in the next section.

Moving to Spheres

The study of spherical geometry is a field dominated by fascinating shortcuts and unusual ways of conceptualizing problems. Nothing from flat geometry can be simply applied verbatim, but there are interesting ways that aspects of spherical problems can be reduced down to planar ones.

The Great Circle

The shortest way to connect any two points on the surface of a sphere is by going *through* the sphere itself. In terms of surface routes, however, the shortest is called a *great-circle path*. It has this name because the connecting arc is *part of a circle that has the same center point as the sphere itself*, perfectly bisecting it. It's the largest possible circle that may be traced on the surface of any sphere.

All longitude lines are great circles, but of the lines of latitude, only the equator qualifies.

This can be counterintuitive at first, especially looking at maps such as the New York to Paris route in Figure 10-4. When trans-Atlantic flights fly great-circle routes through the northern hemisphere, it appears—from a flat map—as though they've taken a bizarre arctic detour. But, as we explained in Chapter 7, the farther away from the equator you look on a Mercator map, the more zoomed-in your scale is. A line through the northern Atlantic is actually traveling less distance, since the scale in that location is larger. A great-circle path, when looking at a globe, makes perfect sense.

Figure 10-4. *A great-circle route from New York to Paris, similar to what Charles Lindbergh followed on his famous 1927 hop across the Atlantic*

■ **Note** Modern New York to Paris flights likely wouldn't follow the exact path shown in Figure 10-4, but their reason for diverging from it would be to take advantage of the jet stream on eastbound flights.

In Figure 10-5, we work forward from that original shortest line—the one that joins two points by passing through the earth itself. If we imagine that we can't travel it directly, but must trace arc routes over the earth's surface, it becomes clear that the *largest* possible radius is what yields the shortest path between the points.

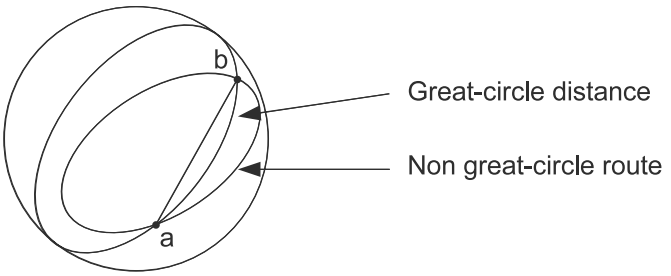


Figure 10-5. *The shortest path has the largest radius.*

Great-Circle Lengths

There are multiple possibilities for how to accurately calculate a great-circle distance between two points. We'll show two methods: the Cartesian method and the Haversine method. Both are considered very reliable. The Cartesian method is simpler to conceptualize. The Haversine method is easier to compute.

■ **Note** An article from the U.S. Census Bureau suggests that the Haversine method is the superior one in most cases. The piece has disappeared from its original location, but it has been mirrored at `http://www.movable-type.co.uk/scripts/GIS-FAQ-5.1.html`.

The Cartesian Method

Taking the great-circle idea and applying Euclidean geometry techniques, we can actually arrive at a perfectly valid formula for calculating the length of a great-circle path. The steps to this solution are as follows:

1. Using trigonometry and the radius of earth, transform each latitude/longitude pair into three-dimensional Cartesian coordinates.
2. Determine the distance between the two points by calculating *x*, *y*, and *z* displacements and then applying the Pythagorean theorem. (In three dimensions, it's exactly the same; just add the square of the Z-dimension under the root sign.)
3. Picture that distance as a cord on a "great circle" around the earth, and then using basic two-dimensional geometry, calculate the arc-length bracketed by the known straight-line length.

Although this method is accurate, unfortunately even in its simplified form, it's frighteningly complex:

$$\Delta\sigma = \arctan\left\{\frac{\sqrt{[\cos\phi_2 \sin\Delta\lambda]^2 + [\cos\phi_1 \sin\phi_1 - \sin\phi_1 \cos\phi_2 \cos\Delta\lambda]^2}}{\sin\phi_1 \sin\phi_2 + \cos\phi_1 \cos\phi_2 \cos\phi\Delta\lambda}\right\}$$

For this reason, we turn to the Haversine formula, a non-Euclidean solution to the problem.

The Haversine Method

As with many of the mathematical tools we've used with the Google Maps API, the Haversine formula has a history with marine navigation. Although both work perfectly well, the Haversine formula has an elegant simplicity that makes it appealing. Indeed, as of version 2.0, the functionality you see here is provided in the Google Maps API, by the `GLatLng::distanceFrom()` method. Here is the Haversine formula:

$$d = 2\arcsin\left(\sqrt{\sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos_1 \cos_2 \sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

You get surface distance by plugging in the two points' latitudes and longitudes into f_1, l_1 and f_2, l_2, respectively, and then multiplying *d* by the radius of the earth, 6,378,137 meters.

JavaScript exposes all of the mathematical functions required to implement an expression such as this in the `Math` object.

Tip An excellent resource for the `Math` object can be found at W3Schools: `http://www.w3schools.com/jsref/jsref_obj_math.asp`.

IS THE EARTH FLAT AFTER ALL?

It comes as a surprise to some that the earth is not *perfectly* spherical. It's flattened slightly, a shape known to mathematicians as an *oblate spheroid*.

At the equator, the earth has a radius of 6,378 kilometers. Measuring from the center to the poles, however, the distance is slightly less—about 6,357 kilometers. For some types of calculations, it's appropriate to use 6,371 kilometers, which is the radius of a theoretical sphere having the same *volume* as the earth.

All of the formulas presented in this chapter, however, operate under the assumption that the earth *is* a sphere, having a radius of exactly 6,378,137 meters. This is, in fact, the same assumption made by the functions in version 2.0 of the Google Maps API, so any slight errors will be in good company.

Area on a Spherical Surface

Our formula from Listing 10-1 operates given a method for computing trapezoidal areas. In order to adapt this method to spherical geometry, we would need to establish a way of computing the area of a trapezoid that is now drawn on the surface of a sphere. But first, let's look at a slightly simpler problem: how to compute the area of a spherical triangle.

A Spherical Triangle

Given three points on the surface of a sphere, it's possible to join them by great-circle arcs and then determine the surface area contained within the area. The process for doing this is an intriguing one, as it's based not around three-sided figures but two-sided ones.

On a flat piece of paper there is no such thing as a two-sided figure. From lines, we make the jump directly to triangles. But on a curved surface, there *is* a two-sided shape, as you can see in Figure 10-6. It's called a *lune*, and it's the orange slice carved out when two noncoincident great circles exist in the same sphere together.

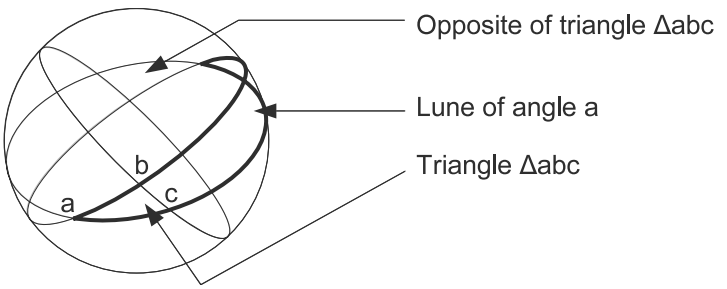


Figure 10-6. *A spherical triangle abc, with the lune formed by angle a highlighted*

Since both curves involved are great circles, determining the surface area of a lune is almost trivial:

$$A = 4\pi r^2\left(\frac{a}{2\pi}\right) = 2r^2 a$$

It's simply the percentage of the sphere's total surface area that the angle *a* is of a full circle (2π, in radians).

Although Figure 10-6 has only one lune highlighted, if you look closely, you can see that there are actually six of them. Each of the points *a*, *b*, and *c* is the endpoint of two opposite lunes: one that encompasses the *abc* triangle, and a second one that includes not the *abc* triangle but the "shadow" *abc*. The key to finding the area of the triangle *abc* is to realize that the surface areas of the six lunes can be summed to get an area that is the sphere's total surface area, plus the *abc* area four extra times:

$$A_{sphere} = A_a + A_b + A_c + A_{a^1} + A_{b^1} + A_{c^1} - 4A_T$$

Remember that the triangle is there twice and there are six lunes, each of which includes the triangle's area once. The area must be subtracted four times in order to get back to just the plain old surface area. After substituting the lune surface area formula and rearranging, we end up with the following formula:

$$4A_T = 4r^2a + 4r^2b + 4r^2c - 4\pi r^2$$

A final factoring leaves us with this simpler formula:

$$A_T = r^2(a + b + c - \pi)$$

Of course, this is not a formula that works from latitudes and longitudes. This still assumes we have the angles between the triangle lines.

Given how simple this formula is, it might be disappointing to discover just how complex a process it is to find the interior angles of the triangle—the *a*, *b*, and *c* values. We must express each of the three points as a vector, so that the surface point *a* becomes a Cartesian vector *A*, pointing from the center of the sphere to the location of *a*. Having these three vectors, the angle at *a* in the original triangle can be determined by the following expression of cross products and dot products (see the next section for a refresher on these vector operations):

$$a + \arctan\left(\frac{(B \times A) \bullet C}{(A \times C) \bullet (A \times B)}\right)$$

Tip For an explanation of the derivation of the expression of cross products and dot products, see `http://www.ral.ucar.edu/research/verification/randy/writeups/earthareas.pdf`.

This is giving us the angle we need, but it's still not starting from latitudes and longitudes. Converting latitudes and longitudes to Cartesian coordinates is not difficult given a pen and a few minutes to mull it over, and we've included the shortcut here. If the latitude and longitude of a point are known, and in *f* and *l*, then the three components of its vector are as follows:

$$x = \cos\phi \sin\lambda$$
$$y = \cos\phi \cos\lambda$$
$$z = \sin\phi$$

Listing 10-2 shows a JavaScript function that can perform this conversion directly from a `GLatLng` object. Notice how it uses radians, since these are the units of the JavaScript trigonometry functions. We don't have a designated class for storing three-dimensional vectors, so we'll simply return it as an array of the three elements. (Creating such a class would be a worthwhile endeavor if you were to venture too much further into this territory.)

Listing 10-2. *Cartesian Coordinates from Latitude and Longitude*

```
function cartesianCoordinates(latlng) {
    var x = Math.cos(latlng.latRadians()) * Math.sin(latlng.lngRadians());
    var y = Math.cos(latlng.latRadians()) * Math.cos(latlng.lngRadians());
    var z = Math.sin(latlng.latRadians());
    return [x, y, z];
}
```

Given these coordinates for each of the three points involved, it's just a matter of a quick refresher on how to do vector cross products and dot products, and then we'll have everything we need to cleanly implement the angle formula. And having the angles, we can find our area.

Vector Operations: Dot Products and Cross Products

When dealing with the "multiplication" of three-dimensional vectors, there are actually two separate operations that can be performed. The first of these yields a scalar (nonvector) value and is called the *dot product*. To compute the dot product, you multiply the *x* of the first vector by the *x* of the second one, and then add that to the product of the two *y* values and the product of the two *z* values.

To see how this works, we'll simply show you our JavaScript implementation in Listing 10-3, which takes two arguments, each of which is assumed to be a three-element array representing a vector.

Listing 10-3. *Function for Calculating a Dot Product*

```
function dotProduct(a, b) {
    return (a[0] * b[0]) + (a[1] * b[1]) + (a[2] * b[2]);
}
```

The other type of vector multiplication returns another vector as its result. This is called the *cross product*, and the resulting vector has the geometric property of being perpendicular to the two initial vectors. Our function for calculating the cross product is shown in Listing 10-4.

Listing 10-4. *Function for Calculating the Cross Product*

```
function crossProduct(a, b) {
    return [(a[1] * b[2]) - (a[2] * b[1]), (a[2] * b[0]) - (a[0] * b[2]),➥
           (a[0] * b[1]) - (a[1] * b[0])];
}
```

Now we can assemble a final solver for a given angle, as in Listing 10-5.

Listing 10-5. *Function for the Angle Between Three Points on a Sphere*

```
function spherePointAngle(A, B, C) { // returns angle at B
    return Math.atan2(dotProduct(crossProduct(C, B), A),➥
        dotProduct(crossProduct(B, A), crossProduct(B, C)));
}
```

And now we have all the pieces to solve for the area of a spherical triangle. Before we get to work on that, though, there's an important thing you should know.

An Extension to Arbitrary Polygons

As it turns out, the triangle formula that we showed at the beginning of this is actually just the *n*=3 case of a general formula for shapes traced on spheres:

$$A = r^2\left[\left(a_1 + a_2 + \ldots + a_n\right) - (n-2)\pi\right]$$

The *a* terms represent the angles at the vertices involved in the shape, and *n* represents the number of vertices.

From a planar geometry perspective, it seems absurd that you would be able to calculate a surface area having only angles and no lengths. But a simple thought experiment can help you persuade yourself that this works. Try to picture the smallest triangle you could draw on a sphere, and then picture the largest.

The smallest triangle is so small that the area it covers is considered flat. As a triangle on a plane, its angles must sum to 180 degrees. But the largest triangle—well, the largest possible joining of three line segments on a sphere—is going to have them all going end to end in a circle *around* it. That is, the "triangle" is simply tracing out a *great circle*, with its area being half the sphere's surface, and the three angles totaling 3 * 180° = 540°.

Clearly, there's a relationship between the total of the angles and the percentage of the sphere covered. And the general formula, derived from the Gauss-Bonnet theorem, expresses this relation.

In Listing 10-6, we've built a general function for determining the area inside a list of points, given as a list of `GLatLng` objects.

Listing 10-6. *General-Purpose Function for Determining Area Inside a List of Points*

```
var earthRadius = 6378137; // in meters

function polylineArea(latlngs) {
    var id, sum = 0, pointCount = latlngs.length, cartesians = [];
    if (pointCount < 3) return 0;

    for (id=0; id < latlngs.length; id++) {
        cartesians[id] = cartesianCoordinates(latlngs[id]);
    }

    // pad out with the first two elements
    cartesians.push(cartesians[0]);
    cartesians.push(cartesians[1]);

    for(id = 0; id < pointCount; id++) {
        var A = cartesians[id];
        var B = cartesians[id + 1];
        var C = cartesians[id + 2];
        sum += spherePointAngle(A, B, C);
    }

    var alpha = Math.abs(sum - (pointCount - 2) * Math.PI);
    alpha -= 2 * Math.PI * Math.floor(alpha / (2 * Math.PI));
    alpha = Math.min(alpha, 4 * Math.PI - alpha);

    return Math.round(alpha * Math.pow(earthRadius, 2));
}
```

To test whether this is working properly, you could pick your favorite rectangular state, plug its corner coordinates into the function, and check whether the returned value corresponds to the established measurements.

Working with Polylines

You've seen a bunch of nifty geometric qualities that you can calculate given groups of points. But it's time we took this code on the road and got it integrated with some working maps. This section's project, shown in Figure 10-7, lets the user input polygon corners, then displays the perimeter and area of the region.

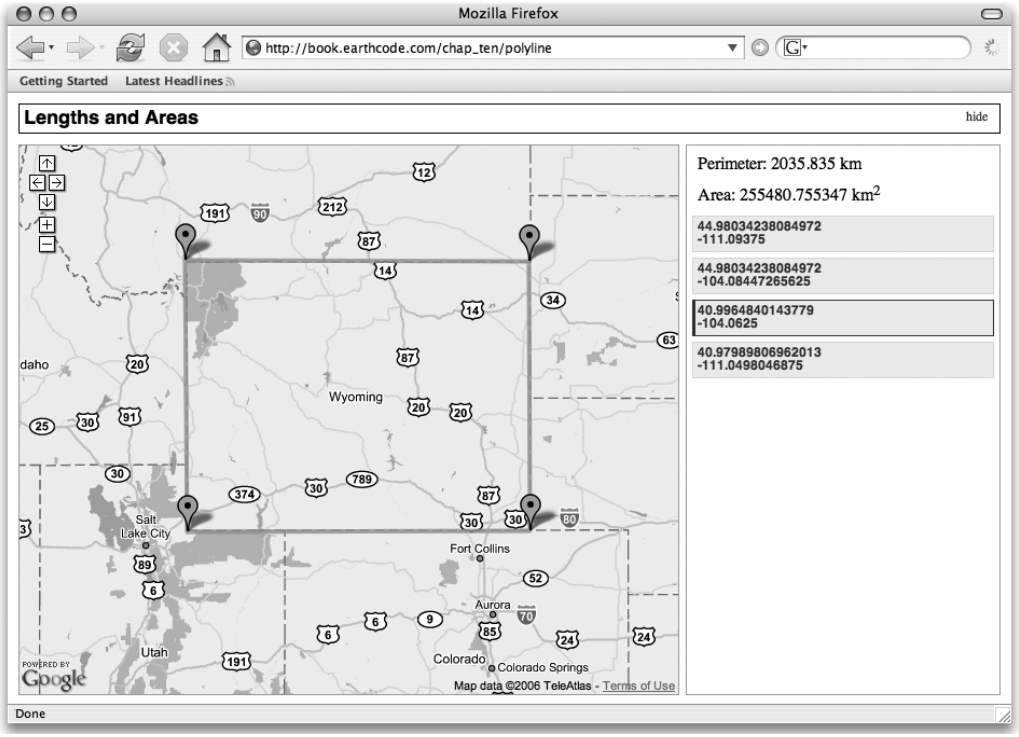


Figure 10-7. *The outline of Wyoming*

Building the Polylines Demo

Our starting setup will be pretty familiar from previous chapters. For markup and styles, establish a basic view with a header and flanking sidebar, as shown in Listing 10-7.

Listing 10-7. *app/views/chap_ten/polyline.rhtml*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:v="urn:schemas-microsoft-com:vml">
<head>
  <script src="http://maps.google.com/maps?file=api&amp;v=2&amp;key=YOUR_KEY_HERE"➥
type="text/javascript"></script>
  <%=javascript_include_tag 'prototype','application_10_2'%>
  <%=stylesheet_link_tag 'style_new'%>
  <!--[if IE]>
  <style type="text/css"> v\:* { behavior:url(#default#VML); }</style>
  <![endif]-->
</head>
```

```
<body id="body">
  <div id="toolbar">
    <h1>Lengths and Areas</h1>
    <ul id="sidebar-controls">
      <li><%=link_to_function 'hide',"Element.addClassName('body','sidebar-off')",➥
{:id=>'button-sidebar-hide'}%></li>
      <li><%=link_to_function 'show',"Element.removeClassName('body',➥
'sidebar-off')", {:id=>'button-sidebar-show'}%></li>
    </ul>
  </div>
  <div id="content">
    <div id="map-wrapper">
      <div id="map"></div>
    </div>
    <div id="sidebar">
      <div id="line-info">
        <p><span id="length-title">Length</span>: <span id="length-data">0➥
</span> km</p>
        <p>Area: <span id="area-data">0</span> km<sup>2</sup></p>
      </div>
      <ul id="sidebar-list">
      </ul>
    </div>
  </div>
</body>
</html>
```

You can see we've added an extra XML namespace, plus a bizarre proprietary style rule contained inside a *conditional comment*. This is a special Microsoft HTML comment that reliably hides the rule from all non-Internet Explorer browsers (see `http://msdn.microsoft.com/workshop/author/dhtml/overview/ccomment_ovw.asp`). Including this rule is a prerequisite to using the `GPolyline` class, if we want our polylines to work in Internet Explorer.

Why such requirements? To render polylines on Internet Explorer, Google Maps uses Vector Markup Language (VML), an XML vector language that was ahead of its time, and sadly never got included in browsers other than Internet Explorer. For nonsupporting user agents, the API simply has Google's servers render a PNG image, which gets draped over the map. In some cases, it will try to render the polyline using Scalable Vector Graphics (SVG), a contemporary standard that occupies the same space VML once did.

We could always stick the VML rule in with all the other rules in our main `style.css` file, but because it's not standard, we should keep it separate and away from browsers that might choke on it. (Generally, it's considered good CSS practice to keep any filters or "hack" style rules separated from the main flow of the style sheet.)

The styles used in this demo are lifted verbatim from the demos in prior chapters.

And, as for the JavaScript, well, a lot of it is similar to what you've seen before, but we've made some changes, too, which are highlighted in the next few listings, starting with Listing 10-8.

Listing 10-8. *Initialization Function in application.js, Containing a GEvent Call*

```
var map;
var centerLatitude = 40.6897;
var centerLongitude = -95.0446;
var startZoom = 5;
var deselectCurrent = function() {};
var removePolyline = function() {};
var earthRadius = 6378137; // in metres

var latlngs = [];

function init() {
        $('button-sidebar-hide').onclick =➥
        function() { return changeBodyClass('sidebar-right', 'nosidebar'); };
    $('button-sidebar-show').onclick =➥
        function() { return changeBodyClass('nosidebar', 'sidebar-right'); };

    handleResize();

    map = new GMap2($("map"));
    map.setCenter(new GLatLng(centerLatitude, centerLongitude), startZoom);
    map.addControl(new GSmallMapControl());

    GEvent.addListener(map, 'click', handleMapClick);
}
```

Most of this should be familiar to you from earlier chapters, including the one line that attaches a click handler to the map object. But, of course, we can't just reference a map click handler function and not show it to you.

The `handleMapClick()` function is designed to build up a list of `GLatLng` objects in an array, and on each new one added, redraw a polyline that connects the lot. Check it out in Listing 10-9.

Listing 10-9. *Handler for Map Clicks, in application.js*

```
function handleMapClick(marker, latlng) {
    if (!marker) {
        latlngs.push(latlng);
        initializePoint(latlngs.length - 1);
        redrawPolyline();
    }
}
```

This function is not a tricky one. It just adds the new `GLatLng` to the accumulating array, initializes the new point, and then has a second function redraw the polyline that connects all the points. So what are the functions `initializePoint()` and `redrawPolyline()`?

The venerable `initializePoint()` function has undergone some slight renovations from previous versions, but large chunks of it will remain familiar in Listing 10-10. The biggest change is that a new `draggable` parameter has been enabled, so that we can move our markers around once they're down on the map.

Listing 10-10. *Function for Initializing Individual Points from a Global Array*

```
function initializePoint(id) {
    var marker = new GMarker(latlngs[id], { draggable:true });
    var listItem = document.createElement('li');
    var listItemLink = listItem.appendChild(document.createElement('a'));
    listItemLink.href = "#";
    listItemLink.innerHTML = '<strong>' + latlngs[id].lat() +➥
'<br />' + latlngs[id].lng() + '</strong>';

    var focusPoint = function() {
        deselectCurrent();
        listItem.className = 'current';
        deselectCurrent = function() { listItem.className = ''; }
        map.panTo(latlngs[id]);
        return false;
    }

    GEvent.addListener(marker, 'click', focusPoint);
    listItemLink.onclick = focusPoint;
    $('sidebar-list').appendChild(listItem);

    map.addOverlay(marker);

    marker.enableDragging();
    GEvent.addListener(marker, 'dragend', function() {
        listItemLink.innerHTML = '<strong>' + latlngs[id].lat() +➥
'<br />' + latlngs[id].lng() + '</strong>';
        latlngs[id] = marker.getPoint();
        redrawPolyline();
    });
}
```

You can see now why it is important to keep `initializePoint()` and `redrawPolyline()` as separate entities—so that a dragged marker could *also* trigger a redrawing of the polyline. Speaking of redrawn polylines, let's take a peek at the `redrawPolyline()` function in Listing 10-11.

Listing 10-11. *Function to Redraw a Polyline from a Global Array*

```
function redrawPolyline() {
    var pointCount = latlngs.length;
    var id;

    map.removeOverlay(polyline)
    // Plot polyline, adding the first element to the end, to close the loop.
    latlngs.push(latlngs[0]);
    var polyline = new GPolyline(latlngs, 'FF6633', 4, 0.8);
    map.addOverlay(polyline);

    // Check total length of polyline (length for 2 points, perimeter > 2 points)
    if (pointCount >= 2) {
        var length = 0;
        for(id = 0; id < pointCount; id += 1) {
            length += latlngs[id].distanceFrom(latlngs[id + 1]);
        }

        if (pointCount > 2) {
            $('length-title').innerHTML = 'Perimeter';
            $('length-data').innerHTML = Math.round(length) / 1000;
        } else {
            $('length-title').innerHTML = 'Length';
            $('length-data').innerHTML = Math.round(length) / 2000;
        }
    }
    latlngs.pop(); // restore the array to how it was

    // Show value of area in square km.
    if (pointCount >= 3) {
        $('area-data').innerHTML = polylineArea(latlngs) / 1000000;
    }

}
```

This function may be long, but it's mostly just a sequence of mundane tasks: pad the list of points, remove the old polyline, draw the new polyline, iterate through to check length, and call our previous function to check area.

PUTTING THE GOOGLE GEOCODER TO WORK

Back in Chapter 4, we mentioned that the Google geocoder is accessible not just through a REST web service, but also directly from the JavaScript API. The polylines project in this chapter is a perfect example of a good use of this tool.

Rather than forcing users to enter points by clicking, we can provide a friendly text box that allows them to search for locations instead. The code required for this feature is not hard. The more important thing to understand is the two different mechanisms you could use to implement this feature:

- The user would submit the search box back to *your* server, and you would send out the REST request to Google, cache the response, and send out the result to your user. If the user decided to save or bookmark that point for later retrieval, you would already have it geocoded from the first request.
- Using the JavaScript geocoder, the user's address query is submitted directly to Google, and the geocoded point is sent straight to the user's browser, without your own server as the broker between them. This means better response time for the user, but also that when the user saves that point, you need to send back the coordinates so the point doesn't need to be re-geocoded on each future request.

To add this to the polyline application of the chapter, you would need to add an input for the search box. We'll use Rails' `button_to_function` helper, which generates a button with an `onClick` event calling the function we specify:

```
<div id="sidebar">
    <div id="line-info">
      <p><span id="length-title">Length</span>: <span id="length-data">0➥
</span> km</p>
      <p>Area: <span id="area-data">0</span> km<sup>2</sup></p>
    </div>
    <p>
    <input type="text" id="s" name="s" />
       <%=button_to_function "Add", "handleSearch()"%>
    </p>
    <ul id="sidebar-list">
    </ul>
</div>
```

And finally, the `handleSearch()` function contains the meat of calling the Google `GClientGeocoder` object. The `GClientGeocoder` object needs to be instantiated before the first use, but apart from that, it really couldn't be simpler: call its `getLatLng()` method, pass in the address string, and pass it a function to execute upon receiving the response. We've bolded the response function in the following listing, so you can see more clearly how it gets passed in:

```
function handleSearch() {
    var searchText = $('s').value;
    if (searchText == '') {
        alert('Please enter a location to search for.');
        return false;
    }
```

```
    if (!geocoder) geocoder = new GClientGeocoder();  // initialize geocoder

    geocoder.getLatLng(searchText,
        function (response) {
            if (!response) {
                alert('Error geocoding address');
            } else {
                latlngs.push(response);
                initializePoint(latlngs.length - 1);
                redrawPolyline();
                $('s').value = ''; // clear the search box
            }
        }
    );

    return false;
}
```

This example is a great opportunity to show you how the `GClientGeocoder` object works, since it's a case where the application is directly geocoding an address input by the user. It's important to realize that in any case where addresses are being sent from your server, you should geocode them on the server. But if you're receiving an address from the user, it's great to code it in JavaScript and then cache the location from there.

To see the modified version of this in action, check it out at `http://book.earthcode.com/chap_ten/client_geocoder/`.

Expanding the Polylines Demo

We wanted to leave you with an example that really brims with possibilities. What could you do to expand this? Well, we've already implemented a search box where users can type in addresses to be geocoded and added to the sequence. Besides cleaning that up and clarifying its function for the user, here are a few other ideas to get you started:

- Add a way to remove points from the list.
- Find an elegant way to insert points *into* the list, rather than just assuming the user wants them at the end of it.
- Try setting up the right sidebar so that the points can be dragged up and down to reorder the list. (Sam Stephenson's Prototype library could help you out with this; see `http://prototype.conio.net/`.)

Plus, of course, what good is it as a tech demo? What kind of use could this be put to in the wild? Property markings, perhaps? For a real estate agent, it would be valuable to plot out lots on a map, particularly those spacious ones where it's important that buyers see *just how deep* the backyard is. In fact, it's applicable for boundaries of all kinds. When the Blue Team gets from the lake to the dining hall, and the Red Team gets from the path up to the service road, who defends more territory, and who has farther to search for the flag? When the phone company moves the rural area codes around, which zones are the largest?

What About UTM Coordinates?

Readers who own or have used GPS devices will know that a latitude/longitude pair is not the only way to describe a global position. Typical handheld units will also provide UTM coordinates, an *easting* and a *northing*, both in units of meters. What is UTM, and how come Google instead chose latitude and longitude for its mapping system?

UTM stands for *Universal Transverse Mercator*. It's a projection system designed by the U.S. Army shortly before World War II. The primary purposes of UTM were to be highly accurate in close detail and to be a good enough flat approximation that accurate distance readings could be taken off a map, using nothing more than the Pythagorean theorem.

So how does it work? The UTM system begins by dividing the earth into narrow slices, each just 6 degrees wide. Each of these slices is then divided into 60 vertical zones, between 80° S and 84° N. As you can see in Figure 10-8, UTM has coverage of all land masses (except for inland Antarctica). Then—and here's the genius of it—each of these trapezoidal zones is presented in a *transverse* Mercator projection—it's Mercator, except rotated 90 degrees. So instead of seeing distortion as you move farther north and south, you see it as you move east and west. Yet, because the slices are so thin, the distortion is never more than 0.1% anywhere within a zone.

If you've ever seen a government topographic map, you've seen the UTM military gridlines on it. Depending on the zoom level, each box might represent a single square meter or some multiple of meters. But they *are* perfectly square boxes, and that makes standard planar trigonometry (as described in the first section of this chapter) "work" using UTM.

You can see from this explanation how ill-suited UTM would be for a global system such as Google Maps. Although calculations *within* a particular zone are made very straightforward by the system, it isn't at all appropriate for performing larger-scale calculations that would span multiple zones.

Indeed, there are special cases of UTM that illustrate very clearly the workarounds caused by this limitation. Zone 32V, which covers southwest Norway, is arbitrarily extended west to a total width of 9 degrees. This is so that it can contain the entire tip of the country and not leave a sliver alone in the otherwise empty zone 31V.

Latitude/longitude is an extremely general system. With no special cases or strange exceptions, it simply and predictably identifies any spot on the globe, and only the most basic knowledge of a protractor is required in order to "get it." UTM is a highly specialized system, designed for taking pinpoint measurements on detailed topographic maps.

It's the general system that's appropriate for the global Google Maps. But the next time you're camping with your GPS and want to plot out a trail, try switching it to UTM mode. You may find that at that level of detail, having simple readings in meters makes the system much more accessible for basic, planar geometric calculations.

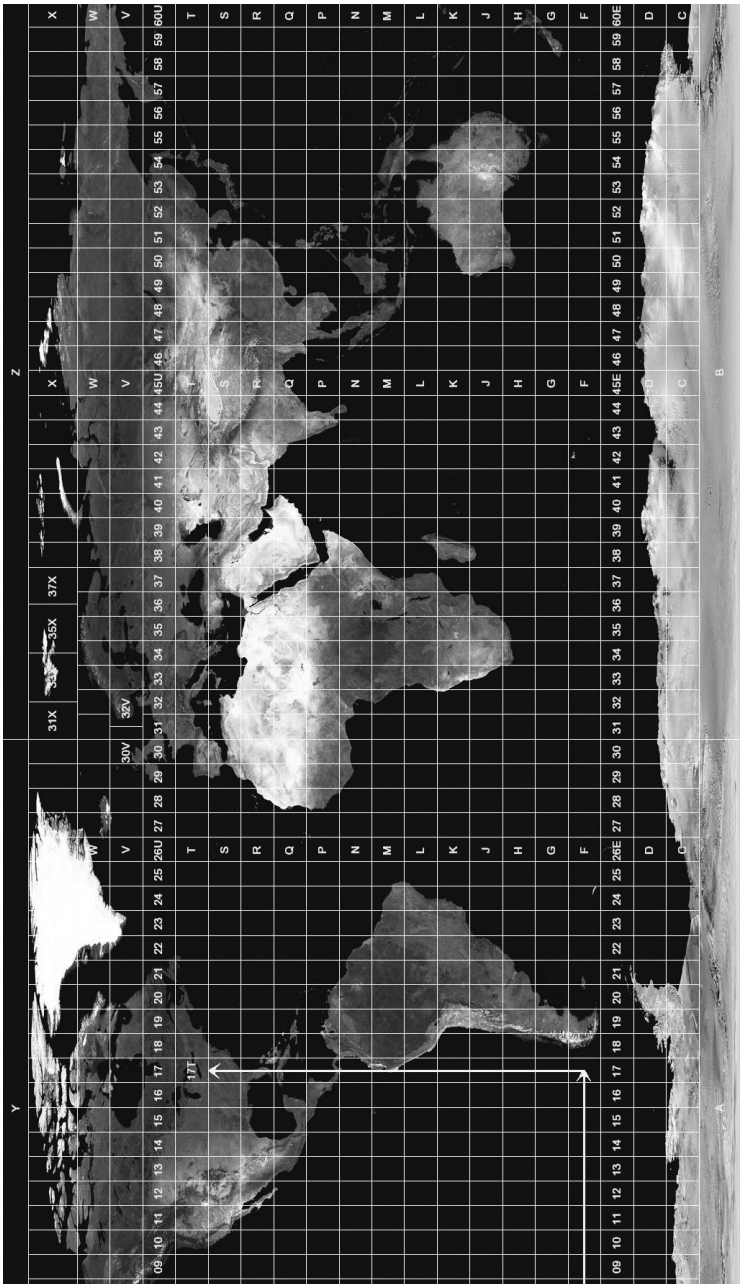


Figure 10-8. *The zones of the UTM projection system*

Running Afoul of the Date Line

In *Around the World in Eighty Days*, protagonist Phileas Fogg accounts for one extra day than his associates back home in London. In making his journey, each 15 degrees traveled east moved him one time zone *earlier*; for each zone crossed, the particular day counted was being shortened by an hour. Throughout the 80-day journey, he had logged days not as 24-hour periods, but as a sunset following a sunrise. In crossing the Pacific Ocean—and the International Date Line—he "gained" an extra day.

We aren't traveling around the world, but the International Date Line has a few implications on map making with the Google API.

When you speak of degrees in a circle, you nearly always think in terms of all 360. A bearing of *due south* is expressed as 270 degrees, not as –90 degrees. Within circles, we think of angles as having a range from 0 to 360.

Well, with degrees of longitude, it's not from 0 to 360. It's –180 to 180. Measuring from the prime meridian at Greenwich, England, degrees eastward are positive and degrees westward are negative. So if Greenwich, is where the 0th degree is, that means there's a point opposite, where the 180th and –180th degrees meet. That line is the International Date Line.

Curiously enough, the International Date Line has no official path that it takes in its deviations from exactly 180 degrees. The countries through which it might pass simply declare themselves to be on the east or the west of it, and it becomes the responsibility of individual cartographers to weave the line between them accordingly. Generally, however, the line divides the Bering Strait (separating Russia from Alaska), and goes down through Oceania with Hawaii and French Polynesia on the eastern side, and nearly everything else on the western side, including New Zealand, Fiji, and the Marshall Islands.

How does this affect Google Maps? The 2.0 API is surprisingly well-equipped to handle International Date Line oddities. Google uses imagery, creating an infinite equator and correctly simulating the continuous nature of a sphere. In Figure 10-9, the two maps are set to be in identical positions, each with a marker in Toronto. The one in front is panned *right* until the marker jumps from one Toronto to the next.

The system isn't foolproof (it moves markers correctly, but it doesn't, at the moment, move the info window), but it's just another one of those trade-offs you deal with for being able to view our round globe through such a conveniently flat medium. And it's pretty elegant, even for a trade-off.

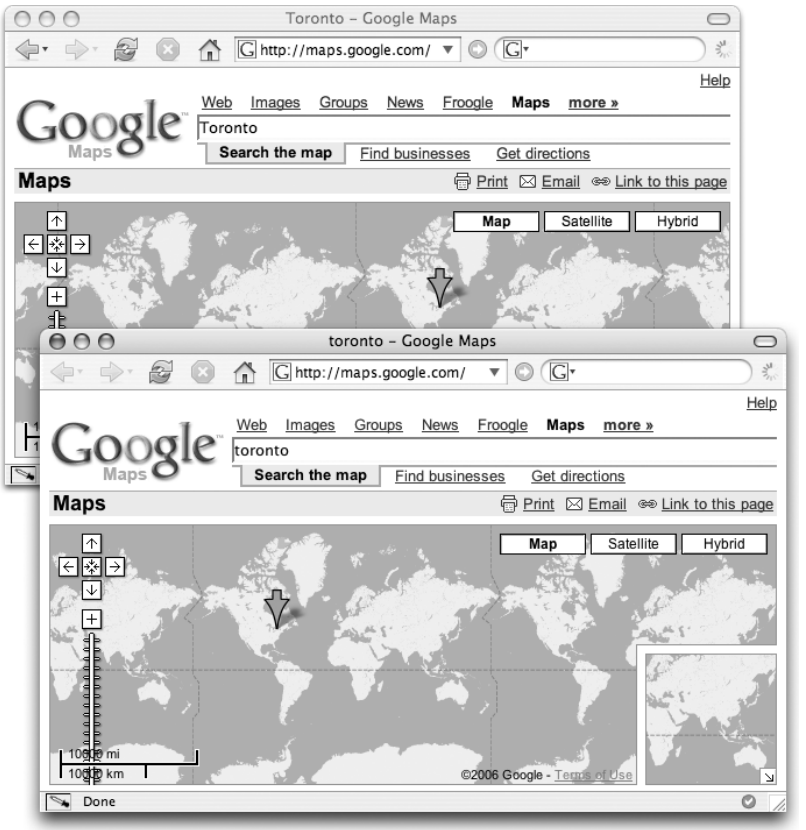


Figure 10-9. *Multiple copies of the earth's land masses, but only one marker*

Summary

This chapter provided an analysis of map regions, including area and perimeter, and described how to plot and handle `GPolyline` objects.

We hope this material was useful to you. Perhaps it has even given you some ideas of what's possible in and beyond the API. When that mashup opportunity comes along—the one with the voting regions, or the school districts, or the shorelines affected by an oil spill—you'll be armed with tools to get a clear and helpful visual look at the matter.

In the next, and last, chapter, we'll be discussing how a geocoder is built from scratch from two different sources of information. This will help you understand the limitations of precision in geocoding, as well as teach you the fundamentals of how to use a *very* rich data source: the U.S. Census Bureau's TIGER/Line files.

CHAPTER 11

Advanced Geocoding Topics

In this chapter, you'll learn how to create your own geocoding service. However, there are geocoding services already available for all sorts of data, and we covered many of them in Chapter 4. Therefore, this chapter is intended for professionals and serious hobbyists who are building web applications where using third-party geocoding tools is not feasible due to cost, rate limiting, or terms of service. In these cases, developers often have no choice but to resort to getting dirty and becoming familiar with the original sources of data to do it themselves. If this describes you, grab a pencil and paper, put on your thinking cap, and read on. We're about to get messy.

In this chapter, you'll learn how to do the following:

- Find sources of information used to create geocoding services.
- Construct a postal code–based geocoding service for the United Kingdom. This example can be easily applied to the United States, Canada, and other countries, assuming you have access to the raw data.
- Build a more complicated and sophisticated geocoding web service for the United States, using the data from the U.S. Census Bureau.

Where Does the Data Come From?

So where do services such as Google and Yahoo get *their* data? How do they convert it into something that we can use to plot things on our maps? For geographic information systems (GIS) enthusiasts, this is a question with a really interesting answer and the topic of most of this chapter.

Almost exclusively, this data comes from various government departments and agencies. Most often, some central authority (such as the U.S. Census Bureau) mandates that each municipality or county must provide data that is accurate to some specified degree. For many counties, meeting this requirement is not a matter of obtaining the data, but merely repurposing it. They already keep geographic information about land surveys, plot locations, and ownerships for taxation and legal purposes; converting it into maps and other GIS-related information is only a matter of time, resources, and incentive.

Note This discussion applies primarily to Canada and the United States. For other areas of the world, similar kinds of information are slowly becoming available, and we are seeing more and more elaborate and complete geocoders each day, including the one introduced by Google. We hope that as the Google developers expand their road network data, they keep their geocoder in sync; however, this chapter should help you understand how to fill in the gaps that they miss.

Sample Data from Government Sources

Figure 11-1 shows an example (a single block) of the kind of data that a typical urban planning department might have created. It shows each plot of land, the intersections, and the points where the road bends. This is a simplified example that we'll build up and use throughout the chapter, so you'll want to refer back to this page.

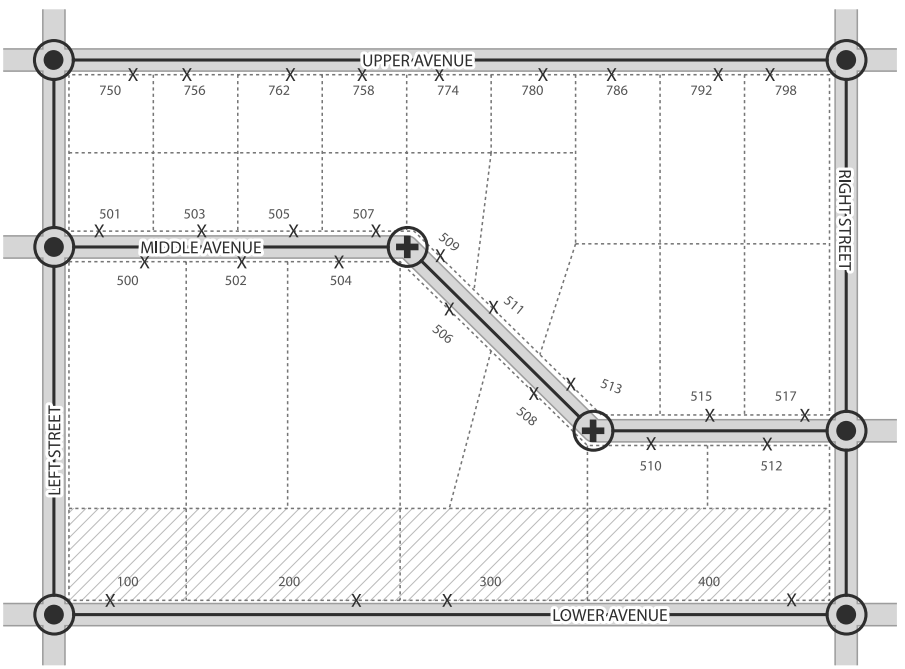


Figure 11-1. *Simplified example of a block of land in an urban planning department database*

You can see from the illustration that each plot of land is individually identified and that the roads are broken up into segments defined by intersections. Table 11-1 shows a representation of the data for each plot of land. Table 11-2 describes the sections of road. Table 11-3 holds the latitude and longitude data for each interior bend in the road, if there are any, and is associated by ID number.

Table 11-1. *A Portion of the Data for the Precise Location of Each Section of Land in Figure 11-1*

Street Name	Street No.	ZIP Code	Latitude	Longitude
Upper Ave.	750	90210	43.1000	-80.1001
Upper Ave.	756	90210	43.1000	-80.1003
Upper Ave.	762	90210	43.1000	-80.1005
Upper Ave.	758	90210	43.1000	-80.1007
Upper Ave.	774	90210	43.1000	-80.1009
Upper Ave.	780	90210	43.1000	-80.1011
Upper Ave.	786	90210	43.1000	-80.1013
Upper Ave.	792	90210	43.1000	-80.1015
Upper Ave.	798	90210	43.1000	-80.1017
Middle Ave.	501	90211	43.1005	-80.1001
Middle Ave.	503	90211	43.1005	-80.1003
Middle Ave.	505	90211	43.1005	-80.1005
Middle Ave.	507	90211	43.1005	-80.1007

Table 11-2. *Road Complete Chain End Points*

ID	Street Name	Start Latitude	Start Longitude	End Latitude	End Longitude
1000	Upper Ave.	43.1000	-80.1000	43.1000	-80.1020
1001	Lower Ave.	43.1010	-80.1000	43.1010	-80.1020
1002	Middle Ave.	43.1005	-80.1000	43.1007	-80.1020
1003	Left Street	43.1000	-80.1000	43.1005	-80.1000
1004	Left Street	43.1005	-80.1000	43.1010	-80.1000
1005	Right Street	43.1000	-80.1020	43.1007	-80.1020
1006	Right Street	43.1007	-80.1020	43.1010	-80.1020

Table 11-3. *Road Complete Chain Interior Points*

End Point ID	Sequence Num.	Latitude	Longitude
1002	1	43.1005	-80.1007
1002	2	43.1007	-80.1013

In this example, Table 11-3 shows the two interior points for Middle Avenue. You can think of the "End Point ID" column as a foreign key to the "ID" column in Table 11-2.

Of course, the Table 11-1 data is ideal for geocoding an address. It's simply a matter of looking up the street name and number, and then reading off the latitude and longitude. This data is also known as *street truth* or *ground truth* data, since it is roughly the same data you would get if you visited each address personally and used a handheld GPS device to read off the

coordinates. Unfortunately, this level of data is rarely available for free, and when it is, it's only on a county-by-county basis.

The data in Tables 11-2 and 11-3, when combined, gives a very accurate picture of the streets' locations and how they intersect, and yet there is no information about the addresses of the buildings *along* those streets.

In reality, a combined set of data is what you're likely to get from a census bureau. Table 11-4 gives an amalgamated view of the records from Tables 11-1 and 11-2. This is roughly the same format that the U.S. Census Bureau provides in its TIGER/Line data set, which we'll introduce in the next section.

Table 11-4. *Road Network Chain End Points*

ID No.	Street Name	Start Latitude	Start Longitude	End Latitude	End Longitude	Left Addr. Start	Left Addr. End	Right Addr. Start	Right Addr. End
1000	Upper Ave.	43.1000	80.1000	43.1000	80.1020			750	798
1001	Lower Ave.	43.1010	80.1000	43.1010	80.1020	100	400		
1002	Middle Ave.	43.1005	80.1000	43.1007	80.1020	501	517	500	512
1003	Left Street	43.1000	80.1000	43.1005	80.1000				
1004	Left Street	43.1005	80.1000	43.1010	80.1000				
1005	Right Street	43.1000	80.1020	43.1007	80.1020				
1006	Right Street	43.1007	80.1020	43.1010	80.1020				

You might be curious what left and right address start and end mean. Presume that you're standing on the intersection defined by a "start" latitude/longitude pair facing the "end" latitude/longitude pair. From this reference point, you can tell that the addresses on one side are "left" and the other side are "right." This is how most GIS data sets pertaining to roads define left vs. right. They cannot be correlated to east or west and merely reflect the order in which the points were surveyed by the municipalities.

By using the start and end addresses on a street segment in conjunction with the start and end latitude and longitude, you can guess the location of addresses in between. This is called *interpolation* and allows the providers of a data source to condense the data without a significant loss in resolution. The biggest problem arises when the size of the land divisions is not proportional to the numbering scheme. In our example (Figure 11-1), this occurs on the south side of Middle Avenue and also on Lower Avenue. This can affect the accuracy of your service, because you are forced to assume that all address numbers between your two end points exist and that they are equally spaced. We'll discuss this further in the "Building a Geocoding Service" section later in this chapter.

In cases where you cannot obtain any data based on streets, you can try to use the information used to deliver the mail. The postal services of most countries maintain a list of postal codes (*ZIP codes* in the United States) that are assigned to a rough geographic area. Often, a list of these codes with the corresponding latitude and longitude of the center of the area is available for free or for a minimal charge. Figure 11-2 shows our sample map with postal codes represented by single letters, *A* through *E*, and shaded areas. The small black *x* represents the latitude and longitude point recorded for each postal code.

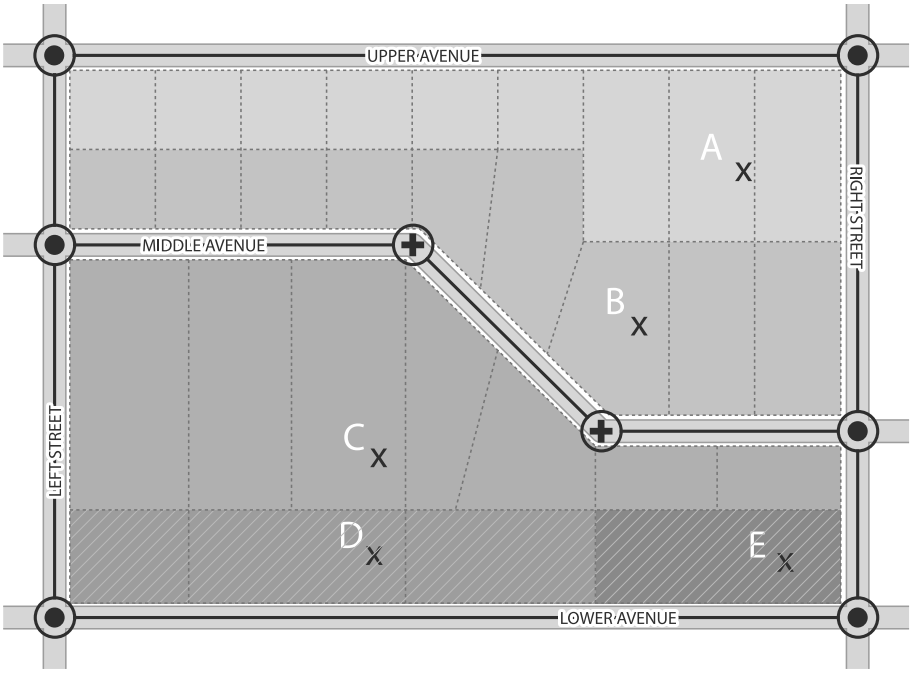


Figure 11-2. *Sample map showing single letters representing postal codes*

In urban areas, where a small segment of a single street may be represented by a unique postal code, this might be enough to geocode your data with sufficient accuracy for your project. However, problems arise when you leave the urban areas and start dealing with the rural and country spaces where mail may not be delivered directly to the houses. In these places, a single unique postal code could represent a post office (for P.O. boxes) or a geographical area as large as 30 square miles or more.

■ **Note** In addition to the freely available data from the governments, in some cases private companies may take multiple sources of data and condense them into a commercial product. Often these commercial products also cross-reference sources of data in an attempt to filter out errors in the original sources. An example of one such product is the Geocoder.ca service discussed in Chapter 4.

Sources of Raw GIS Data

In the United States, a primary source of GIS data is the TIGER/Line (Topologically Integrated Geographic Encoding and Referencing system) information, which is currently being revised by the U.S. Census Bureau. This data set is huge and very well-documented. As of this writing, the most current version of this data is the 2005 second edition data set (released in June 2006), which is available from the official web site at `http://www.census.gov/geo/www/tiger/index.html`. The online geocoding service Geocoder.us relies on the TIGER/Line data, and we suspect that

this data is also used (at least in part) by all of the other U.S.-centric geocoding services, such as Google and Yahoo.

For Canada, the Road Network File (RNF) provided by the Canadian census department's Statistics Canada is excellent. You can find it at `http://geodepot.statcan.ca/Diss/Data/Data_e.cfm`. The current version as of this writing is the 2005 RNF. This data is available in a number of formats for various purposes. For the sake of programmatically creating a geocoder, you'll probably want the Geographic Markup Language (GML) version, since it can be processed with standard XML tools. The people who built Geocoder.ca used the RNF, combined with the Canadian Postal Code Conversion File (`http://www.statcan.ca/bsolc/english/bsolc?catno=92F0153X`) and some other commercial sources of data to create a unified data set. They attempted to remove any errors in an individual data set by cross-referencing all the sources of data.

For the United Kingdom, you can find a freely redistributable mapping between UK postal codes and crude latitude and longitude floating around the Internet. We've mirrored the information on our site at `http://googlemapsbook.com/chapter11/uk-postcodes.csv`. This information was reportedly created with the help of many volunteers and was considered reasonably accurate as of 2004. If you want to use the information for more than experimenting, you might consider obtaining the official data from the UK postal service.

For the rest of the world, you can use data provided by the National Geospatial Intelligence Agency (NGA). This data should be useful in geocoding the approximate center of most populated areas on the planet. The structure of the data provides for alternative names and permanent identifiers. For more information about this data set, see the section about geographic names data in Appendix A of this book.

The parsing and lookup methods used in the "Using the TIGER/Line Data" section later in this chapter also generally apply to the Canadian RNF and the geographical names data sets, so we won't cover them with examples directly.

■ **Note** In Japan, at least in some places, the addressing scheme is determined by the order in which the buildings were constructed, rather than their relative positions on the street. For example, 1 Honda Street is not necessarily next to, or even across the street from, 2 Honda Street. Colleagues who have visited Japan report that navigation using handheld GPS and landmarks is much more common than using street number addresses, and that many businesses don't even list their street numbers on the sides of the buildings or in any marketing material.

Geocoding Based on Postal Codes

Let's start to put some of this theory into practice. We'll begin with a geocoding solution based on the freely available UK postal code data mentioned in the previous section.

First, you'll need to get the raw CSV data from `http://googlemapsbook.com/chapter11/uk-postcodes.csv` and place it in `lib/chap_eleven/uk-postcodes.csv` on your workstation. This file should be about 90KB. Listing 11-1 shows a small sample of the contents of this file.

Listing 11-1. *Sample of the UK Postal Code Database for This Example*

```
postcode,x,y,latitude,longitude
AB10,392900,804900,57.135,-2.117
AB11,394500,805300,57.138,-2.092
AB12,393300,801100,57.101,-2.111
AB13,385600,801900,57.108,-2.237
AB14,383600,801100,57.101,-2.27
AB15,390000,805300,57.138,-2.164
AB16,390600,807800,57.161,-2.156
AB21,387900,813200,57.21,-2.2
AB22,392800,810700,57.187,-2.119
AB23,394700,813500,57.212,-2.088
AB25,393200,806900,57.153,-2.112
AB30,370900,772900,56.847,-2.477
AB31,368100,798300,57.074,-2.527
AB32,380800,807200,57.156,-2.317
```

The `postcode` field in this case simply denotes the forward sorting area, or *outcode*. The outcodes are used to get mail to the correct postal office for delivery by mail carriers. A full postal code would have a second component that identifies the street and address range of the destination and would look something like AB37 A5G. Unfortunately, we were unable to find a free list of full postal codes. The `x` and `y` fields represent meters relative to a predefined point inside the borders of the United Kingdom. The equation for converting these to latitude and longitude is long, involved, and not widely applicable, so we won't cover it here. Last are the fields we're interested in: `latitude` and `longitude`. They contain the latitude and longitude in decimal notation, ready and waiting for mapping on your Google map mashup.

■ **Note** For most countries you can find sources of data that have full postal codes mapped to latitude and longitude. However, this data is often very pricey. If you're interested in obtaining data for a specific country, be sure to check out the GeoNames data and try searching online; but you may need to directly contact the postal service of the country you're interested in and pay its licensing fees.

Next, you need to create the model and migration for the UK postal codes data. From the command line, enter **ruby script/generate model UkPostcode** to create the model and migration files. The resulting migration file should be `db/migrations/007_create_uk_postcodes.rb`. Copy the code from Listing 11-2 into this file.

■ **Note** If you are out of step with the examples in this book, your migration may be different (i.e., `00X_create_uk_postcodes.rb`). That's fine as long you can run the migration.

Listing 11-2. *New Migration db/migrations/007_create_uk_postcodes.rb*

```
class CreateUkPostcodes < ActiveRecord::Migration
  def self.up
    create_table :uk_postcodes do |t|
      t.column :outcode, :string, :limit=>4, :null=>false
      t.column :latitude, :float, :default=>0, :null=>false
      t.column :longitude, :float, :default=>0, :null=>false
    end
  end

  def self.down
    drop_table :uk_postcodes
  end
end
```

Once you've created the migration, run it with the command **rake db:migrate**. Now you need to import the CSV data into your database. For this you can use the Rake task in Listing 11-3; just copy this code into a new file, `/lib/tasks/importing_tasks.rake`.

Listing 11-3. *Rake Importing Tasks, /lib/tasks/importing_tasks.rake*

```
task :import_uk_postcodes => :environment do
  begin
    csv=CSV.open("#{RAILS_ROOT}/lib/chap_eleven/uk-postcodes.csv", "r")
    csv.shift  # skip header row
    csv.each_with_index do |row,i|
      outcode,x,y,latitude,longitude = row
      UkPostcode.create(:outcode => outcode, :latitude => latitude,➥
 :longitude => longitude)
      puts "#{i} postcodes imported" if i % 50 == 0
    end
  ensure
    csv.close unless csv.nil?
  end
end
```

This code uses the standard library CSV functionality to open the CSV file at the location where we saved it. Then it skips the header row and iterates through the remaining lines, using the `UkPostcode` model to store the data in the database.

Execute the task from the command line by typing **rake import_uk_postcodes**. You can verify the results of the import with a console session (bold lines are your input):

```
>ruby script/console
Loading development environment.
>> UkPostcode.count
=> 2821
```

For a public-facing geocoder, we'll need some code to expose a simple web service, allowing users to query our database from their application. A simple action takes care of this. First,

create a new controller (**ruby script/generate controller chap_eleven** from the command line). Then add the code in Listing 11-4 to the `/app/controllers/chap_eleven_controller.rb` file.

Listing 11-4. *Gecoding REST Service for UK Outcodes*

```
def geocode_uk
  code = params[:code].upcase.gsub(/[^A-Z0-9]/,'')[0..4]
  result=UkPostcode.find_by_outcode(code)
  if result
    render :xml=>result.to_xml(:except => :id,:root=>'result')
  else
    render :xml=>{:error=>'No Matches'}.to_xml(:root=>'result')
  end
end
```

The code is very straightforward. First, we clean up the `code` parameter by converting the string to uppercase (`upcase`), stripping out nonalphanumeric characters (`gsub`), and finally reducing the length to four characters (using the array index notation). Next, we simply query the model, looking for an exact match, and output the answer if we find one. Note that the `gsub` method also strips whitespace by removing all nonalphanumeric characters.

To test the web service, direct your browser to `http://localhost:3000/chap_eleven/get_uk_geocode?code=AB10`. The outcode looked up in this call is specified in the `code` URL parameter, with a value (in this case) of `AB10`. Depending on your browser settings, you may have to view the document source to see the XML. You should see the following:

```
<result>
  <latitude>57.135</latitude>
  <longitude>-2.117</longitude>
  <outcode>AB10</outcode>
</result>
```

FUZZY PATTERN MATCHING

If you would prefer to allow people to match on partial strings, you'll need to be a bit more creative. Since the finder is a little more complicated, it makes sense to push the logic into the model. You can add the following code into `app/models/uk_postcode.rb` to provide fuzzy matching functionality:

```
def self.fuzzy_find_by_outcode(code)
  code = code.upcase.gsub(/[^A-Z0-9]/,'')[0..4]
  result=nil
  loop do
    break if code.length == 0 || result
    result=self.find :first,
      :conditions=>['outcode like ?', "#{code}%"]
    code.chop! unless result
  end
```

```
    return result
  end
```

Now you can simply replace the `result=UkPostcode.find_by_outcode(code)` line in your controller with `result=UkPostcode.fuzzy_find_by_outcode(code)`. The fuzzy find method just queries the database with a wildcard at the end of the requested code. If it doesn't find anything, it keeps stripping off a character and requerying until it finds something or runs out of characters. Only if the length of the request code is zero does it give up and return an error.

In case you are wondering, it's the `self` keyword in the `self.fuzzy_find_by_outcode` syntax that makes the method a class method rather than an instance method. This is what allows us to call the method using the class itself (`UkPostcode` in this case) rather than an instance of the class.

Using the TIGER/Line Data

So what about street address geocoding? In this section, we discuss the U.S. Census Bureau TIGER/Line data in detail. You can approach this data for use in a homegrown, self-hosted geocoder in two ways:

- Use the Perl programming language and take advantage of the `Geo::Coder::US` module that powers `http://www.geocoder.us`. It's free, fairly easy to use—if you already know Perl (or someone who does)—and open source, so it should continue to live for as long as someone finds it useful.
- Learn the structure of the data and how to parse it using Ruby. This is indeed much more involved. However, it has the benefit of opening up the entire data set to you. There is much more information in the TIGER/Line data set than road and street numbers (see Appendix A). Knowing how to use this data will open a wide variety of possible mapping applications to you, and therefore we feel it is worthwhile to show you how it works.

We'll begin by giving you a bit of a primer on the structure of the data files then get into parsing them with Ruby and finish off by building a basic geocoder.

As we mentioned earlier in the chapter, the TIGER/Line data is currently being revised and updated. The goal of this project is to consolidate information from many of the various sources into a widely applicable file for private and public endeavors. Among other things, the U.S. Census Bureau is integrating the Master Address File originally used to complete the 2000 census, which should increase the accuracy of the address range data. The update project is scheduled to be complete in 2008, so anything you build based on these files will likely need to be kept up to date manually for a few years.

Understanding and Defining the Data

Before you can begin, you'll need to select a county. For this example, we selected San Francisco County in California. Looking up the FIPS (Federal Information Processing Standards) codes for the county and state in the documentation (`http://www.census.gov/geo/www/tiger/tiger2005se/TGR05SE.pdf`), we find on page A-3 that they are 075 and 06, respectively. You can use any county and state you prefer; simply change the parameters in the examples that follow.

Note A FIPS code has been assigned to each state and county, allowing you to identify with numbers the various different entities quickly. There has been much discussion lately about replacing FIPS with something that gives a more permanent number (FIPS codes can change), and also at the same time allows you to infer proximity based on the code. We encourage you to Google **FIPS 55 changes** for the latest information.

Next, you need to download the corresponding TIGER/Line data file so that you can play with it and convert it into a set of database tables for geocoding. In our case, the file is located at `http://www2.census.gov/geo/tiger/tiger2005se/CA/tgr06075.zip`. Download and unzip it to see the raw data files.

Note The second edition of the 2005 TIGER/Line data files was released on June 27, 2006. Data sets are released approximately every six months. We suggest grabbing the most recent set of data, with the understanding that minor things in these examples may change.

Inside the ZIP file, you'll find a set of text files, all with an `.rt*` extension. We've spent many days reading through the documentation to determine which of these files are really necessary for our geocoder. You're welcome to read the documentation for yourself, but to save you time and a whopping headache, we'll be working with the RT1, RT2, and RTC files. In addition, we'll briefly describe the RT4, RT5, and RT6 files (and what you could get out of them), but we won't be working with them directly in this example.

The RT1 file contains the end points of each complete chain. A *complete chain* defines a segment of something linear such as a road, a highway, a stream, or train tracks. A *segment* exists between intersections with other lines (usually of the same type). A *network chain* is composed of a series of complete chains (connected in order) to define the entire length of a single line.

Note In our case, we'll be ignoring all of the complete chains that do not represent streets with addresses. Therefore, we will refer to them as *road segments*.

The RT1 file ties everything else together by defining a field called `TLID` (for TIGER/Line ID) and stores the start and end points of the road segments along with the primary address ranges, ZIP codes, and street names. The RT2 file can be linked with the RT1 file via the `TLID` field and gives the internal line points that define bends in the road segment.

The RTC file contains the names of the populated places (towns, cities, etc.) referenced in the PLACE fields in RT1. If you are importing a densely populated county, you may only get one record from the RTC file.

The RT4 file provides a link between the TLID values in the RT1 file and another ID number in the RT5 file: the FEAT (for feature) identifier. FEAT identifiers are used to link multiple names to a single road segment record. This is handy because many streets that are lined with residential housing also double as highways and major routes. If this is the case, a single road might be referred to by multiple names (highway number, city-defined name, etc.). If someone is looking up an address and uses the less common name, you should probably still give the user an accurate answer.

The RT6 file provides additional address ranges (if available) for records in RT1.

■ **Caution** Both RT4 and RT6 have a field called RTSQ. This represents the order in which the elements should be applied, but *cannot be used to link RT4 and RT6 together*. This means that a corresponding value of RTSQ does not imply that certain address ranges link with specific internal road segments for a higher level of positional accuracy. As tantalizing as this would be, we've confirmed this lack of correlation directly with the staff at the U.S. Census Bureau.

We won't get into too much detail about the contents of each record type until we start talking about the importing routines themselves. What we will talk about now is the relational structure used to hold the data. Unlike with the previous postal code example, it doesn't make sense to store the street geocoder in a single, spreadsheetlike table. Instead, we'll break it up into four distinct SQL tables:

- *The* `places` *table*: Stores the FIPS codes for the state, county, and place (city, town, etc.), as well as the actual name of the place. We're going to override Rails' default behavior for the primary key on this table and make the `id` column a string. The `id` string is a concatenation of the state, county, and place FIPS codes. It is nine or ten characters long. This data is acquired from various FIPS files that we'll talk about shortly and the TIGER/Line RC file. We are using the specially formulated string `id` column so that other imported records can refer to these rows by formulating the same ID value. If we let Rails assign numeric IDs, we'd have to do an expensive lookup during subsequent import operations. Note that since we are going to override the default behavior for the primary key, our code will be responsible for inserting only unique primary key values into the `id` column.

- *The* `street_names` *table*: Stores the names, directions, prefixes, and suffixes of the streets and attaches them to `place.id` values. It is primarily derived from the RT1 and RT5 records. It also stores the official TLID from the TIGER/Line data set, so that you can easily update your data in the future.

- *The* `complete_chains` *table*: Stores the latitude/longitude pairs that define the path of each road segment. This table also stores a sequence number that can be used to sort the chain into the order that it would be plotted on a map. This data comes from the RT1 and RT2 records.
- *The* `address_ranges` *table*: Holds various address ranges attached to each road segment, as the name implies. Most of this data will come from the RT1 records, though any applicable RT6 records can also be placed here.

The migration code for creating these four tables is shown in Listing 11-5. Generate the migration (**ruby script/generate migration create_tigerline_geocoder**) and place the following code into the resulting new file, `db/migrate/008_create_tigerline_geocoder.rb`.

Listing 11-5. *Migration to Create the Tables for the TIGER/Line Geocoder*

```
class CreateTigerlineGeocoder < ActiveRecord::Migration
  def self.up
    create_table :places, :id=>false do |t|
      t.column :id, :string, :limit=>14, :null=>false, :default=>''
      t.column :state_fips, :string, :limit=>2, :null=>false, :default=>''
      t.column :county_fips, :string, :limit=>3, :null=>false, :default=>''
      t.column :place_fips, :string, :limit=>5, :null=>false, :default=>''
      t.column :state_name, :string, :limit=>60, :null=>false, :default=>''
      t.column :county_name, :string, :limit=>30, :null=>false, :default=>''
      t.column :place_name, :string, :limit=>30, :null=>false, :default=>''
    end
    execute 'ALTER TABLE places ADD PRIMARY KEY (id)'

    create_table :street_names do |t|
      t.column :tlid, :integer, :null=>false, :default=>0
      t.column :place_id, :string, :limit=>15, :null=>false, :default=>''
      t.column :cfcc, :string, :limit=>3, :null=>false, :default=>''
      t.column :dir_prefix, :string, :limit=>2, :null=>false, :default=>''
      t.column :name, :string, :limit=>30, :null=>false, :default=>''
      t.column :street_type, :string, :limit=>4, :null=>false, :default=>''
      t.column :dir_suffix, :string, :limit=>2, :null=>false, :default=>''
      t.column :cfcc, :string, :limit=>3, :null=>false, :default=>''
    end
    add_index :street_names, [:tlid,:name]

    create_table :address_ranges do |t|
      t.column :tlid, :integer, :null=>false, :default=>0
      t.column :range_id, :integer, :null=>false, :default=>0
      t.column :first, :string, :limit=>11, :null=>false, :default=>0
      t.column :last, :string, :limit=>11, :null=>false, :default=>0
    end
```

```
    add_index :address_ranges, :tlid
    add_index :address_ranges, [:first,:last]

    create_table :complete_chains do |t|
      t.column :tlid, :integer, :null=>false, :default=>0
      t.column :seq, :integer, :null=>false, :default=>0
      t.column :latitude, :float, :null=>false, :default=>0
      t.column :longitude, :float, :null=>false, :default=>0
    end
    execute("ALTER TABLE complete_chains MODIFY latitude numeric(15,10);")
    execute("ALTER TABLE complete_chains MODIFY longitude numeric(15,10);")
    add_index :complete_chains, :tlid
  end

  def self.down
    drop_table :places
    drop_table :street_names
    drop_table :address_ranges
    drop_table :complete_chains
  end
end
```

Note that as indicated earlier, we made the primary key column of the `places` table a string rather than the default integer. To do this (and still name the column `id`), we've specified `:id=>false` in the `create_table` statement for the `places` table, and then specified the primary key later with `execute 'ALTER TABLE places ADD PRIMARY KEY (id)'`.

To create these tables, run **rake db:migrate** from the command line.

Parsing and Importing the Data

Next, we need to determine how we are going to parse the data. The U.S. Census Bureau has complicated our parsing a bit in order to save the nation's bandwidth. There is no need to include billions of commas or tabs in the data when you can simply define a parsing structure and concatenate the data into one long string. Chapter 6 of the official TIGER/Line documentation defines this structure for each type of record in the data set. Table 11-5 shows the simplified version we've created to aid in our automated parsing of the raw data.

■ **Caution** Our dictionaries are not complete representations of each record type. We've omitted the record fields that we are not interested in to speed up the parsing when importing. Basically, we don't really care about anything more than the field name, starting character, and field width. We've left the human-readable names in for *your* convenience. We've also omitted many field definitions for information we're not interested in (such as census tracts or school districts).

Table 11-5. *Data Dictionary for RT1*

Field Name	Start Char	Length	Description
TLID	6	10	TIGER/Line ID, Permanent 1-Cell Number
FEDIRP	18	2	Feature Direction, Prefix
FENAME	20	30	Feature Name
FETYPE	50	4	Feature Type
FEDIRS	54	2	Feature Direction, Suffix
CFCC	56	3	Census Feature Class Code
FRADDL	59	11	Start Address, Left
TOADDL	70	11	End Address, Left
FRADDR	81	11	Start Address, Right
TOADDR	92	11	End Address, Right
PLACEL	161	5	FIPS 55 Code (Place/CDP), 2000 Left
PLACER	166	5	FIPS 55 Code (Place/CDP), 2000 Right
FRLONG	191	10	Start Longitude
FRLAT	201	9	Start Latitude
TOLONG	210	10	End Longitude
TOLAT	220	9	End Latitude

Importing and manipulating the data will require considerable amounts of time and processing resources. A single county (San Francisco) processed on our laptop-based development environment took about 25 minutes to import. If you're just experimenting with these techniques, we suggest that you pick a single county (preferably your own, so the results are familiar) instead of working with a whole state or more.

With all of this in mind, let's get started. First, download the dictionaries from `http://googlemapsbook.com/chapter11/tiger_dicts.zip`. Unzip the files and place them in `lib/chap_eleven/tigerline/dictionaries/`. Take a look at the files; you'll see they are tab-delimited text. To parse these dictionaries and use them on the raw TIGER/Line data, we'll create the helper `LineParser` class, as shown in Listing 11-6. You can place this class wherever you like, as long as it's in the application path. We placed ours in `lib/tigerline_geocoder.rb`.

Listing 11-6. *LineParser Helper Class for Importing TIGER/Line Data*

```
class LineParser
  attr_accessor :type

  def initialize(root_path,type)
    @type=type
    @fields = Hash.new
    IO.foreach(File.join(root_path,'dictionaries',"#{type}.dict")) do |line|
      dict_fields=line.split(/\t/)
      @fields[dict_fields[0].intern]={
              :start=>dict_fields[1].to_i,
              :length=>dict_fields[2].to_i,
```

```
                :name=>dict_fields[3].chomp}
    end
  end

  def parse(line)
    res=Hash.new
    @fields.each_key {|field|
        res[field] = (line[@fields[field][:start]-1,
                      @fields[field][:length]]).strip}
    return res
  end
end
```

This class takes care of parsing the raw TIGER/Line files into the fields that we care about. When you create an instance of the class, the instance opens the appropriate dictionary file (based on the value of the `type` argument) in the `dictionaries` directory. Then, it sets up the `@fields` member so it can easily parse lines in the `parse` method.

The `parse` method will be called for each line in the TIGER/Line data file. It utilizes the `@fields` data structure to pick the values out of the line and place them in a hash table with the appropriate keys.

Now that we know where we are going (our table structure created via migrations) and how to get there (our parsing helper class), let's import some data. To facilitate explanation of the process, we'll break the importer out into a separate listing for each record type. In reality, *all of these listings form a single Rake task*, but for the purposes of describing each stage of the process, it makes sense to break it into segments. Listing 11-7 covers the importing of the RT1 data file.

Listing 11-7. *Rake Task for Importing RT1 Records (Add to lib/tasks/importing_tasks.rake)*

```
# include the file you put the parsing helper class in
require 'tigerline_geocoder'

# begin the rake task
task :import_tigerline => :environment do
  #set state and county
  state='06'
  county='075'
  type='rt1'

  file_prefix="TGR#{state}#{county}"
  root_dir="#{RAILS_ROOT}/lib/chap_eleven/tigerline"

  i=0
  tlids=Hash.new
  parser=LineParser.new(root_dir,type)

# loop through each line in the file
IO.foreach("#{RAILS_ROOT}/lib/chap_eleven/tigerline/➥
#{file_prefix}.#{type.upcase}") do |line|
```

```
    # parse the line
    l=parser.parse(line)

    # Reference 1: we're not interested in the line of data in the following cases
    # A. its CFCC type is not group A (i.e., it's not a road)
    next if l[:CFCC][0,1] != 'A'
    # B. there are no addresses for either side of the street
    next if l[:FRADDL] == '' and l[:FRADDR] == ''
    # C. if no city is associated with the road, it'll be hard to identify
    next if l[:PLACEL] == '' and l[:PLACER] == ''

    # Reference 2: parse the from and to lat/lngs
    from_latitude = l[:FRLAT][0,l[:FRLAT].size-6] + '.' +➥
l[:FRLAT][l[:FRLAT].size-6,6]
    from_longitude = l[:FRLONG][0,l[:FRLONG].size-6] + '.' +➥
l[:FRLONG][l[:FRLONG].size-6,6]
    to_latitude = l[:TOLAT][0,l[:TOLAT].size-6] + '.' +➥
l[:TOLAT][l[:TOLAT].size-6,6]
    to_longitude = l[:TOLONG][0,l[:TOLONG].size-6] + '.' +➥
l[:TOLONG][l[:TOLONG].size-6,6]

    # decide if this is a boundary of a place
    places = Array.new
    if l[:PLACEL] != l[:PLACER]
      places.push(l[:PLACEL]) if l[:PLACEL] !=''
      places.push(l[:PLACER]) if l[:PLACER] !=''
    else
      places.push(l[:PLACEL])
    end

    # we're only using a transaction because it's faster
    StreetName.transaction do
      # loop and process the places array
      # ( there will be up to two elments in the array)
      places.each do |place_fips|
        # Reference 3
        # A. create a street_name
        StreetName.create(:tlid=>l[:TLID],
            :place_id=>"#{state}#{county}#{place_fips}",
            :cfcc=>l[:CFCC], :dir_prefix=>l[:FEDIRP], :name=>l[:FENAME],
            :street_type=>l[:FETYPE],:dir_suffix=>l[:FEDIRS])

        # B. create one or two address_ranges
        AddressRange.create(:tlid=>l[:TLID], :range_id=>-1,:first=>l[:FRADDR],➥
:last=>l[:TOADDR]) if l[:FRADDR] !=''
        AddressRange.create(:tlid=>l[:TLID], :range_id=>-2,:first=>l[:FRADDL],➥
:last=>l[:TOADDL]) if l[:FRADDL] !=''
```

```
        # C. create two complete_chains
        CompleteChain.create(:tlid=>l[:TLID], :seq=>0, :latitude=>from_latitude,➥
 :longitude=>from_longitude)
        CompleteChain.create(:tlid=>l[:TLID], :seq=>5000, :latitude=>to_latitude,➥
 :longitude=>to_longitude)

      end # end of loop through places
    end # end of the transaction

    tlids[l[:TLID]]=true
    i=i+1
    puts "imported #{i} #{type} records" if i % 200 == 0
  end

end
```

Three key things are happening here:

- We're selectively ignoring lines that are irrelevant to geocoding. See the comment `Reference 1` in the code to understand where this logic resides. Structures such as bridges, rivers, and train tracks, plus places such as parks, bodies of water, and landmarks, are all listed in the RT1 file along with the roads. We can identify the type of thing represented by each row by looking at the `CFCC` field and using only items that start with an *A*. In addition to using only roads, we don't care about roads that have no address ranges (how would you identify a single point on the line?) or that are not part of a populated area such as a city or a town.
- The latitude and longitude need to have their decimal symbols reinserted (they were stripped to save bandwidth). All coordinates are listed to six decimal places. See the comment `Reference 2` in the code to understand where this logic resides.
- We're splitting up the data as we described for our schema. See the comment `Reference 3` in the code to understand where this logic resides. For simplicity, we remove the left and right side awareness for the address ranges and list the same segment twice, if it is a boundary between two populated places. We also place the starting latitude/longitude pair into the `complete_chains` table with a sequence number of 1, and the end pair with a sequence number of 5,000. We do this because the documentation states that no chain will have more than 4,999 latitude/longitude pairs, and we haven't yet parsed the RT2 records to determine how many other points there may be.

Caution The TIGER/Line documentation is very careful to state that just because the latitude and longitude data is listed to six decimal places does not mean that it is *accurate* to six decimal places. In some cases it may be, but in others it may also be third- or fourth-generation interpolated data.

This brings us nicely to parsing the RT2 records. Listing 11-8 shows the code that follows the parsing of RT1 inline in our Rake task.

Listing 11-8. *Parsing for RT2 Records*

```
  type='rt2'
  i=0
  parser=LineParser.new(root_dir,type)

  #  loop through each line in the file
  IO.foreach("#{RAILS_ROOT}/lib/chap_eleven/tigerline/➥
                    #{file_prefix}.#{type.upcase}") do |line|

    # parse the line
    l=parser.parse(line)

    # skip this record if we didn't encounter its TLID
    # while processing rt1
    next if !tlids[l[:TLID]]
    # we're only doing the transaction because it's faster
    CompleteChain.transaction do
      # loop through each of the 10 points, looking for one that's 0,0
      (1..10).each do |j|
        latitude=l["LAT#{j}".intern]
        longitude=l["LONG#{j}".intern]
        # Reference 1
        break if longitude.to_i == 0 && latitude.to_i == 0

        #parse some data
        latitude = latitude[0,latitude.size-6] + '.' + latitude[latitude.size-6,6]
        longitude = longitude[0,longitude.size-6] + '.' +➥
longitude[longitude.size-6,6]
        seq = l[:RTSQ].rjust(2,'0')

        #insert the record
        CompleteChain.create(:tlid=>l[:TLID], :seq=>seq, :latitude=>latitude,➥
:longitude=>longitude)
      end #end of 10-loop
    end # end of transaction

    i=i+1
    puts "imported #{i} #{type} records" if i % 200 == 0
  end # foreach
```

Basically, we're just adding records to the `complete_chains` table for any `TLID` that we deem important while we parse the RT1 records. Each RT2 record has up to ten additional interior points. It keeps looping until it finds a point with a latitude of zero and a longitude of zero. See the comment `Reference 1` in the code to see the logic to break out of the loop. Technically, the

point corresponding to this special case is a valid point on the surface of the earth, but it's outside the borders of the United States, so we'll ignore this technicality.

Lastly, we need to determine the city and town names where these streets are located. For this, we'll parse the RTC file, as shown in Listing 11-9. Append this code to the end of your `import_tigerline` Rake task.

Listing 11-9. *Converting the RTC Records into Place Names*

```
  type='rtc'
  i=0
  place_ids = Hash.new
  parser=LineParser.new(root_dir,type)

  # loop through each line of the file
IO.foreach("#{RAILS_ROOT}/lib/chap_eleven/tigerline/#{file_prefix}➥
#{type.upcase}") do |line|

    # parse the line
    l=parser.parse(line)

    place_id="#{state}#{county}#{l[:FIPS]}"
    # don't process if FIPS is blank or NOT type C
    # don't process if we've already seen this place_id
    next if l[:FIPS] == ''
    next if l[:FIPSTYPE] != 'C'
    next if place_ids[place_id]

    place_ids[place_id] = true

    # All looks good. Insert into places
    p=Place.new(:id=>place_id, :state_fips=>state, :county_fips=>county,
                :place_fips=>l[:FIPS], :state_name=>'California',
                :county_name=>'San Francisco',
                :place_name=>l[:NAME])
    p.id=place_id
    p.save

    i=i+1
    puts "imported #{i} #{type} records" #if i % 200 == 0
  end # foreach
```

Here, we're looking for two very simple things: the FIPS code must be present, and the FIPS type must begin with *C*. If these two things are true, the name at the end of the line should be imported into the `places` database table.

For the sake of brevity, we've omitted the sample code for importing alternative spellings and names for the streets as well as importing additional address ranges. We've accounted for them in our data structures, and we'll give you a couple hints about how you could add this easily into your own geocoder:

- For the alternative names, the basic idea is to simply keep doing more of the same parsing techniques while using the RT4 and RT5 records. For each entry in RT4 with a `TLID` for a record we have kept, look up the corresponding `FEAT` records in RT5. When inserting, simply copy the `place_id` from the existing record with the same `TLID` and replace the street name details with the new information.
- Alternative address ranges are even easier. Simply parse the RT6 file looking for matching `TLID` values and insert those address ranges into the `address_ranges` table.

Stepping back to look at the big picture, you now have the data you need to create a geocoder. The rather complicated import process you just went through took the data for a single U.S. county (from TIGER/Line flat-file data), and massaged it into a form you can query.

Next, you will build the query mechanism, in the form of a web service, which takes an address and outputs an XML result with latitude and longitude values.

Building a Geocoding Service

Now we finally get to the fun stuff: the geocoder itself. With a state, a city, a street name, and an address number, you will try to return a corresponding latitude and longitude. As a REST service, your script will expect a format like this:

```
http://localhost:3000/chap_eleven/geo?state=California&city=
San+Francisco&street=Dolores&number=140
```

When you're finished, your service for this address should return something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
  <success>true</success>
  <lat>37.7672215342249</lat>
  <lng>-122.426544056488</lng>
</result>
```

■ **Note** We've chosen this particular address because we have "street truth" data for it. For testing, we selected an address at random and used a GPS device to get a precise latitude and longitude reading. The most accurate information we have for this address is N 37.767367, W 122.426067. As you will see, the geocoder we're about to build has reasonable accuracy (to three decimal places, in this example).

To achieve this, we'll start by finding the correct place by city and state. Then we'll search for the street name and number by joining the `street_names` and `address_ranges` tables. This should give us a single, precise `TLID` to use to look up in the `complete_chains` table. We'll grab all of the latitude and longitude points for the segment and interpolate a single point on the line that represents the address requested. Seems simple, eh? As you'll see in Listing 11-10, the devil is in the details. You can add this code to your existing `lib/tigerline_geocoder.rb` file.

Listing 11-10. *Preliminary U.S. Geocoder Based on TIGER/Line Data*

```
  def geocode(number,street,city,state)
    # variable prep
    number=number.to_i

    # get the place, and only continue if we find it
    # Reference 1
    place=Place.find_by_place_name_and_state_name(city,state)
    if place != nil
      # get the street name and address, and only continue if we find it
      # Reference 2
      address_range=AddressRange.find :first,
        :select => "ar.tlid as tlid, first, last, (#{number}-first) as diff",
        :joins => 'AS ar INNER JOIN street_names AS sn ON ar.tlid=sn.tlid',
        :conditions => ["sn.place_id = ? AND sn.name = ? AND➥
? BETWEEN ar.first AND ar.last", place.id, street, number ],
        :order=>'diff'

      if address_range != nil
        first_address = address_range.first.to_i
        last_address = address_range.last.to_i

        # we now have a single tlid, grab all the points in the chain
        # Reference 3
        chains = CompleteChain.find_all_by_tlid address_range.tlid, :order=> 'seq'

        segment_lengths = Array.new
        num_segments=chains.length-1
        total_length=0
        i=0
        # compute the lengths of all the segments in the chain
        # Reference 4
        while (i < num_segments)
           length=line_length(chains[i].latitude, chains[i].longitude,➥
chains[i+1].latitude, chains[i+1].longitude)
          segment_lengths.push(length)
          total_length += length
          i +=1
        end

        # avoid division-by-zero errors.
        # if total legth is 0, return with the lat/lng of end point
        # Reference 5
        if total_length == 0
          result={:sucess=>true, :lat=>chain[0].latitude, :lng=>chain[0].longitude}
```

```
        else
          # compute how far along the chain our address is
          address_position = (number - first_address).abs.to_f
          num_addresses = (first_address - last_address).abs.to_f
          distance_along_line = (address_position / num_addresses) * total_length

          # figure out which segment our address is in, and interpolate its location
          travel_distance = 0
          i=0
          # Reference 6
          while i < num_segments
            bottom_address = first_address + (travel_distance / total_length *➥
num_addresses)
            travel_distance += segment_lengths[i]

            # Reference 7
            if travel_distance >= distance_along_line
              # we've found our segment, do the final computation
              top_address = first_address + ((travel_distance / total_length) *➥
num_addresses)

              # determine how far along this segment our address is
              # Reference 8
              seg_addr_total = (top_address - bottom_address).abs.to_f
              addr_position = (number - bottom_address).abs / seg_addr_total

              # determine the deltas within this segment
              delta_y = (chains[i+1].latitude - chains[i].latitude).to_f
              delta_x = (chains[i+1].longitude - chains[i].longitude).to_f

              x=chains[i].longitude + delta_x*addr_position
              y=chains[i].latitude + delta_y*addr_position

              segment_num=i+1
              break
            end # end if we've found our segment
            i +=1
          end # end iteration through segments

          # Reference 9
          result={:success=>true, :lat=>y, :lng=>x}
        end # end if total_length == 0
      end # end if address_range != nil
    end # enf if place != nil

    return result
  end
```

```
# helper function to compute the length of a line
def line_length(x1,y1,x2,y2)
  delta_x = (x1-x2).abs
  delta_y = (y1-y2).abs
  return Math.sqrt(delta_x**2 + delta_y**2)
end
```

Keep in mind the big picture for this code: it is the querying mechanism for our geocoder. The `geocode` method in Listing 11-10 does all the heavy lifting. The method takes a street number, a street name, a city, and a state as input. It utilizes the data we have carefully prepared in the import process to translate this input into latitude and longitude values. In the process of doing this, it will utilize all the tables we imported data into: `places`, `street_names`, `address_ranges`, and `complete_chains`.

Next, let's look at the steps the code goes through to accomplish this.

We begin by selecting a place based on the state and city name. If we don't get a match, the geocoder returns failure. If you wanted to make the geocoder more forgiving, you could program a fuzzy match if the exact match fails. See the comment `Reference 1` in the code to understand where this logic resides.

Next, we use MySQL's `BETWEEN` clause to find all of the road segments with our given street name and an address range that bounds our input address. See the comment `Reference 2` in the code to understand where this logic resides. Again, for real production use, you would want a more forgiving matching method if an exact match on a street name failed.

At this point, we should have a single `TLID`. Using this information, we can get the latitude and longitude coordinates of all points on the segment from the `complete_chains` table. See the comment `Reference 3` in the code to understand where this logic resides.

Once we have the list of `complete_chains`, we can start calculating the information we want. We start by using the Pythagorean theorem to compute the length of each line segment in the network chain. This simple equation is implemented in the `line_length` helper function at the end of Listing 11-10, and represented by l_1, l_2, and l_3 in Figure 11-3. See the comment `Reference 4` in the code to see the calls to the `line_length` function. The call is inside a loop because we are summing the distances between multiple points.

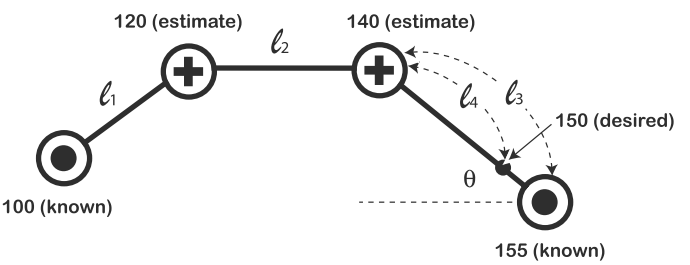


Figure 11-3. *Example of road segment calculations*

However, we immediately run into a problem: very short line segments return a length of zero due to precision problems. To avoid this, and thus increase the accuracy, you might try converting the latitude and longitudes into feet or meters before making your computations, but that conversion process also has its problems. Therefore, if we compute the total length of the chain to be zero, we don't have much choice other than to return one of the end points of the line as our answer. See the `Reference 5` comment in the code to see the logic that handles the case of a zero-length segment.

If we can, we next compute the approximate location of our address (150 in Figure 11-3) along the overall segment. To do this, we assume the addresses are evenly distributed, and calculate our address as a percentage of the total number of addresses and multiply by the total line length.

■ **Caution** For the sake of simplicity, we're making the incorrect assumption that the last address is always larger than the first address. In practice, you'll need to account for this.

So in which segment of the line is our address located? To find out, we walk the line starting from our first end point, using the lengths of line segments we calculated earlier, and keep going until we pass our address. This gives us the top end point, and we simply take the one before it for our bottom end point. See the `Reference 6` comment in the code to see the loop through the segments, and `Reference 7` to see the check for the correct segment.

Once we know which two `complete_chains` points we need to use for our calculations, we again determine (as a percentage) how far along the segment our address is. See the `Reference 8` comment in the code to see where this logic resides. Using this percentage (l_4 in Figure 11-3), we calculate the proper position along the segment for our target address. We return the results in a simple hash structure, with keys `:lat` and `:lng` representing the latitude and longitude, and `:success` indicating that the geocoding operation was successful. See the `Reference 9` comment in the code to see where the result hash is returned.

The only remaining task is to create the REST web service itself. Listing 11-11 shows a simple controller you can add to your `app/controllers/chap_eleven_controller.rb` file to provide a REST interface to your geocoder.

Listing 11-11. *An Action for a REST Web Service*

```
def geo
  result=geocode(params[:number],params[:street], params[:city], params[:state])
  if result
    render :xml=>result.to_xml(:root=>'result')
  else
    render :xml=>{:error=>'No Matches'}.to_xml(:root=>'result')
  end
end
```

And there you have a geocoding web service. Now we need to point out some limitations you'll want to overcome before using this code in production. We've talked about things such as misspellings in the street, state, and place names, as well as division by zero when the segments are very short. Here are a few more issues that we've encountered:

Address ranges that are not integers (i.e., contain alphabet characters): The TIGER/Line documentation suggests that this is a possibility that will break our SQL `BETWEEN` optimization. You could replace the numeric comparison in the SQL with a string-based one. This will mean that an address such as 1100 will match ranges such as 10–20 and 10000–50000. This is due to the character-based comparison used for strings in the database. `BETWEEN` will still help you get a small subset of the database, but you'll need to do more work in Ruby to determine which result is the best match for your query.

Street type or direction separation: We are doing no work to separate out the street type (road, avenue, boulevard, etc.) or the direction (NE, SW, etc.) in our users' input. The street type and direction are stored separately in the database and would help in narrowing down the possible address ranges considerably if we used them. The TIGER/Line documentation enumerates each possible value for these fields, so using them is a matter of finding them in your user's input. You could ask for each part separately, as we have done with the number and street name, or you could use regular expressions, heuristics, and brute force to split a user's string into components. Google's geocoder goes to this effort to great success. It's not trivial, but might be well worth the effort.

Address spacing: We've assumed that all addresses are evenly spaced along our line segment. Since we have the addresses for only the end points, we have no idea which addresses actually exist. There might be as few as two actual addresses on the line, where for a range such as 100–150, we are assuming there are 50. This means that simply because we are able to compute where an address *would* be, we have no idea if it is actually there.

Summary

Creating a robust geocoder is a daunting task and could be the topic of an entire book. Offering it as a service to the general public involves significant bandwidth requirements, severe uptime expectations, and some pretty well-established competition (including Google). However, if you're simply looking for an alternative to paying per lookup, or you've found some source of data that no one has turned into a service yet, it's probably worth your time to build one. The techniques used for interpolating an address based on a range and a multipoint line, as well as finding the closest matching postal code can be widely reused. Even some of the basic ideas for parsing will apply to a wide variety of sources. However, keep in mind that the TIGER/Line data is organized in a rare and strange way and is in no way a worldwide standard. That said, the TIGER/Line data is probably also the most complete single source of free information for the purposes of geocoding. The GML version of the Canadian Road Network File is a distant second.

If you've made it this far, then congratulate yourself. There was some fairly involved mental lifting in this chapter and in the many chapters that came before it. We hope that you put this information to great use and build some excellent new services for the rest of us map builders. If you do, please be sure to let us know, so that we can visit and possibly promote it to other readers via our web site.

PART 4

Appendixes

APPENDIX A

Finding the Data You Want

In order to keep our book a reasonable length, we couldn't possibly use in our examples every neat idea or data source we have found. In Chapter 4, we used information from the Ron Jon Surf Shop chain of stores. We used the U.S. Federal Communications Commission (FCC) Antenna Structure Registration (ASR) database in Chapters 5, 6, and 7. Also, in Chapter 5, we retrieved information on the locations of all of the capital cities in the world (through screen scraping) and used that data in our examples in Chapter 7. In Chapter 11, we used a wide range of street and address data from the U.S. Census Bureau's TIGER/Line data set. There is much more interesting information contained in the TIGER/Line data than roads and geocoding, and we'll touch on that in this appendix.

This appendix contains some of the other interesting sources of data we found while researching this book, as well as some lessons we learned while hunting them down. We hope that this compilation will inspire you to take your new (or at least newly refined) skills and build something exciting and unique. At the very least, these links will give you more interesting data to experiment with while you wait for that killer idea to arrive.

You might consider this as the exercises section of a textbook, or *a problem set*, for you academics out there. Regardless, please drop us a line if you use this information in a map, as we would love to see what you've come up with.

Knowing What to Look For: Search Tips

We covered screen scraping at the end of Chapter 5, but we didn't discuss how to actually find the data you want to scrape or otherwise acquire. Your Googling skills are probably excellent, so we'll cover just a few tips we learned during our own research.

Finding the Information

The data you're typically most interested in is a precompiled list of addresses or latitude/longitude points for some topic. Finding these types of lists can be tricky, and you have to ask yourself, "Besides my idea, does it make sense for anyone to have ever compiled a list like this?" If the answer is no, don't spend too long looking for a single list, and instead focus on finding the precursor information that will help you build the list yourself.

An example would be a list of all of the veterinary clinics in Canada. You probably won't find such a list precompiled from a single source. However, you could use the address data in a phonebook to geocode this kind of information, giving you the list you need. Another problem is that you probably won't find a single phonebook for all of the regions in Canada, and will instead need to assemble the data from various online and CD resources. It's an involved job, but certainly not an impossible one.

Additionally, if you can find whole catalogs of data, they can be both a great source of information for an idea you have, as well as a source of inspiration for expanding on your idea. One such catalog is available from the National Center for Atmospheric Research (NCAR). The wealth of data that is available from `http://dss.ucar.edu/datasets/` is huge, though some of it will be hard to manipulate for amateurs, and other parts are not free. We present the link here as an example of yet another kind of resource you might be able to find.

Specifying Search Terms

Using the term *latitude* in your Google search terms isn't nearly as effective as using *longitude*, or both. Why? Many things (such as temperature, rainfall, vegetation, and population) naturally change with distance north or south of the equator, and people use the term *latitude* in their discussions of such phenomena.

Using *longitude* is much more likely to provide you a *list* or a database of discrete points that you can use in a mapping application. This is, of course, an anecdotal assessment, but it turned out to be very consistent in our research. Another term we found useful is *locations database*. Your mileage may vary.

Also, you'll probably need to keep your search fairly vague to find the information you want. This means *a lot* of results. We found that government or research-related sites are usually the best sources of information. This is typically because they are publicly funded, and the raw data they collect is required to be released to support other research programs. This is definitely a Western point of view, and possibly a North American one, so again, your mileage may vary.

Watching for Errors

Lastly, as you no doubt have learned, you can't trust everything you find on the Internet. Most of the data you find will contain at least a few errors, even if they are just from the transcription process.

Sometimes, the data you find has been scraped from another source, and you'll get a much more accurate and complete set of information if you keep looking and find the original.

If accuracy is a concern, try to corroborate two sources of the same information. An example would be a list of capital cities for all of the countries of the world. Compare the latitude and longitude of the points for each city from each list, and question anything with a deviation in the first or second decimal place.

The Cat Came Back: Revisiting the TIGER/Line

The TIGER/Line data covered in depth in Chapter 11 is much more than just a compilation of roads and addresses. It's a reasonably complete database for all things related to mapping (in our case). For instance, we omitted some of the road segments in Chapter 11 because they

either had no addresses or no name associated with them. However, if we pay careful attention to the CFCCs (Census Feature Class Codes), we find that TIGER gives us mappable information for everything from roads that are accessible only via an all-wheel-drive vehicle, to walking trails, railroads, and rail-transfer yards.

But wait, it gets better. There is also data on landmarks such as parks, churches, prisons, military installations, and hospitals. Table A-1 contains a list of the ones we think might be the most interesting, and page 3-25 in the official TIGER/Line documentation starts off nearly 20 pages of additional options.

Table A-1. *CFCCs for Other Interesting Data in the TIGER/Line Files*

CFCC	Description
A51	Vehicular trail; road passable only by four-wheel-drive vehicle; unseparated
A67	Toll booth barrier to travel
A71	Walkway or trail for pedestrians, usually unnamed
B11	Railroad main track, not in tunnel or underpassing
B14	Abandoned/inactive rail line with tracks present
B19	Railroad main track, bridge
B3*x*	Railroad yard (multiple subclasses exist)
C20	Power transmission line; major category used alone
D10	Military installation or reservation; major category used alone
D21	Apartment building or complex
D23	Trailer court or mobile home park
D24	Marina
D27	Hotel, motel, resort, spa, hostel, YMCA, or YWCA
D28	Campground
D31	Hospital, urgent care facility, or clinic
D33	Nursing home, retirement home, or home for the aged
D35	Orphanage
D36	Jail or detention center
D37	Federal penitentiary or state prison
D42	Convent or monastery
D43	Educational institution, including academy, school, college, and university
D44	Religious institution, including church, synagogue, seminary, temple, and mosque
D45	Museum, including visitor center, cultural center, and tourist attraction
D46	Community center
D47	Library
D51	Airport or airfield
D52	Train station including trolley and mass transit rail system
D53	Bus terminal
D54	Marine terminal

(Continued)

Table A-1. *Continued*

CFCC	Description
D61	Shopping center or major retail center
D64	Amusement center, including arena, auditorium, stadium, coliseum, race course, theme park, and shooting range
D71	Lookout tower
D72	Transmission tower, including cell, radio, and TV
D73	Water tower
D74	Lighthouse beacon
D78	Monument or memorial
D81	Golf course
D82	Cemetery
D83	National Park Service land
D84	National forest or other federal land
D85	State or local park or forest
D86	Zoo
D87	Vineyard, winery, orchard, or other agricultural or horticultural establishment
D88	Landfill, incinerator, dump, spoil, or other location for refuse
E23	Island, identified by name
E27	Dam
Fxx	Nonvisible boundaries, such as political divisions and property lines
Hxx	Water-related polygon (lake), line (river), or point (small ponds)

In Chapter 11, we used mostly data from record type 1 (files with the `.RT1` extensions), and this is where you'll find all of the data (or at least the start of the data) for CFCCs in classes A, B, or C. For the rest of the landmark features, you'll need to use record types 7, 8, and P (files with `.RT7`, `.RT8`, and `.RTP` extensions, respectively). The parsing strategies in Chapter 11 should serve you well in extracting this data into a database for your own use; so if you don't care about building your own geocoder, you might still be interested in reading the first few parts of Chapter 11 so you can get a better handle on how to extract this data.

Lastly, there is one more source of census-related data that we found (we're sure there are some we've missed). It's the summary files for the U.S. 2000 census, located at `http://www.census.gov/prod/cen2000/`. These contain condensed information such as population, number of houses per city, economic data, and other general demographics. They are undoubtedly easier to work with if all you're looking for are these simple statistics. The summary files won't help you find all of the zoos or lighthouses in the United States, but they'll tell you the number of people per house for a given city.

Airports in TIGER/Line

While we did find that the TIGER/Line CFCC D51 denotes an airfield or airport, the TIGER/Line covers only the United States; therefore, we have a few other sources of worldwide information to pass along for this category.

The Global Airport Database is a simple database listing nearly 10,000 airports (both large and small) from around the world. More important, it is explicitly *free*, just like the government data we've been primarily working with here. It can be found at `http://www.partow.net/miscellaneous/airportdatabase/`. It makes no claims to be complete, and we're not sure what the underlying source of the information is, so you might want to cross-reference this with the TIGER/Line data for points inside the United States.

Another interesting source of data related to U.S. airports can be found at the Federal Aviation Administration's site at `http://www.faa.gov/ATS/ata/ata100/120/stdatafiles.html`. The information here ranges from the polygons that define the airspace for each airport to the interairport routes that constitute the nation's highways in the sky.

The Government Standard: The GeoNames Data

When various U.S. government departments and agencies need to refer to geographic entities by name or location, they check the databases maintained by the U.S. Board on Geographic Names. The U.S. Board on Geographic Names is a federal body organized in 1890 and created in its present form by law in 1947. The board's database provides a way to maintain uniform geographic name usage throughout the U.S. federal government. The board comprises representatives of federal agencies concerned with geographic information, population, ecology, and management of public lands. These include, but are not limited to, the Federal Bureau of Investigation (FBI), the Central Intelligence Agency (CIA), the U.S. Geological Survey (USGS), and the U.S. Census Bureau.

This database is very useful for mapping purposes, as it provides (among other things) latitudes and longitudes for many of the world's cities, along with population data. Like the TIGER/Line data, the U.S. Board of Geographic Names data includes churches, schools, monuments, and landmarks. Unlike with the TIGER/Line data, many of these are located outside the United States.

We think this database would make a great cross-reference resource for data found in other places. For example, comparing a list of the locations for cities in Canada with one supplied by the Canadian government would likely weed out any strange anomalies.

The home page for the domestic database is located at `http://geonames.usgs.gov/`. The foreign data is available in raw downloadable form at `http://earth-info.nga.mil/gns/html/index.html`. The folks at GeoNames have gone a long way in converting this data into something you can use quite easily. They also seem to have found and integrated a few other sources of data, such as postal codes for many European countries. They have even done some inspiring Google Maps version 2 mashups, such as the one of the world's most populated cities at `http://www.geonames.org/maps/cities.html`.

Shake, Rattle, and Roll: The NOAA Goldmine

While zooming around the satellite images in Google Maps and Google Earth, we occasionally spotted an active steam or smoke plume coming from a volcano (check out Hawaii's Kilauea), and that led us to hunt for a source of names and locations of current volcanoes. We found a database of more than 1,500 worldwide volcanoes and volcano-related features from the National Geophysical Data Center (NGDC), which is part of the National Oceanographic and Atmospheric Administration (NOAA). Little did we know that this was just the tip of the iceberg of raw, map-oriented data that is available for free download and analysis.

The databases that are available from the various NOAA departments cover everything from volcanoes and earthquakes to hot springs, hurricanes, and hail. They even have a high-resolution data set for the elevation above sea level of each square kilometer of the world's land masses!

We won't talk about each individual data source, since they are all fairly well-documented. Instead, we simply provide a list of links for you to discover what we've uncovered. If you would like to avoid typing each link in to your browser, you can simply visit `http://googlemapsbook.com/appendixa/` and browse from there instead. The first link in each list is the official starting point for the data from the NOAA. The rest are either maps we've found that are based on the same data (Google-based or otherwise), or data we've found for the same topic from other sources. You might want to use secondary sources of data for cross-referencing to weed out errors.

Volcanoes

- `http://www.ngdc.noaa.gov/seg/hazard/volcano.shtml` (data)
- `http://www.volcano.si.edu/world/globallists.cfm` (data)
- `http://www.geocodezip.com/volcanoBrowser.asp` (map)

Earthquakes

- `http://www.ngdc.noaa.gov/seg/hazard/earthqk.shtml` (data)
- `http://earthquake.usgs.gov/eqcenter/recenteqsww/catalogs/` (data)

Tsunamis

- `http://www.ngdc.noaa.gov/seg/hazard/tsu.shtml` (data)
- `http://map.ngdc.noaa.gov/website/seg/hazards/viewer.htm` (map)

Wildfires

- `http://www.ngdc.noaa.gov/seg/hazard/wildfire.shtml` (data)

Hot Springs

- `http://www.ngdc.noaa.gov/seg/geotherm.shtml` (data)
- `http://www.acme.com/jef/hotsprings/` (map)

Hurricanes

- `http://hurricane.csc.noaa.gov/hurricanes/download.html` (data)
- `http://www.nhc.noaa.gov/` (data)
- `http://www.hurricane.com/hurricane-season/hurricane-season-2005.html` (data)
- `http://flhurricane.com/cyclone/stormlist.php?year=2005` (data and maps)

Hail, Tornados, and High Winds

- http://www.spc.noaa.gov/archive/ (data)
- http://www.ems.psu.edu/~nese/ (research data before 1995)
- http://www.stormreportmap.com/ (map)

Geomagnetism and Gravity

- http://www.ngdc.noaa.gov/seg/geomag/geomag.shtml (data)
- http://www.ngdc.noaa.gov/seg/gravity/welcome.shtml (data)

Weather Prediction and Forecasting

- http://www.weather.gov/organization.php (data)
- http://www.spc.noaa.gov/ (data)
- http://www.cpc.noaa.gov/ (data)
- http://www.ncep.noaa.gov/ (data)
- http://api.weatherbug.com/api/ (data)

Worldwide Elevations

- http://www.ngdc.noaa.gov/mgg/topo/globe.html (data)

Everything Else

- http://www.ngdc.noaa.gov/ngdcinfo/onlineaccess.html (data)

One idea we had was to use a variety of the destructive weather databases to create a historic map combining references to encyclopedic articles for many (or all) of the items on the map. It would take some research to find or write the articles and cross-reference the names, but it would be a neat map. This might make a good high-school multidisciplinary project—computer science, geography, and writing skills would all be required. You could even throw in some environmental sciences and math, too.

Another idea would be to combine the databases on tourist-attraction-style features such as hot springs with some other travel-related material. You might even be able to make some money using Google AdSense or other contextual advertising programs.

For the Space Aficionado in You

We're geeks, we admit it. This led us to some interesting ideas for a satellite mashup of the earth's crater impacts, as well as more fiction-related maps such as those of UFO/UAP sightings.

Crater Impacts

We managed to dig up some data that is absolutely screaming to be mashed up. In a sense, it already has been mashed, just not using Google's maps, but rather using the vaguely competing World Wind project. The World Wind project is an open source analog of Google Earth. It takes satellite imagery and topographical data and works them into a desktop application that allows the browser to "fly" around the maps. It was originally started as a project at the National Aeronautics and Space Administration (NASA) and has medium-resolution data (15 meters) driving it. For more information, visit `http://www.worldwindcentral.com`.

The data found at `http://www.worldwindcentral.com/hotspots/` is a lot like the Google sightseeing forum at `http://googlesightseeing.com/`. Visiting the craters category (`http://www.worldwindcentral.com/hotspots/index.php?cat=54`) yields a list of latitude and longitude coordinates, as well as a short blurb submitted by the poster. You might request permission to use this data as the starting point for your own visitor-annotated map using the Google Maps API.

UFO/UAP Sightings

OK, so this is part science and part science fiction/wishful thinking, but we did consider taking the various unidentified flying object (UFO)—alternatively, unidentified aerial phenomena (UAP)—reporting sites and mashing them up using the Google Maps API.

Most of these reporting sites have at least a city and a country associated with them. Using the U.S. Board on Geographic Names data discussed earlier in this appendix, you could easily create a mashup of the individual sightings with at least city-level accuracy. You might even create an "outbreak" style map that adds markers over time, based on the sighting date(s). If you find enough data, using an overlay (see Chapters 7 and 9) might be interesting as well.

The first site we found was the Mutual UFO Network's site at `http://www.mufon.com`. Using the data search tool, we were able to find out about reports on a wide range of criteria. They seem to be limited to state/country location information, but often have images associated with them and long descriptions of the circumstances surrounding the event being reported.

The second is the obvious National UFO Reporting Center at `http://www.nuforc.org/`. This site has a lot of data (hundreds of items per month). However, most of it appears to be uncorroborated, and the site operators state that they have been experiencing problems with falsified reports coming from bored students. Reports include date, time, city, state, type, duration, and eyewitness description. Apparently, the data is only for the United States.

While looking for some data outside the United States with city-level accuracy, the best (though not great) source we found covers a portion of the United Kingdom and seems to stop abruptly in May 2003. We include it here since it does appear to have some interesting data that could be used to cross-reference the data we were not able to find, on the chance that you do. The link is `http://www.uform.org/sightings.htm`. This link should be treated as "volatile" since the most recent data is several years old.

As you can see, there is a huge and wide-ranging array of information that can be used to make your mashup ideas a reality. This list is very U.S.-centric, but it should give readers in (or building maps for) other countries a sense of where they might find the same data in their own governments. Many Western governments have a freedom of information policy that should allow you to obtain some or all of the data, even if they haven't yet made it available online. We wish you luck and success in all of your Google Maps endeavors, and hope that this list of resources can at least provide some inspiration.

APPENDIX B

Google Maps API

This appendix provides a detailed explanation of all the methods, classes, constants, and objects available through the Google Maps API as of version 2.66. For the most up-to-date list, visit `http://www.google.com/apis/maps/documentation/reference.html`.

Note The class, method, and function arguments listed within square brackets are optional.

class GMap2

`GMap2` (aka `GMap`) is the central class in the API. If you've loaded the API using the `v=2` flag, you may also use `GMap` to refer to the `GMap2` class; however, `GMap` is provided only for better backward compatibility, and Google recommends that you modify your code to call `GMap2` to conform with the most current API.

GMap2 Constructor

Constructor	Description
`GMap2(containerDomElement, [opts])`	Instantiating this object creates a new map inside the given DOM element, usually a `div`. The optional `opts` argument should be an instance of `GMapOptions`. If no map types are defined in `opts`, the default `G_DEFAULT_MAP_TYPES` set is used. Likewise, if no size is defined in `opts`, the size of the `containerDomElement` is used. If a size has been defined in `opts`, the `containerDomElement` will be resized accordingly.

GMap2 Methods

Configuration

Method	Returns	Description
`enableDragging()`		Enables the dragging of the map (dragging is enabled by default).
`disableDragging()`		Disables the dragging of the map.
`draggingEnabled()`	`Boolean`	Returns `true` if the map is draggable.
`enableInfoWindow()`		Enables the info window operations for the map (the info window is enabled by default).
`disableInfoWindow()`		Disables the opening of a new info window, and if one is already open, closes the existing one.
`infoWindowEnabled()`	`Boolean`	Returns `true` if the info window is enabled.
`enableDoubleClickZoom()`		Enables double-click to zoom. If enabled, double-clicking with the left mouse button will zoom in on the map, and double-clicking with the right mouse button will zoom out on the map. This overrides the initial functionality of double-clicking to recenter the map. It is disabled by default.
`disableDoubleClickZoom()`		Disables double-click to zoom. See `enableDoubleClickZoom()`.
`doubleClickZoomEnabled()`	`Boolean`	Returns `true` if double-click to zoom is enabled, otherwise returns `false`.
`enableContinuousZoom()`		Enables a smooth zooming transition, similar to the Google Earth desktop software, for Firefox and Internet Explorer browsers running under Windows. By default this is disabled.
`disableContinuousZoom()`		Disables the smooth zooming transition. See `enableContinuousZoom()`.
`continuousZoomEnabled()`	`Boolean`	Returns `true` if smooth zooming transitions are enabled, otherwise returns `false`.

Controls

Method	Returns	Description
`addControl(control, [position])`		Adds the given `GControl` object to the map. The optional position argument should be an instance of the `GControlPosition` class and is used to determine the position of the control on the map. If no position is given, the position of the control will be determined by the `GControl.getDefaultPosition()` method. You can add only one instance of each control to a map.
`removeControl(control)`		Removes the control from the map.
`getContainer()`	`Node`	Returns the HTML DOM object that contains the map (usually a `div`). Called by `GControl.initialize()`.

Map Types

Method	Returns	Description
`getMapTypes()`	Array of `GMapType`	Returns as an array all of the `GMapType` objects registered with the map.
`getCurrentMapType()`	`GMapType`	Returns the `GMapType` object for the currently selected map type.
`setMapType(type)`		Sets the map type for the map. The `GMapType` object for the map type must have been previously added using the `addMapType()` method.
`addMapType(type)`		Adds a new `GMapType` object to the map. See Chapter 9 and the `GMapType` class for more on how to define custom map types.
`removeMapType(type)`		Removes the `GMapType` object from the map.

Map State

Method	Returns	Description
`isLoaded()`	`Boolean`	Returns `true` if the map has been initialized by `setCenter()`.
`getCenter()`	`GLatLng`	Returns the geographical coordinates for the center point of the current viewport.
`getBounds()`	`GLatLngBounds`	Returns the geographical boundary of the map represented by the visible viewport.
`getBoundsZoomLevel(bounds)`	`Number`	Returns the zoom level at which the given `GLatLngBounds` object will fit entirely in the viewport. The zoom level may vary depending on the active map type.
`getSize()`	`GSize`	Returns the size of the map viewport in pixels.
`getZoom()`	`Number`	Returns the current zoom level.

Map State Modifications

Method	Returns	Description
`setCenter(center, [zoom], [type])`		Loads the map centered on the given `GLatLng` with an optional zoom level as an integer and an instance of a `GMapType` object. The map type must have been previously added using the `addMapType()` method and be available in the allowed list defined in the constructor. This method must always be called first after instantiation of the `GMap` object to set the initial state of the map.
`panTo(center)`		Changes the center location of the map. If the given `GLatLng` is already visible elsewhere in the viewport, the pan will be animated as a smooth slide.
`panBy(distance)`		Starts a pan animation, sliding the map by the given `GSize` object.
`panDirection(dx, dy)`		Starts a pan animation, sliding the map by half the width and height in the given direction. `+1` is right and down, and `-1` is left and up.
`setZoom(level)`		Changes the zoom level of the map.
`zoomIn()`		Increases the zoom level by 1. Larger zoom levels are closer to the earth's surface.
`zoomOut()`		Decreases the zoom level by 1. Smaller zoom levels are farther away from the earth's surface.
`savePosition()`		Tells the map to internally store the current map position and zoom level for later retrieval using `returnToSavedPosition()`.
`returnToSavedPosition()`		Restores the map position and zoom level saved by `savePosition()`.
`checkResize()`		Notifies the map of a change of the size of its container. You must call this method if you change the size of the containing DOM element so that the map can adjust itself to fit the new size.

Overlays

Method	Returns	Description
`addOverlay(overlay)`		Adds a `GOverlay` object to the map.
`removeOverlay(overlay)`		Removes a `GOverlay` object from the map. The `removeoverlay()` event is triggered only if the `GOverlay` object exists on the map.
`clearOverlays()`		Removes all `GOverlay` objects from the map.
`getPane(pane)`	`Node`	Returns the `div` DOM element that holds the object in the given `GMapPane` layer.

Info Window

Method	Returns	Description
`openInfoWindow(latlng, dom, [opts])`		Opens an info window at the given `GLatLng` location. If the info window is not fully visible on the map, the map will pan to fit the entire window in the viewport. The content of the info window must be defined using a DOM node.
`openInfoWindowHtml(latlng, html, [opts])`		Opens an info window at the given `GLatLng` location. If the info window is not fully visible on the map, the map will pan to fit the entire window in the viewport. The content of the info window must be defined using an HTML string.
`openInfoWindowTabs(latlng, tabs, [opts])`		Opens a tabbed info window at the given `GLatLng` location. If the info window is not fully visible on the map, the map will pan to fit the entire window in the viewport. The content of the info window must be defined using a DOM node.
`openInfoWindowTabsHtml(latlng, tabs, [opts])`		Opens a tabbed info window at the given `GLatLng` location. If the info window is not fully visible on the map, the map will pan to fit the entire window in the viewport. The content of the info window must be defined using an HTML string.
`showMapBlowup(latlng, [opts])`		Opens an info window at the given `GLatLng`, which contains a close-up view on the map centered on the given `GLatLng`.
`closeInfoWindow()`		Closes the current info window.
`getInfoWindow()`	`GInfoWindow`	Returns the info window object of this map. If no info window exists, it is created but not displayed. `enableInfoWindow()` does not affect the result of `getInfoWindow()`.

Coordinate Transformations

Method	Returns	Description
`fromLatLngToDivPixel(latlng)`	`GPoint`	Returns the `GPoint` pixel coordinates of the given `GLatLng` geographical location, relative to the DOM element that contains the draggable map
`fromDivPixelToLatLng(pixel)`	`GLatLng`	Returns the `GLatLng` geographical coordinates of the given `GPoint` pixel coordinates, relative to the DOM element that contains the draggable map
`fromContainerPixelToLatLng(pixel)`	`GLatLng`	Returns the `GLatLng` geographical coordinates of the given `GPoint` pixel coordinates, relative to the DOM element that contains the map on the page

Events

Event	Arguments	Description
`addmaptype`	`maptype`	Fired when a map type is added to the map using `addMapType()`.
`removemaptype`	`maptype`	Fired when a map type is removed from the map using `removeMapType()`.
`click`	`overlay`, `latlng`	Fired when the map is clicked with the mouse. If the click is on a `GOverlay` object such as a marker, the overlay is passed to the event handler through the `overlay` argument and the overlay's `click` event is fired. If no overlay is clicked, the `GLatLng` location of the click is passed in the `latlng` argument.
`movestart`		Fired when the map tiles begin to move. This will fire when dragging the map with the mouse, in which case a `dragstart` is also fired, or by invoking the movement using one of the `GMap` methods.
`move`		Fired while the map is moving. This event may fire repeatedly as the map moves.
`moveend`		Fired when the map stops moving.
`zoomend`	`oldLevel`, `newLevel`	Fired when the map reaches a new zoom level.
`maptypechanged`		Fired when another map type is selected.
`infowindowopen`		Fired when the info window opens.
`infowindowclose`		Fired when the info window closes. If a currently open info window is reopened at a different point using another call to `openInfoWindow*()`, then `infowindowclose` will fire first.
`addoverlay`	`overlay`	Fired when an overlay is added using `addOverlay()`. The overlay is passed to the event handler.

Event	Arguments	Description
removeoverlay	overlay	Fired when a single overlay is removed by the method removeOverlay(). The overlay that was removed is passed as an argument to the event handler.
clearoverlays		Fired when all overlays are removed by clearOverlays().
mouseover	latlng	Fired when the mouse moves into the map from outside the map. A GLatLng location is passed to the event handler.
mouseout	latlng	Fired when the user moves the mouse off the map. A GLatLng location is passed to the event handler.
mousemove	latlng	Fired when the user moves the mouse inside the map. This event is repeatedly fired while the user moves around the map. A GLatLng location is passed to the event handler.
dragstart		Fired when the user starts dragging the map.
drag		Repeatedly fired while the user drags the map.
dragend		Fired when the user stops dragging the map.
load		Fired when everything on the map has loaded, with the exception of the image tiles, which load asynchronously.

class GMapOptions

The GMapOptions class, instantiated as an object literal, is used to provide optional arguments to the GMap class constructor.

GMapOptions Properties

Property	Type	Description
size	GSize	The size of the map container. If the container is of a different size, the container will be resized to the given GSize. If no size is passed, the map will assume the current size of the container.
mapTypes	Array of GMapType	The array of GMapType constants to allow for the map. If no mapTypes are defined, the constant G_DEFAULT_MAP_TYPES is used.
draggableCursor	String	The cursor to be displayed for draggable maps (since version 2.59).
draggingCursor	String	The cursor to be displayed while the map is being dragged (since version 2.59).

enum GMapPane

As discussed in Chapter 9, the `GMapPane` constants define the various layers of the map used to place overlays and their complementary icons and shadows.

GMapPane Constants

Constant	Description
`G_MAP_MAP_PANE`	The bottom layer, directly on top of the map. Used to hold overlays such as polylines.
`G_MAP_MARKER_SHADOW_PANE`	The pane containing the shadow of the markers. Lies directly beneath the markers.
`G_MAP_MARKER_PANE`	The pane containing the markers.
`G_MAP_FLOAT_SHADOW_PANE`	The pane containing the shadow of the info window. It lies above the `G_MAP_MARKER_PANE` to allow the markers to appear in the shadow of the info window.
`G_MAP_MARKER_MOUSE_TARGET_PANE`	The pane that holds transparent objects that react to the DOM mouse events registered on the overlays. It lies above the `G_MAP_FLOAT_SHADOW_PANE` to allow all the markers on the map to be clickable, even if they lie in the shadow of the info window.
`G_MAP_FLOAT_PANE`	The topmost layer. This pane contains any overlays that appear above all others but under the controls, such as the info window.

class GKeyboardHandler

You can instantiate a `GKeyboardHandler` to add your own keyboard bindings to a map.

GKeyboardHandler Bindings

Key Action	Description
`up`, `down`, `left`, `right`	Continuously moves the map while the key is pressed. If two nonopposing keys are pressed simultaneously, the map will move diagonally.
`page down`, `page up`, `home`, `end`	Triggers an animated pan by three-quarters of the height or width in the corresponding direction.
`+`, `-`	Adjusts the zoom level of the map by one level closer (`+`) or farther away (`-`).

GKeyboardHandler Constructor

Constructor	Description
`GKeyboardHandler(map)`	Creates a keyboard event handler for the given `map`

interface GOverlay

As discussed in detail in Chapters 7 and 9, the `GOverlay` interface is implemented by the `GMarker`, `GPolyline`, and `GInfoWindow` classes, as well as any custom overlays you create. The `GOverlay` instance must be attached to the map using the `GMap2.addOverlay()` method. Upon addition, the map will call the `GOverlay.initialize()` method. Whenever the map display changes, the map will call `GOverlay.redraw()`.

GOverlay Constructor

Constructor	Description
`GOverlay()`	Creates the default implementation of the `GOverlay` methods and should be used when inheriting from the class

GOverlay Static Method

Static Method	Returns	Description
`getZIndex(latitude)`	`Number`	Returns the CSS `z-index` value for the given latitude. By default, overlays that are farther south have higher `z-index` values, so that the overlays will appear stacked when close together.

GOverlay Abstract Methods

Method	Returns	Description
`initialize(map)`		Called by `GMap2.addOverlay()` so the overlay can draw itself into the various panes of the map.
`remove()`		Called by `GMap2.removeOverlay()` and `GMap2.clearOverlays()`. The overlay should use this method to remove itself from the map.
`copy()`	`GOverlay`	Returns an uninitialized copy of itself.
`redraw(force)`		Called when the map display changes. `force` will be `true` only if the zoom level or the pixel offset of the map view has changed.

class GInfoWindow

GInfoWindow is always created by the GMap or GMarker class and accessed by their methods.

GInfoWindow Methods

Method	Returns	Description
selectTab(index)		Selects the tab with the given index.
hide()		Makes the info window invisible but does not remove it from the map.
show()		Makes the info window visible if it's currently invisible.
isHidden()	Boolean	Returns true if the info window is hidden or closed.
reset(latlng, tabs, size, [offset], [selectedTab])		Resets the state of the info window to the given arguments. If the argument value is null, that item will maintain its current value.
getPoint()	GLatLng	Returns the geographical point at which the info window is anchored. The default info window points to this point, modulo the pixel offset.
getPixelOffset()	GSize	Returns the offset, in pixels, of the tip of the info window from the anchor point.
getSelectedTab()	Number	Returns the index of the selected tab. The first leftmost tab is index 0.
getTabs()	Array of GInfoWindowTabs	Returns an array with the tabs in this info window (since version 2.59).
getContentContainers()	Array of Note	Returns an array of raw DOM nodes. The DOM nodes hold the contents of the info window's tabs (since version 2.59).

GInfoWindow Event

Event	Arguments	Description
closeclick		Fired when the info window's close button (X) is clicked

class GInfoWindowTab

Instances of GInfoWindowTab are passed as an array to the tabs argument of GMap2.openInfoWindowTabs(), GMap2.openInfoWindowTabsHtml(), GMarker.openInfoWindowTabs(), and GMarker.openInfoWindowTabsHtml().

GInfoWindowTab Constructor

Constructor	Description
GInfoWindowTab(label, content)	Creates a tab object that can be passed to the tabs argument for all openInfoWindowTabs*() methods. The label is the text that appears on the tab. The content can be either an HTML string or a DOM node, depending on which openInfoWindowTabs*() method you plan to use.

class GInfoWindowOptions

The GInfoWindowOptions class, instantiated as an object literal, is used to provide optional arguments for the GMap and GMarker methods: openInfoWindow(), openInfoWindowHtml(), openInfoWindowTabs(), openInfoWindowTabsHtml(), and showMapBlowup().

GInfoWindowOptions Properties

Property	Type	Description
selectedTab	Number	This sets the window to open at the given tab. The first leftmost tab is index 0. By default, the window will open on tab 0.
maxWidth	Number	This sets the maximum width in pixels of the info window content.
onOpenFn	Function	This is called after the info window has finished opening and the content is displayed.
onCloseFn	Function	This is called when the info window has been closed.
zoomLevel	Number	This is the zoom level of the blowup map in the info window; the argument applies only when using showMapBlowup().
mapType	GMapType	This is the map type of the blowup map in the info window; the argument applies only when using showMapBlowup().

class GMarker

An instance of the GMarker class is used to mark a geographical location on a map. It implements the GOverlay interface and is added to the map using the GMap2.addOverlay() method.

GMarker Constructor

Constructor	Description
GMarker(latlng, [opts])	Creates a new marker at the given GLatLng with optional arguments specified by GMarkerOptions.

GMarker Methods

Method	Returns	Description
`openInfoWindow(content, [opts])`		Opens the info window over the icon of the marker. The `content` of the info window must be defined using a DOM node. Optional arguments are passed using the `GInfoWindowOptions` class.
`openInfoWindowHtml(content, [opts])`		Opens the info window over the icon of the marker. The `content` of the info window must be defined using a string of HTML. Optional arguments are passed using the `GInfoWindowOptions` class.
`openInfoWindowTabs(tabs, [opts])`		Opens the tabbed info window over the icon of the marker. The content of the info window must be defined as an array of `GInfoWindowTab` instances that contain the tab content as DOM nodes. Optional arguments are passed using the `GInfoWindowOptions` class.
`openInfoWindowTabsHtml(tabs, [opts])`		Opens the tabbed info window over the icon of the marker. The content of the info window must be defined as an array of `GInfoWindowTab` instances that contain the tab content as a string of HTML. Optional arguments are passed using the `GInfoWindowOptions` class.
`showMapBlowup([opts])`		Opens the info window over the icon of the marker. The content of the info window becomes a close-up of the area around the info window's anchor. Optional arguments are passed using the `GInfoWindowOptions` class.
`getIcon()`	`GIcon`	Returns the `GIcon` associated with this marker as defined in the constructor.
`getPoint()`	`GLatLng`	Returns the `GLatLng` geographical coordinates of the marker's anchor. The anchor is set by the constructor or modified by `setPoint()`.
`setPoint(latlng)`		Sets the geographical coordinates of the marker's anchor to the given `GLatLng` instance.
`enableDragging()`		Enables dragging/dropping the marker on the map. The marker must be initialized with `GMarkerOptions.draggable = true` for this call to have any effect.
`disableDragging()`		Turns off dragging/dropping for the marker.
`Draggable()`	`Boolean`	Returns `true` if the marker is initialized with the option `GMarkerOptions.draggable = true`, otherwise returns `false`.
`draggingEnabled()`	`Boolean`	Returns `true` if dragging/dropping is currently enabled for the marker.

GMarker Events

Event	Arguments	Description
`click`		Fired when the marker is clicked with the mouse. The `GMap`'s `click` event will also fire with the marker passed as the overlay argument.
`dblclick`		Fired when the marker icon is double-clicked.
`mousedown`		Fired when the DOM `mousedown` event is fired on the marker icon.
`mouseup`		Fired for the DOM `mouseup` on the marker.
`mouseover`		Fired when the mouse moves into the area of the marker icon.
`mouseout`		Fired when the mouse moves out of the area of the marker icon.
`infowindowopen`		Fired when the info window of the map is opened using one of the `GMarker` info window methods.
`infowindowclose`		Fired when the info window, opened using `GMarker.OpenInfoWindow*()`, is closed or if the info window is opened on another marker.
`remove`		Fired when the marker is removed from the map.
`dragstart`		Fired when the user starts to drag the marker.
`drag`		Fires repeatedly while the marker is being dragged.
`dragend`		Fired when the user finishes dragging the marker.

class GMarkerOptions

The `GMarkerOptions` class, instantiated as an object literal, is used to provide optional arguments for the `GMarker` class.

GMarkerOptions Properties

Property	Type	Description
`icon`	`GIcon`	This is an instance of the `GIcon` class. If not specified, `G_DEFAULT_ICON` is used.
`dragCrossMove`	`Boolean`	Setting this value to `true` keeps the marker underneath the mouse cursor and moves the cross downward. Normally, markers float up and away from the mouse cursor while being dragged. The default is `false` (since version 2.63).
`title`	`String`	The title will appear as a tool tip on the marker, like the `title` attribute on HTML elements.
`clickable`	`Boolean`	If set to `false`, the marker becomes inert and consumes fewer resources. Inert markers will not respond to any events. By default, this option is `true` and markers are clickable.
`draggable`	`Boolean`	If set to `true`, the marker is draggable by users. Note that draggable markers consume more resources, so only enable dragging if necessary. The default is `false`. Markers that are draggable are made clickable and bouncy by default (since version 2.61).

Continued

GMarkerOptions Properties *(Continued)*

Property	Type	Description
bouncy	Boolean	If set to true, the marker will bounce up and down after the dragging operation is done. The default is false (since version 2.61).
bounceGravity	Number	This number specifies the acceleration rate of the marker when it drops back to earth after dragging is complete. The default is 1 (since version 2.61).

class GPolyline

If available, the GPolyline class draws a polyline on the map using the browser's built-in vector-drawing facilities. Otherwise, the polyline is drawn using an image from Google servers.

GPolyline Constructor

Constructor	Description
GPolyline(points, [color], [weight], [opacity])	Creates a polyline from the array of GLatLng instances. Optionally, the color of the line can be defined as a string in the hexadecimal format RRGGBB; the weight can be defined in pixels; and the opacity can be defined as a number between 0 and 1, where 0 is transparent and 1 is opaque.

GPolyline Factory Methods

Method	Returns	Description
fromEncoded(color, weight, opacity, points, zoomFactor, levels, numLevels)	GPolyline	Creates an antialiased, semitransparent polyline from specially encoded strings. color contains a hexadecimal numeric HTML style, that is, #RRGGBB. weight is the width of the line in pixels. opacity is a number between 0 and 1. points contains the encoded latitude and longitude coordinates in encoded string form. levels is a string containing the encoded polyline zoom level groups. numLevels specifies the number of zoom levels contained in the encoded levels string. zoomFactor specifies the magnification between sets of zoom levels in the levels string (since version 2.63).

GPolyline Methods

Method	Returns	Description
getVertexCount()	Number	Returns the number of vertices in the polyline
getVertex(index)	GLatLng	Returns the vertex with the given index in the polyline starting at 0 for the first vertex

GPolyline Event

Event	Arguments	Description
remove		Fired when the polyline is removed from the map

class GIcon

The GIcon class specifies the image to display as the icon for the GMarker on the map. If no icon is specified, G_DEFAULT_ICON is used.

GIcon Constructor

Constructor	Description
GIcon([copy], [image])	Creates a new GIcon object. Existing GIcon's properties can be copied by passing the existing icon into the copy argument. The optional image argument can be used as a shortcut to the image property.

GIcon Constant

The G_DEFAULT_ICON constant specifies the default marker icon.

GIcon Properties

Property	Type	Description
image	String	URL for the foreground image.
shadow	String	URL for the shadow image.
iconSize	GSize	The pixel size of the foreground image.
shadowSize	GSize	The pixel size of the shadow image.
iconAnchor	GPoint	The pixel coordinates of the image's anchor relative to the top left corner of the image.
infoWindowAnchor	GPoint	The pixel coordinates of the point where the info window will be anchored, relative to the top left corner of the image.
printImage	String	URL of the foreground image used for printed maps. It must be the same size as the image property.
mozPrintImage	String	The URL of the foreground icon image used for printed maps in Firefox/Mozilla. It must be the same size as the image property.
printShadow	String	The URL of the shadow image used for printed maps. Most browsers can't accurately print PNG transparency, so this property should be a GIF.
transparent	String	The URL used to represent the clickable part of the icon in Internet Explorer. This should be a URL to a 24-bit PNG version of the main icon image with 1% opacity and the same shape and size as the image property.
imageMap	Array of Number	The URL used to represent the clickable part of the icon in browsers other than Internet Explorer. This should be an array of integers representing the X and Y coordinates of the clickable image area.

class GPoint

In version 1 of the API, a `GPoint` represented a geographical latitude and longitude. In version 2 of the API, a `GPoint` represents a point on the map by its *pixel coordinates*. Now, for geographical latitude and longitude, see the `GLatLng` class.

Unlike regular HTML DOM elements, the map coordinates increase to the left and down, so the X coordinate increases as objects are farther west, and the Y coordinate increases as objects are farther south.

■ **Note** Although the `x` and `y` properties are accessible and modifiable, Google recommends you always create a new `GPoint` instance and avoid modifying an existing one.

GPoint Constructor

Constructor	Description
`GPoint(x,y)`	Creates a `GPoint` object

GPoint Properties

Property	Type	Description
`x`	`Number`	X coordinate, increases to the left
`y`	`Number`	Y coordinate, increases downward

GPoint Methods

Method	Returns	Description
`equals(other)`	`Boolean`	Returns `true` if the other given `GPoint` has equal coordinates
`toString()`	`String`	Returns a string that contains the X and Y coordinates, separated by a comma and surrounded by parentheses, in the form `(x,y)`

class GSize

A `GSize` is a width and height definition, in pixels, of a rectangular area on the map. Note that although the `width` and `height` properties are accessible and modifiable, Google recommends that you always create a new `GSize` instance and avoid modifying an existing one.

GSize Constructor

Constructor	Description
`GSize(width,height)`	Creates a `GSize` object

GSize Properties

Property	Type	Description
width	Number	The width in pixels
height	Number	The height in pixels

GSize Methods

Method	Returns	Description
equals(other)	Boolean	Returns true if the other given GSize has exactly equal components
toString()	String	Returns a string that contains the width and height coordinates, separated by a comma and surrounded by parentheses, in the form (width,height)

class GBounds

A GBounds instance represents a rectangular area of the map in pixel coordinates. The GLatLngBounds class represents a rectangle in geographical coordinates.

GBounds Constructor

Constructor	Description
GBounds(points)	Constructs a rectangle that contains all the given points in the points array

GBounds Properties

Property	Type	Description
minX	Number	The X coordinate of the left edge of the rectangle
minY	Number	The Y coordinate of the top edge of the rectangle
maxX	Number	The X coordinate of the right edge of the rectangle
maxY	Number	The Y coordinate of the bottom edge of the rectangle

GBounds Methods

Method	Returns	Description
toString()	String	Returns a string containing the northwest and the southeast corners of the area separated by a comma, surrounded by parentheses, in the form (nw,se)
min()	GPoint	Represents the point at the upper left corner of the box
max()	GPoint	Represents the point at the lower right corner of the box
containsBounds(other)	Boolean	Returns true if the other GBounds is entirely contained in this GBounds
extend(point)		Increases the size of the bounds so the given GPoint is also contained in the bounds
intersection(other)	GBounds	Returns a new GBounds object that represents the overlapping portion of this and the given GBounds

class GLatLng

A GLatLng instance represents a geographical longitude and latitude on the map projection.

■ **Note** Although longitude is representative of an X coordinate on a map, and latitude with the Y coordinate, Google has chosen to follow customary cartography terminology where the latitude coordinate is written first, followed by the longitude as represented in the GLatLng constructor arguments.

GLatLng Constructor

Constructor	Description
GLatLng(lat, lng, [unbounded])	Creates a new GLatLng instance. If the unbounded flag is true, the latitude and longitude will be used as passed. Otherwise, latitude will be restricted to between –90 degrees and +90 degrees, and longitude will be wrapped to lie between –180 degrees and +180 degrees.

GLatLng Properties

There are a few GLatLng properties; however, they exist only for backward compatibility with version 1 of the API. Therefore, we do not list them here. If you would like to reference them, see Google's online documentation at http://www.google.com/apis/maps/documentation/reference.html#GLatLng.

GLatLng Methods

Method	Returns	Description
lat()	Number	Returns the latitude coordinate in degrees.
lng()	Number	Returns the longitude coordinate in degrees.
latRadians()	Number	Returns the latitude coordinate in radians, as a number between –PI/2 and +PI/2.
lngRadians()	Number	Returns the longitude coordinate in radians, as a number between –PI and +PI.
equals(other)	Boolean	Returns true if the other GLatLng has equal components (within an internal round-off accuracy).
distanceFrom(other)	Number	Returns the distance, in meters, from this GLatLng to the other GLatLng. Google's API approximates the earth as a sphere, so the distance could be off by as much as 0.3%.
toUrlValue()	String	Returns a string representation of this point that can be used as a URL parameter value. The string is formatted with the latitude and the longitude in degrees rounded to six decimal digits, separated by a comma, without whitespace.

class GLatLngBounds

A GLatLngBounds instance represents a rectangle in geographical coordinates. The GBounds class represents a rectangle in pixel coordinates.

GLatLngBounds Constructor

Constructor	Description
GLatLngBounds([sw], [ne])	Creates a new instance of GLatLngBounds with a boundary defined by the southwest and northeast corners

GLatLngBounds Methods

Method	Returns	Description
equals(other)	Boolean	Returns true if the other GLatLngBounds has equal components (within an internal round-off accuracy).
contains(latlng)	Boolean	Returns true if the geographical coordinates of the given GLatLng lie within the boundary.
intersects(other)	Boolean	Returns true if the given GLatLngBounds intersects this GLatLngBounds.
containsBounds(other)	Boolean	Returns true if the given GLatLngBounds is contained entirely within this GLatLngBounds.
extend(latlng)		Increases the size of the bounds so the given GLatLng is also contained in the bounds. When calculating the longitude change, the bounds will be enlarged in the smaller of the two possible ways given the wrapping of the map. If both directions are equal, the bounds will extend at the eastern boundary.
getSouthWest()	GLatLng	Returns the latitude and longitude at the southwest corner of the rectangle.
getNorthEast()	GLatLng	Returns the latitude and longitude at the northeast corner of the rectangle.
toSpan()	GLatLng	Returns a GLatLng with latitude and longitude degrees representing the height and width, respectively.
isFullLat()	Boolean	Returns true if this boundary extends the full height of the map, from the south pole to the north pole.
isFullLng()	Boolean	Returns true if this boundary extends fully around the earth.
isEmpty()	Boolean	Returns true if this boundary is empty.
getCenter()	GLatLng	Returns the center point of the rectangle.

interface GControl

As discussed in Chapter 9, the GControl interface is implemented by all control objects, and implementations must be added to the maps using the GMap2.addControl() method.

GControl Constructor

Constructor	Description
`GControl([printable], [selectable])`	Creates the prototype instance for a new control class. If the `printable` flag is `true`, the control will appear when printed. Use the `selectable` argument to indicate if the control contains text that should be selectable.

GControl Methods

Method	Returns	Description
`printable()`	`Boolean`	Returns `true` to the map if the control should be printable, otherwise returns `false`.
`selectable()`	`Boolean`	Returns `true` to the map if the control contains selectable text, otherwise returns `false`.
`initialize(map)`	`Node`	Initializes the map. This method is called by `GMap2.addControl()`; there is no need to call it manually.
`getDefaultPosition()`	`GControlPosition`	Returns to the map the `GControlPosition` representing where the control appears by default. This can be overridden by the second argument to `GMap2.addControl()`.

class GControl

The following are existing instances of the `GControl` interface.

GControl Constructors

Constructor	Description
`GSmallMapControl()`	Creates a control with buttons to pan in four directions and zoom in and zoom out
`GLargeMapControl()`	Creates a control with buttons to pan in four directions and zoom in and zoom out, and creates a zoom slider
`GSmallZoomControl()`	Creates a control with buttons to zoom in and zoom out
`GScaleControl()`	Creates a control that displays the map scale
`GMapTypeControl()`	Creates a control with buttons to switch between map types

class GControlPosition

The `GControlPosition` class describes the position of a control in the map container. A corner from one of the `GControlAnchor` constants and an offset relative to that anchor determine the position.

GControlPosition Constructor

Constructor	Description
`GControlPosition(anchor, offset)`	Creates a new control position

enum GControlAnchor

The `GControlAnchor` constants are used to reference the position of a `GControl` within the map viewport. You will need these if you are creating your own control objects, as discussed in Chapter 9.

GControlAnchor Constants

Constant	Description
`G_ANCHOR_TOP_RIGHT`	Anchored in the top right corner of the map
`G_ANCHOR_TOP_LEFT`	Anchored in the top left corner of the map
`G_ANCHOR_BOTTOM_RIGHT`	Anchored in the bottom right corner of the map
`G_ANCHOR_BOTTOM_LEFT`	Anchored in the bottom left corner of the map

class GMapType

As discussed in Chapter 9, the `GMapType` is the grouping of a map projection and tile layers.

GMapType Constructor

Constructor	Description
`GMapType(layers, projection, name, [opts])`	Creates a new `GMapType` instance with the given layer array of `GTileLayers`, the given `GProjection`, a name for the map type control, and optional arguments from `GMapTypeOptions`.

GMapType Methods

Method	Returns	Description
`getSpanZoomLevel (center, span, viewSize)`	`Number`	Returns the zoom level at which the `GLatLng span`, centered on the `GLatLng center`, will fit in the `GSize` defined by `viewSize`.
`getBoundsZoomLevel (latlngBounds, viewSize)`		Returns the zoom level at which the `GLatLngBounds` will fit in the `GSize` defined by `viewSize`.
`getName(short)`	`String`	Returns the name of the map type. If `short` is `true`, the short name will be returned; otherwise, the full name will be returned.
`getProjection()`	`GProjection`	Returns the `GProjection` instance.

Continued

GMapType Methods *(Continued)*

Method	Returns	Description
`getTileSize()`	`Number`	Returns the tile size in pixels. The tiles are assumed to be quadratic, and all tile layers have the same tile size.
`getTileLayers()`	Array of `GTileLayer`	Returns the array of tile layers.
`getMinimumResolution([latlng])`	`Number`	Returns the lowest zoom level.
`getMaximumResolution([latlng])`	`Number`	Returns the highest zoom level.
`getTextColor()`	`String`	Returns the color that should be used for text, such as the copyright, overlaid on the map.
`getLinkColor()`	`String`	Returns the color that should be used for a hyperlink overlaid on the map.
`getErrorMessage()`	`String`	Returns the error message to display on zoom level where this map type doesn't have any map tiles.
`getCopyrights(bounds, zoom)`	Array of `String`	Returns the copyright messages appropriate for the given `GLatLngBounds` bounds at the given zoom level.
`getUrlArg()`	`String`	Returns a value that can be used as a URL parameter value to identify this map type in the current map view. Useful for identifying maps and returning to the same location via hyperlinks in web applications.

GMapType Constants

Constant	Description
`G_NORMAL_MAP`	The normal street map type
`G_SATELLITE_MAP`	The Google Earth satellite images
`G_HYBRID_MAP`	The transparent street maps over Google Earth satellite images
`G_DEFAULT_MAP_TYPES`	An array of `G_NORMAL_MAP`, `G_SATELLITE_MAP`, and `G_HYBRID_MAP`

GMapType Event

Event	Argument	Description
`newcopyright`	`copyright`	Fired when a new `GCopyright` instance is added to the `GCopyrightCollection` associated with one of the tile layers contained in the map type

class GMapTypeOptions

The `GMapTypeOptions` class, instantiated as an object literal, is used to provide optional arguments for the `GMapType` constructor.

GMapTypeOptions Properties

Property	Type	Description
shortName	String	The short name that is returned from GMapType.getName(true). The default is the same as the name from the constructor.
urlArg	String	The URL argument that is returned from GMapType.getUrlArg(). The default is an empty string.
maxResolution	Number	The maximum zoom level. The default is the maximum from all tile layers.
minResolution	Number	The minimum zoom level. The default is the minimum from all tile layers.
tileSize	Number	The tile size for the tile layers. The default is 256.
textColor	String	The text color returned by GMapType.getTextColor(). The default is "black".
linkColor	String	The text color returned by GMapType.getLinkColor(). The default is "#7777cc".
errorMessage	String	The error message returned by GMapType.getErrorMessage(). The default is an empty string.
alt	String	The alternative map type represented as a text string.

interface GTileLayer

As explained in Chapters 7 and 9, you use the GTileLayer interface to implement your own custom tile layers.

GTileLayer Constructor

Constructor	Description
GTileLayer(copyrights, minResolution, maxResolution)	Creates a new tile layer instance. The arguments for the constructor can be omitted if instantiated as a prototype for your custom tile layer. copyrights is an array of GCopyright objects. minResolution and maxResolution refer to the minimum and maximum zoom levels, respectively.

GTileLayer Methods

Method	Returns	Description
minResolution()	Number	Returns the lowest zoom level for the layer.
maxResolution()	Number	Returns the highest zoom level for the layer.
getTileUrl(tile, zoom)	String	Returns the URL of the map tile. tile is a GPoint representing the x and y tile index. zoom is the current zoom level of the map. It must be implemented for custom tile layers.
isPng()	Boolean	Returns true if the tiles are PNG images; otherwise, GIF is assumed. It is abstract and must be implemented for custom tile layers.
getOpacity()	Number	Returns the layer opacity between 0 and 1, where 0 is transparent and 1 is opaque. It is abstract and must be implemented in custom tile layers.

GTileLayer Event

Event	Argument	Description
newcopyright	copyright	Fired when a new GCopyright instance is added to the GCopyrightCollection of this tile layer

class GCopyrightCollection

The GCopyrightCollection is a collection of GCopyright objects for the current tile layer(s).

GCopyrightCollection Constructor

Constructor	Description
GCopyrightCollection([prefix])	Creates a new copyright collection. If the prefix argument is defined, the copyright messages all share the same given prefix

GCopyrightCollection Methods

Method	Returns	Description
addCopyright(copyright)		Adds the GCopyright object to the collection
getCopyrights(bounds, zoom)	Array of String	Returns all copyrights for the given GLatLng bounds at the given zoom level
getCopyrightNotice(bounds, zoom)	String	Returns the prefix concatenated with all copyrights for the given GLatLng bounds at the given zoom level, separated by commas

GCopyrightCollection Event

Event	Argument	Description
newcopyright	copyright	Fired when a new GCopyright is added to the GCopyrightCollection

class GCopyright

The GCopyright class defines which copyright message applies to a boundary on the map at a given zoom level.

GCopyright Constructor

Constructor	Description
GCopyright(id, bounds, minZoom, copyrightText)	Creates a new GCopyright object with the given id, the given GLatLng bounds, and the minimum zoom level with which the copyright applies

GCopyright Properties

Property	Type	Description
id	Number	A unique identifier
minZoom	Number	The lowest zoom level at which this information applies
bounds	GLatLngBounds	The latitude and longitude boundary for the copyright
text	String	The copyright message

interface GProjection

As explained in Chapter 9, the GProjection interface is responsible for all the mathematical calculations related to placing objects on the map. The GMercatorProjection, for example, is used by all predefined map types and calculates geographical positions based on the Mercator mapping projection.

GProjection Methods

Method	Returns	Description
fromLatLngToPixel(latlng, zoom)	GPoint	Returns the map coordinates in pixels from the given GLatLng geographical coordinates and the given zoom level.
fromPixelToLatLng(point, zoom, [unbounded])	GLatLng	Returns the geographical coordinates for the given GPoint and the given zoom level. The unbounded flag, when true, prevents the geographical longitude coordinate from wrapping when beyond the –180 or +180 degrees meridian.
tileCheckRange(tile, zoom, tilesize)		Returns true if the index of the tile given in the tile GPoint is in a valid range for the map type and zoom level. If false is returned, the map will display an empty tile. In some cases where the map wraps past the meridian, you may modify the tile index to point to an existing tile.
getWrapWidth(zoom)		Returns the number of pixels after which the map repeats itself in the longitudinal direction. The default is Infinity, and the map will not repeat itself.

class GMercatorProjection

This GMercatorProjection class is an implementation of the GProjection interface and is used by all the predefined GMapType objects.

GMercatorProjection Constructor

Constructor	Description
GMercatorProjection(zoomlevels)	Creates a GProjection object based on the Mercator projection for the given number of zoom levels

GMercatorProjection Methods

Method	Returns	Description
fromLatLngToPixel(latlng, zoom)	GPoint	See GProjection
fromPixelToLatLng(pixel, zoom, [unbounded])	GLatLng	See GProjection
checkTileRange(tile, zoom, tilesize)		See GProjection
getWrapWidth(zoom)		See GProjection. Returns the width of the map for the entire earth, in pixels, for the given zoom level

namespace GEvent

The GEvent namespace contains the methods you need to register and trigger event listeners on objects and DOM elements. The events defined by the Google Maps API are all custom events and are fired internally using GEvent.triggerEvent().

GEvent Static Methods

Static Method	Returns	Description
addListener (object, event, handler)	GEventListener	Registers an event handler for the event on the object. Returns the GEventListener handle that can be used to deregister the handler with GEvent.removeListener(). When referencing 'this' from within the supplied handler function, 'this' will refer to the JavaScript object supplied in the first argument.
addDomListener (dom, event, handler)	GEventListener	Registers an event handler for the event on the DOM object. Returns the GEventListener handle that can be used to deregister the handler with GEvent.removeListener(). When referencing 'this' from within the supplied handler function, 'this' will refer to the DOM object supplied in the first argument.

Static Method	Returns	Description
`removeListener(handler)`		Removes the handler. The handler must have been created using `addListener()` or `addDomListener()`.
`clearListeners(source, event)`		Removes all handlers on the given source object or DOM for the given event that were registered using `addListener()` or `addDomListener()`.
`clearInstanceListeners(source)`		Removes all handlers on the given object or DOM for all events registered using `addListener()` or `addDomListener()`.
`trigger(source, event, ...)`		Fires the given event on the source object. Any additional arguments after the event are passed as arguments to the event-handler functions.
`bind(source, event, object, method)`		Registers the specified method on the given object as the event handler for the custom event on the given source object. You can then use the `trigger()` method to execute the event.
`bindDom(source, event, object, method)`		Registers the specified method on the given object as the event handler for the custom event on the given source object. Unlike `bind()`, the source object must be an HTML DOM element. You can then use the `trigger()` method to execute the event.
`callback(object, method)`		Calls the given method on the given object.
`callbackArgs (object, method, ...)`		Calls the given method on the given object with the given arguments.

GEvent Event

Event	Argument	Description
`clearlisteners`	`event`	Fired for the object when `clearListeners()` or `clearInstanceListeners()` is called on that object

class GEventListener

The `GEventListener` class is opaque. There are no methods or constructor. Instances of the `GEventListener` are returned only from `GEvent.addListener()` and `GEvent.addDomListener()`. Instances of `GEventListener` can also be passed back to `GEvent.removeListener()` to disable the listener.

namespace GXmlHttp

The GXmlHttp namespace provides a browser-agnostic factory method to create an XmlHttpRequest (Ajax) object.

GXmlHttp Static Method

Static Method	Returns	Description
create()	GXmlHttp	This is a factory method for creating new instances of XmlHttpRequest

namespace GXml

The GXml namespace provides browser-agnostic methods to handle XML. The methods will function correctly only in browsers that natively support XML.

GXml Static Methods

Static Method	Returns	Description
parse(xmlString)	Node	Parses the given XML string into a DOM representation. In the event that the browser doesn't support XML, the method returns the DOM node of an empty div element.
value(xmlDom)	String	Returns the text value of the XML document fragment given in DOM representation.

class GXslt

The GXslt class provides browser-agnostic methods to apply XSLT to XML. The methods will function correctly only in browsers that natively support XSL.

GXslt Static Methods

Static Method	Returns	Description
create(xsltDom)	GXslt	Creates a new GXslt instance from the DOM representation of an XSLT style sheet.
transformToHtml	Boolean	Transforms the xmlNode DOM representation(xmlDom, htmlDom) of the XML document using the XSLT from the constructor. The resulting HTML DOM object will be appended to the htmlDom. In the event that the browser does not support XSL, this method will do nothing and return false.

namespace GLog

The GLog namespace is not directly related to the mapping functions of the map but is provided to help you debug your web applications. As discussed in Chapter 9, you can use the write*() methods to open a floating log window and record and debug messages.

GLog Static Methods

Static Method	Returns	Description
`write(message, [color])`		Writes a message to the log as plain-text. The message text will be escaped so HTML characters appear as visible characters in the log window.
`writeUrl(url)`		Writes a URL to the log as a clickable link.
`writeHtml(html)`		Writes HTML to the log as rendered HTML (not escaped).

class GDraggableObject

Use this class to make a DOM element draggable. Note that changing the static members representing the drag cursor will affect all subsequently created draggable objects.

GDraggableObject Static Methods

Static Method	Returns	Description
`setDraggableCursor(cursor)`		Sets the hover-over cursor for subsequently created draggable objects
`setDraggingCursor(cursor)`		Sets the cursor for subsequently created draggable objects, to be displayed while the object is being dragged

GDraggableObject Constructor

Constructor	Description
`GDraggableObject(src, [opts])`	Sets up the necessary event handlers so the source element can be dragged (since version 2.59)

GDraggableObject Properties

Property	Type	Description
`left`	`Number`	The object's left starting position
`top`	`Number`	The object's top starting position
`container`	`Node`	A DOM element, outside of which the object cannot be dragged
`draggingCursor`	`String`	The cursor to show on hover
`draggingCursor`	`String`	The cursor to show while dragging

GDraggableObject Methods

Method	Returns	Description
`setDraggableCursor(cursor)`		Sets the hover-over cursor for this individual object
`setDraggingCursor(cursor)`		Sets the cursor for this individual draggable object, to be displayed while the object is being dragged

enum GGeoStatusCode

The `GGeoStatusCode` constants are returned from the geocoder.

GGeoStatusCode Constants

Constant	Description
`G_GEO_SUCCESS`	The supplied address was successfully recognized and no errors were reported
`G_GEO_SERVER_ERROR`	The server failed to process the request
`G_GEO_MISSING_ADDRESS`	The address is `null`
`G_GEO_UNKNOWN_ADDRESS`	The supplied address could not be found
`G_UNAVAILABLE_ADDRESS`	The address was found; however, it cannot be exposed by Google for legal reasons
`G_GEO_BAD_KEY`	The supplied API key is invalid

enum GGeoAddressAccuracy

There are no symbolic constants defined for this enumeration.

Constant	Description
`0`	Unknown location
`1`	Country-level accuracy
`2`	Region-level accuracy
`3`	Subregion-level accuracy
`4`	Town-level accuracy
`5`	Postal code/ZIP code–level accuracy
`6`	Street-level accuracy
`7`	Intersection-level accuracy
`8`	Address-level accuracy

class GClientGeocoder

Use the `GClientGeocoder` class to geocode addresses using Google's geocoding service.

GClientGeocoder Constructor

Constructor	Description
`GClientGeocoder([cache])`	Creates a new instance of a geocoder. You may optionally supply your own client-side `GFactualGeocodeCache` object.

GClientGeocoder Methods

Method	Returns	Description
`getLatLng(address, callback)`		Retrieves the latitude and longitude of the supplied address. If successful, the callback function receives a populated `GLatLng` object. If the address can't be found, the callback receives a `null` value.
`getLocations(address, callback)`		Retrieves one or more geocode locations based on the supplied address and passes them as a response object to the callback function. The response contains a `Status` property (`response.Status`) that can be examined to determine whether the response was successful.
`getCache()`	`GGeocodeCache`	Returns the cache in use by the geocoder instance.
`setCache(cache)`		Tells the geocoder instance to discard the current cache and use the supplied `GGeocodeCache` cache object. If `null` is passed, caching will be disabled.
`reset()`		Resets the geocoder and the cache.

class GGeocodeCache

Use the `GGeocodeCache` class to create a cache for `GClientGeocoder` requests.

GGeocodeCache Constructor

Constructor	Description
`GGeocodeCache()`	Creates a new cache object for storing encoded addresses. When instantiated, the constructor calls `reset()`.

GGeocodeCache Methods

Method	Returns	Description
`get(address)`	`Object`	Retrieves the stored response for the given address. If the address can't be found, it will return `null`.
`isCachable(reply)`	`Boolean`	Determines whether the given address should be cached. This method is used to avoid caching `null` or invalid responses and can be extended in your custom cache objects to provide more control of the cache.
`put(address, reply)`		Stores the given reply/address combination in the cache based on the results of the `isCacheable()` and `toCanonical()` methods.
`reset()`		Empties the cache.
`toCanonical(address)`	`String`	Returns a canonical version of the address by converting the address to lowercase and stripping out commas and extra spaces.

class GFactualGeocodeCache

The GFactualGeocodeCache class is a stricter version of the GGeocodeCache class. It restricts the cache to replies that are unlikely to change within a short period of time.

GFactualGeocodeCache Constructor

Constructor	Description
GFactualGeocodeCache()	Creates a new instance of the cache

GFactualGeocodeCache Method

Method	Returns	Description
isCachable(reply)	Boolean	Set this to true to perform a more thorough check of the status code. When true, it only replies with Status.code of G_GEO_SUCCESS, or replies that are known to be invalid are considered cacheable. Replies that time-out or result in a generic server error will not be cached.

class GMarkerManager

Use the GMarkerManager to manage the visibility of hundreds of markers on the map, based on the map's size and zoom level.

GMarkerManager Constructor

Constructor	Description
GMarkerManager(map, [opts])	Creates a new marker manager (since version 2.67)

GMarkerManager Methods

Method	Returns	Description
addMarkers(markers, minZoom, maxZoom)		Adds a batch of markers. You must call the refresh method on the marker manager for the markers to actually appear. Once placed on the map, markers are only shown when the map is within the minZoom and maxZoom values specified.
addMarker(marker, minZoom, maxZoom)		Adds a single marker to the marker manager. Markers added using this method will display immediately; there is no need to call the refresh method. The marker will only be shown when the map is within the minZoom and maxZoom values specified.
refresh()		Forces the marker manager to update markers. You must call this method after adding markers using the addMarkers method.
getMarkerCount(zoom)	Number	Returns the number of markers that are potentially visible at the given zoom level.

GMarkerManager Events

Events	Arguments	Description
changed	bounds, markerCount	This event fires when markers managed by the manager are added to or removed from the map. The event handler function will receive two arguments. The first is the rectangle representing the bounds of the visible grid. The second argument provides the number of markers currently visible on the map.

class GMarkerManagerOptions

Use this class for the optional arguments provided to the GMarkerManager constructor.

GMarkerManagerOptions Properties

Property	Type	Description
borderPadding	Number	The extra padding outside the map's viewport monitored by the manager. Markers within this padded area are added to the map, even if they are not fully visible. The value is in pixels.
maxZoom	Number	The maximum zoom level monitored by the marker manager. It defaults to the highest possible zoom level if no value is provided.
trackMarkers	Boolean	The determinant of whether the marker manager tracks markers' movements. If you wish to move managed markers using the setPoint method, you should set this value to true. The default is false.

Functions

Along with the classes and objects, the API includes a few functions that don't require you to instantiate them as new objects.

Function	Returns	Description
GDownloadUrl(url, onload)		Retrieves the resource from the given URL and calls the onload function with the results of the resource as the first argument and the HTTP response status code as the second. The URL should be an absolute or relative path. This function is a simplified version of the GXmlHttp class; it is discussed in Chapter 3. It is subject to the same-origin restriction of cross-site scripting and, like the GXmlHttp class, it is executed asynchronously.
GBrowserIsCompatible()	Boolean	Returns true if the browser supports the API. Use this function to determine whether the browser is compatible with the Google Maps API.
GUnload()		Dismantles the map objects to free browser memory and avoid leaks and bugs. Call this function in the unload event handler for your web page to free up browser memory and help clean up browser leaks and bugs. Calling this function will disable all the map objects on the page.

Index

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z