Beginning Google Maps Applications with PHP and Ajax

From Novice to Professional

Michael Purvis
Jeffrey Sambells
and Cameron Turner

Apress®

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

ISBN-13 (pbk): 978-1-59059-707-1

ISBN-10 (pbk): 1-59059-707-9

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

Lead Editor: Jason Gilmore
Technical Reviewer: Terrill Dent
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Elizabeth Seymour
Copy Edit Manager: Nicole LeClerc
Copy Editor: Marilyn Smith
Assistant Production Director: Kari Brooks-Copony
Production Editor: Katie Stence
Compositor: Kinetic Publishing Services, LLC
Proofreader: Liz Welch
Indexer: Beth Palmer
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 http://www.apress.com in the Source Code section or at the official book site, http://googlemapsbook.com .

To Anne and Jim, that with GodÕs grace,
I might one day be so loving a parent.
ÑMichael Purvis

Dedicated to my loving wife, Stephanie, always by my side as my navigator in life.
May we never lose our way in this crazy world.
And also to my parents, Frank and Linda,
who taught me to always look beyond the horizon.
ÑJeffrey Sambells

I dedicate this book to my amazing wife, Tanya, and our son, Owen.
Tanya is the ultimate teammate and life partnerÑ
always willing to dive into an adventure or opportunity regardless of the size.
IÕd also like to thank my parents, Barry and Lorna, for supporting me
in all my ambitions and encouraging me to take risks and pursue dreams.
Without all of you, I would never have agreed to write my first book
about a moving-target topic such as Google Maps,
on a compressed timeline, with a newborn baby!
To everyone else who helped out in the last few months, thank you.
We couldnÕt have completed this book without your help and patience.
ÑCameron Turner

Contents at a Glance

Contents

Foreword

In the Beginning. . .

In the history of the Internet, 2005Ð2006 will be remembered as the year when online mapping finally came of age. Prior to 2005, MapQuest and other mapping services allowed you to look up directions, search for locations, and map businesses, but these searches were limited, usually to the companies the services had partnered with, so you couldnÕt search for any location. On February 8, 2005, Google changed all that. As it does with many of its services, Google quietly released the beta of Google Maps to its Labs incubator (http://labs.google.com) and let word-of-mouth marketing promote the new service.

By all accounts, Google Maps was an instant hit. It was the first free mapping service to provide satellite map views of any location on the earth, allowing anyone to look for familiar places. This started the ÒI can see my house from hereÓ trend, and set the blogosphere abuzz with links to Google Maps locations around the world.

Like other mapping services, Google Maps offered directions, city and town mapping, and local business searches. However, what the Google Maps engineers buried within its code was something that quickly set it apart from the rest. Although unannounced and possibly unplanned, they provided the means to manipulate the code of Google Maps to plot your own locations. Moreover, you could combine this base mapping technology with an external data source to instantly map many location-based points of information. And all of this could be done on privately owned domains, seemingly independent of Google itself.

At first, mapping ÒhackersÓ unlocked this functionality, just as video gamers hack into games by entering simple cheat codes. They created their own mapping services using Google Maps and other sources. One of the first these was Housingmaps.com, which combined the craigslist.org housing listings with a searchable Google Maps interface. Next came Adrian HolovatyÕs chicagocrime.org , which offered a compelling way to view crime data logged by the Chicago Police Department. These home-brewed mapping applications were dubbed Òhacks,Ó since Google had not sanctioned the use of its code in external domains on the Web.

The major change came in June 2005, when Google officially introduced the Google Maps API, which is the foundation for this book. By releasing this API, Google allowed programmers the opportunity to build an endless array of applications on top of Google Maps. Hundreds of API keys were registered immediately after the announcement, and many sites integrating Google Maps appeared within days. The map mashup was born.

The Birth of the Google Maps Mania Blog

The Google Maps labs beta site had been public for barely a month when I tried it for the first time. I was fascinated. While combing through the blogosphere looking for more information, I started to see a trend toward Google Maps hacks, how-to sites, Firefox extensions, and websites indexing specific satellite images. I thought that others could benefit from an aggregation of all of these ideas into one themed blog. Thus, my Google Maps Mania blog was born.

Google Maps Mania is more accurately described as a Òmeta-site,Ó as host Leo Laporte pointed out when I was a guest on his NPR G4techTV radio show in November 2005.

April 13, 2005, saw these as my first posts:

> Title: Google Maps Mania
>
> If youÕre like me you were absolutely floored when Google came out with the Google Maps service. Sure, itÕs just another mapping service. Until you realize itÕs full potential. The ability to toggle between regular street/road maps and a satellite view is unreal. IÕve started to see a lot of buzz around the blogging community about Google Maps so IÕve decided to help you keep up with the Google Maps related sites, blogs and tools that are cropping up. Stay tuned.
>
> Title: Google Sightseeing
>
> The first Google Maps related site of note is Google Sightseeing. This blog tracks interesting satellite shots as submitted by its visitors, then organizes them by interest area like buildings, natural landmarks and stadiums. ItÕs a pretty nifty site. Google Sightseeing even posted my suggestion of TorontoÕs Rogers Centre (Skydome) and the CN Tower!
>
> Title: Flickr Memory Maps
>
> HereÕs a Flickr group that took off fast. Memory Maps is a Flickr group that contains maps with captions describing memories they have of those areas or specific notes about different areas. Kind of cool.
>
> Title: Make your own multimedia Google map
>
> Google Blogoscoped tipped me off on this link. Seems Engadget has a page which gives some pretty good directions on how to create your own annotated multimedia Google map. There is some pretty serious direction here which includes inserting pictures and movies from the annotations. IÕd like to see an example of this.
>
> Title: My GMaps
>
> myGmaps enables you to create, save and host custom data files and display them with Google Maps. Create push-pin spots on any map of your choice. Mark your house, where an event will be held, or the route of a fun-run as a few examples. Then you can publish the map that youÕve created to your own website.

These postings represented an interesting cross-section of the ideas, concepts, and websites that I had come across in the two short months since Google Maps came to the Web. In the year between the start of Google Maps Mania and the release of the second-generation API (which this book is based on) in April 2006, I have made over 900 posts and attracted more than 6,000 daily readers to the blog, including the architects of the API itself. IÕve been Slashdotted, Dug (at Digg), and linked to from the *New York Times* site, as well as the sites of hundreds of other mainstream papers and magazines. In June 2006, Google arranged for my entire family to travel across the country so I could speak at the Google Geo Developer Day in advance of the Where 2.0 conference.

So many interesting mashups have been created using the Google Maps API that itÕs becoming impossible to keep up with all of them. I liken this to the early days of the Web when search directories began to manually catalog new web pages as they came online. The volume of new sites quickly became too huge to handle manually, and Google itself was born.

You can see why the Google Maps API offers the key for the next killer apps on the Web. It has been the missing link to take the Web to the next level.

This book will provide you the means to take part in this evolution of the Web. I hope to be posting about the interesting and unique map creations that you build after reading this book. Your creations will inspire others to do similar things, and together, we will continue to grow the Internet, one mapping application at a time. Let me know if you build something cool!

Mike Pegg
Google Maps Mania (http://www.gmapsmania.com)

About the Authors

MICHAEL PURVIS is a Mechatronics Engineering student at the University of Waterloo, in Ontario. He is a mostly self-taught programmer. Prior to discovering PHP, he was busy making a LEGO® Mindstorms kit play Connect 4. Currently, he maintains an active community site for classmates, built mostly from home-brewed extensions to PunBB and MediaWiki.

He has written about CSS for Position Is Everything, and occasionally participates in the css-discuss mailing list. He particularly enjoys those clever layouts that mix negative margins, relative positioning, and bizarre float tricks to create fiendish, cross-browser, flexible-width concoctions. These and other nontechnical topics are discussed on his weblog at uwmike.com.

Offline, he enjoys cooking, cycling, and social dancing. He has worked with We-Create, Inc. on a number of 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 along with 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 late 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 website at 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, and their little dog, Milo.

CAMERON TURNER has been programming computers since his first VIC 20 at age 7. He has been developing interactive websites since 1994. In 1999, he cofounded We-Create, Inc., which specializes in Internet software development. He is now the companyÕs 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.

Cam lives in CanadaÕs technology capital of Waterloo, Ontario, with his wife, Tanya, son Owen, and dog Katie. His hobbies include biking, hiking, water skiing, and painting. He maintains a personal blog at CamTurner.com, discussing nontechnical topics, thoughts, theories, and family life.

About the Technical Reviewer

TERRILL DENT is enrolled in Honors Mathematics at the University of Waterloo. His major interests center around Internet culture, twentieth century history, and economic theory. Terrill.ca is home to his weblog, and MapLet.ca is the front for his web application ventures, where he lets his acute attention to detail show through. Apart from work, he busies himself with fine arts, cycling, and an occasional novel.

Acknowledgments

The authors would like to thank Mike Pegg of Google Maps Mania for giving Apress our names when contacted about doing a book on Google Maps. This book would not have been possible without his encouragement, support, generosity, and friendship.

Thanks to Terrill for finding the errors of our bleary-eyed coding sessions and helping make this book what it is today.

Thanks to Jason, Elizabeth, Marilyn, Katie, Julie, and the rest of the team at Apress. We hope that working with us has been as much fun for you as working with you was for us.

PART 1

Your First Google Maps

CHAPTER 1

Introducing Google Maps

ItÕs hard to argue that Google Maps hasnÕt had a fundamental effect on the mapping world. While everyone else was still doing grainy static images, Google developers quietly developed the slickest interface since Gmail. Then they took terabytes of satellite imagery and road data, and just gave it all away for free.

WeÕre big fans of Google Maps and excited to get started here. WeÕve learned a lot about the Google Maps API since it was launched, and even more during the time spent writing and researching for this book. Over the course of the coming chapters, youÕre 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.

A lot of important web technologies and patterns have emerged in parallel with the Google Maps API. But whether you call it Ajax or Web 2.0 is less important than what it means: that the little guy is back.

You donÕt need an expensive development kit to use the Google Maps API. You donÕt need a computer science degree, or even a lot of experience. You just need a feel for whatÕs important data and an idea of what you can do to present it in a visually persuasive way.

We know youÕre eager to get started on a map project, but before we actually bust out the JavaScript, we wanted to show you two simple ways of creating ultra-quickie maps: using KML files and through the Wayfaring map site.

Using either of these approaches severely limits your ability to create a truly interactive experience, but no other method will give you results as quickly.

KML: Your First Map

The map weÕre working on here is actually Google Maps itself. In June 2006, Google announced that the official maps site would support the plotting of KML files. You can now simply plug a URL into the search box, and Google Maps will show whatever locations are contained in the file specified by the URL. We arenÕt going to go in depth on this, but weÕve made a quick example to show you how powerful the KML method is, even if it is simple.

Note KML stands for Keyhole Markup Language, which is a nod to both its XML structure and Google EarthÕs heritage as an application called Keyhole. Keyhole was acquired by Google late in 2004.

We created a file called toronto.kml and placed the contents of Listing 1-1 in it. The paragraph blurbs were 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://googlemapsbook.com/chapter1/kml/toronto.kml), we included two more Placemarkelements, which point to other well-known buildings in Toronto. To view this on Google Maps, paste that URL into the Google Maps search field. Alternatively, you can just visit this link:

http://maps.google.com/maps?f=q&hl=en&q=http://googlemapsbook.com/chapter1/kml/toronto.kml

You can see the results of this in Figure 1-1.

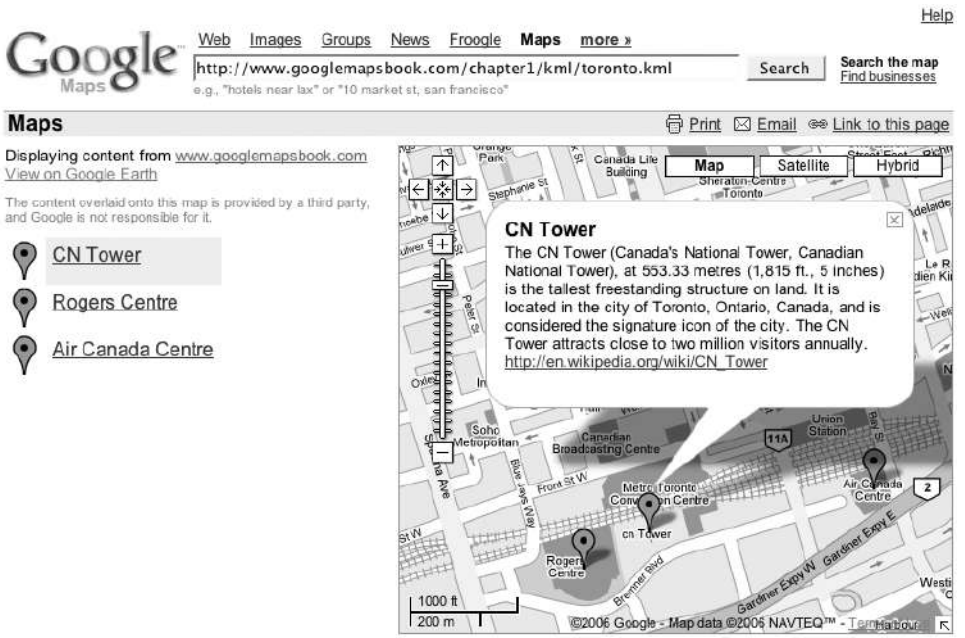


Figure 1-1.A custom KML data file being displayed at maps.google.com

Now, is that a quick result or what? Indeed, if all you need to do is show a bunch of locations, itÕs possible that a KML file will serve your purpose. If youÕre 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 wouldnÕt be any fun, would it? After all, as cool as the KML mapping is, it doesnÕt actually offer any interactivity to the user. In fact, most of the examples youÕll 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, youÕll start to see things that you can doonly when you harness the full power of the Google Maps API.

Before moving on, though, weÕll 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 weÕll demonstrate here, is Wayfaring.com (Figure 1-2). Wayfaring has received attention and praise for its classy design, community features (such as commenting and shared locations), and the fact that itÕs built using the popular Ruby on Rails framework.

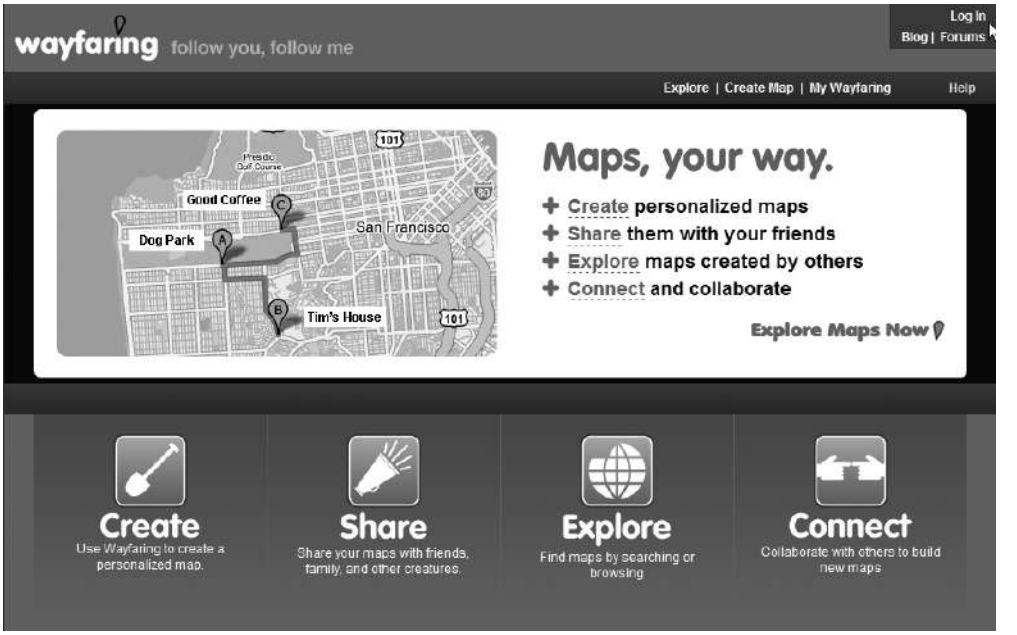


Figure 1-2.Wayfaring.com home page

Wayfaring is a mapping service that uses the Google Maps API and allows users to quickly create maps of anything they would like. For example, some people have made maps of their vacations; others have identified interesting aspects of their hometown or city. As an example, weÕll 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. Once youÕve created and activated your account, you can begin building your map. Click the Create link.

Adding the First Point

WeÕll start by adding the home airport for our imaginary journey. In our case, that would be Pearson International Airport in Toronto, Ontario, Canada, but you could use the one closest to you. Since Pearson is an international location (outside the United States), we need to drag and zoom the map view until we find it. If youÕre in the United States, you could use instead the nifty Jump To feature to search by text string. Figure 1-3 shows Pearson nicely centered and zoomed.

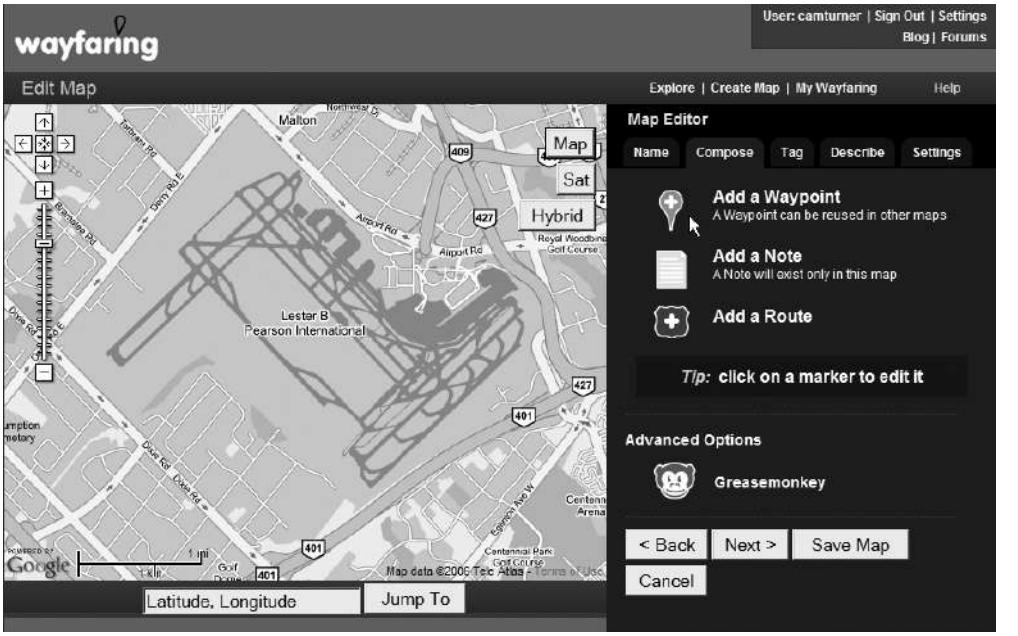


Figure 1-3. Lester B. Pearson International Airport, Toronto, Ontario

Once youÕve found your airport, you can click Next and name the map. After clicking ahead, you should be back at the main Map Editor screen.

Select Add a Waypoint from the list of options on the right. YouÕll be prompted to name the waypoint. WeÕll call ours ÒLester B Pearson International Airport.Ó However, as we type, we find that Wayfaring is suggesting this exact nam e. This means that someone else on some other map has already used this waypoint, and the system is giving us a choice of using their point or making one of our own. ItÕs 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 completeness, weÕll 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. WeÕll add the tags Òairport Toronto Ontario CanadaÓ 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 weÕre going to add to our map is a route. A route is a line made up of as many points as you would like. WeÕll 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 GoogleÕs 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 Òairplane tripÓ), and then click your airport. A small, white dot appears on the place you clicked. This is the first point on your line. Now zoom out, scroll over to California, and zoom in on San Francisco. The airport

weÕll be landing at 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 how far your flight was on the right side of the screen, underneath the route label. Wow, our flight seems to have been over 2000 miles! If you made a mistake and accidentally clicked a few extra times in the process of getting to San Francisco, you can use the Undo Last option. Otherwise, click Save.

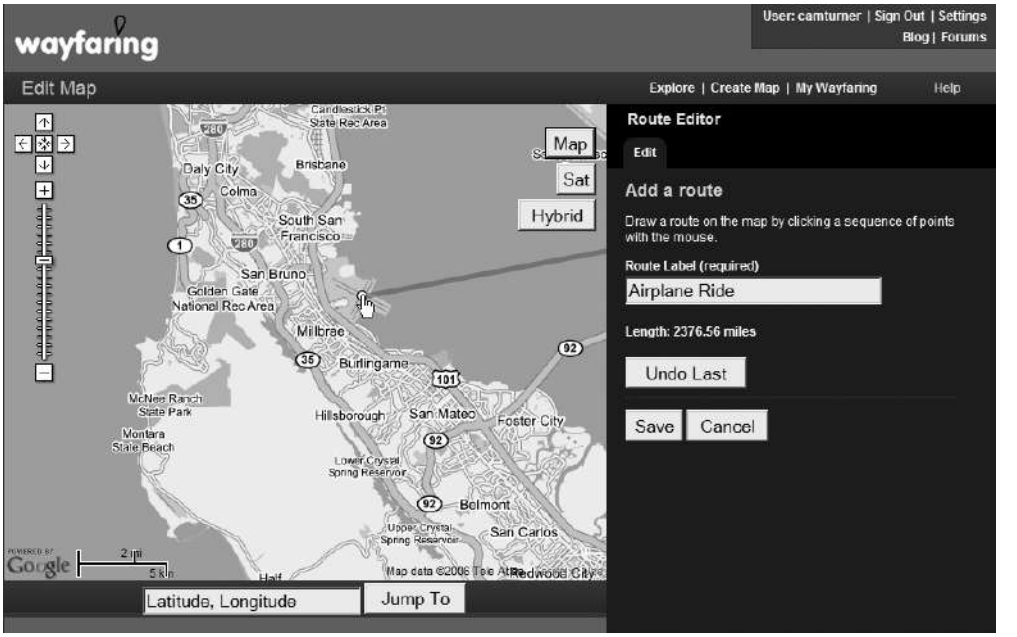


Figure 1-4.Our flight landing at San Francisco International Airport

Adding the Destination Point

Now that youÕre in San Francisco, letÕs figure out how to get to the Googleplex directly. Click Add a Waypoint. Our destination is Google, so weÕve called the new point ÒThe GoogleplexÓ and used the address box feature to jump directly to 1600 Amphitheatre Pky, Mountain View, CA 94043. Wayfaring is able to determine latitude and longitude from an address via a process called geocoding, which youÕll be seeing a lot more of in Chapter 4.

To confirm youÕre 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 close to Figure 1-5.

Figure 1-5. The Googleplex

Excellent! Save that waypoint.

Adding a Driving Route

Next, letÕs figure out how far of a drive we have ahead of us. Routes donÕt really have a starting and ending point in Wayfaring from a visual point of view, so we can start our route from the Googleplex and work our way backwards. Switch back into map (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 (US Highway 101) to the airport. By our tracing, we end up with about 23 miles of fun driving on this California highway, as shown in Figure 1-6.

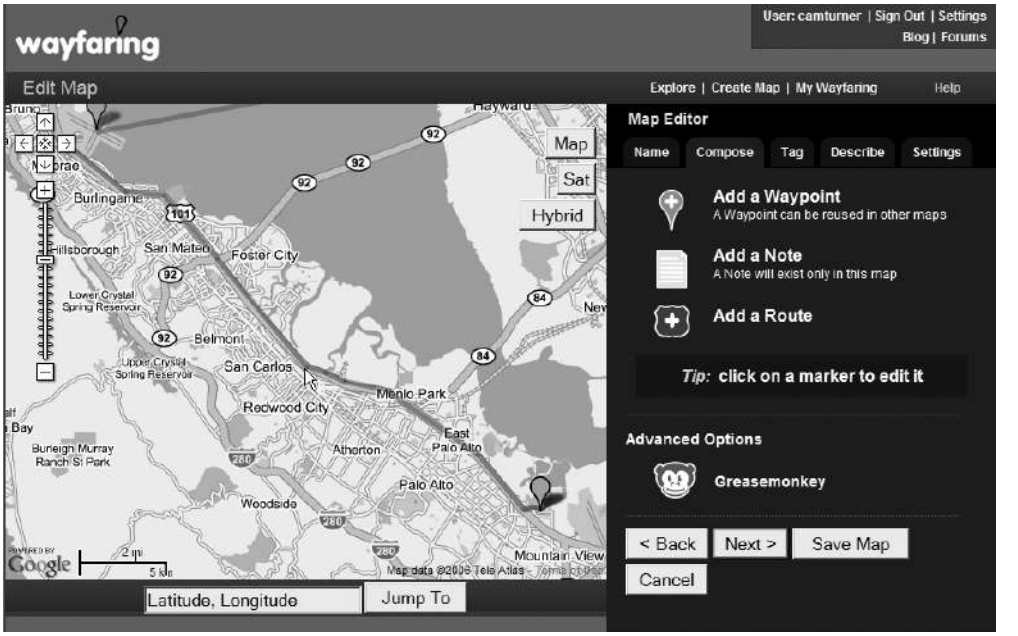


Figure 1-6. The drive down the Bayshore Freeway to the Googleplex

ThatÕs it. You can use the same principles to make an annotated map of your vacation or calculate how far youÕre going to travel, and best of all, itÕs a snap to share it. To see our map live, visit http://www.wayfaring.com/maps/show/17131 .

Of course, since this is a programming book, youÕre probably eager to dig into the code and make something really unique. Wayfaring may be nice, but the whole point of a mashup is to automate the process of getting a lot of data combined together.

Tip *Mashup* is a term that originates from DJs and other musicians who create new compositions by ÒmashingÓ together samples from existing songs. A classic example is *The Grey Album*, which joins the a capella versions of tracks from Jay-ZÕs *The Black Album* with unauthorized clips from *The White Album* by The Beatles. In the context of this book, *mashup* refers to the mashing of data from one source with maps from Google.

WhatÕs Next?

Now that these examples are out of the way, we hope youÕre eager to learn how to build your own mashups from the ground up. By the end of Part 1 of this book, youÕll have the skills to do everything youÕve just done on Wayfaring (except the route lines and distances, which are covered in Chapter 10) using JavaScript and XHTML. By the bookÕs conclusion, youÕll have learned most of the concepts needed to build your own Wayfaring clone!

So what exactly is to come? WeÕve divided the book 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. YouÕll 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, like 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 tooltip, 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.

We hope you enjoy!

CHAPTER 2

Getting Started

In this chapter, youÕll 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, youÕll also start to pick up a few essentials about that language along the way.

In this chapter, youÕll see how to do the following:

- Get off the ground with a basic map and a Google Maps API key.
- Separate the map applicationÕs JavaScript functions, data, and XHTML.
- Unload finished maps to help browsers free their memory.
- Create map markers and respond to clicks on them with an information pop-up.

The First Map

In this section, youÕll obtain a Google Maps API key, and then begin experimenting with it by retrieving GoogleÕs starter map.

Keying Up

Before you start a Google Maps web application, you need sign up for a Google Maps API key. To obtain your key, you must accept the Google Maps API Terms of Use, which stipulate, among other things, that you must not steal GoogleÕs imagery, obscure the Google logo, or hold Google responsible for its software. Additionally, youÕre prevented from creating maps that invade privacy or facilitate illegal activities.

Google issues as many keys as you need, but separate domains must apply for a separate key, as each one is valid for only a specific domain and subdirectory within that domain. For your first key, youÕll want to give Google the root directory of your domain or the space in which youÕre working. This will allow you to create your project in any subdirectory within 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 weÕre about to demonstrate in Listing 2-1.

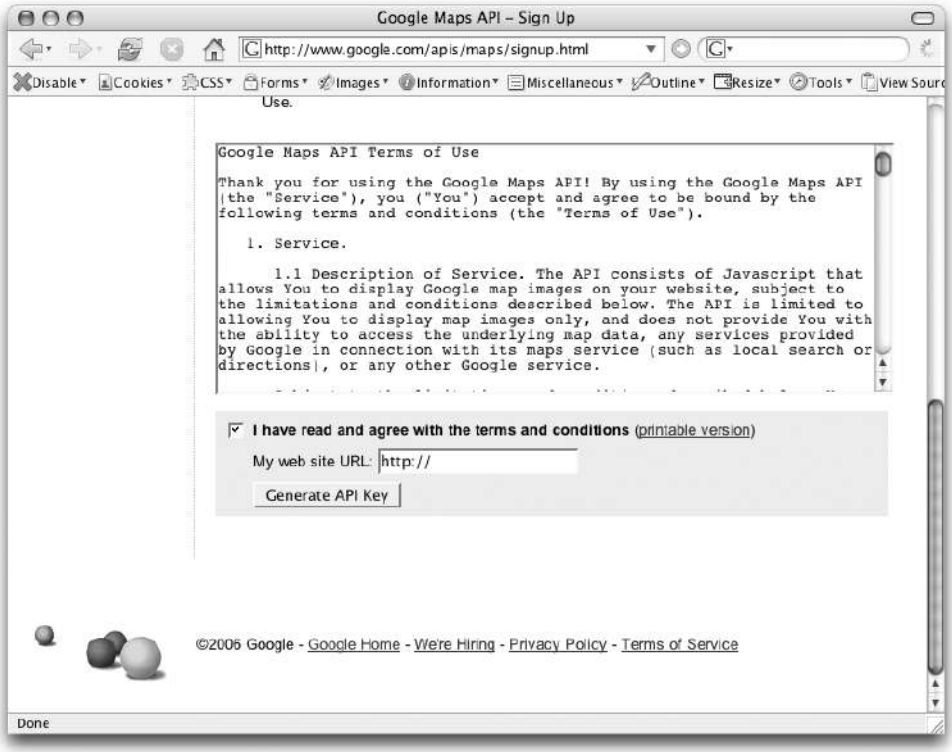


Figure 2-1. Signing up for an API key. Check the box, and then enter the URL of your webspace.

Note Why a key? Google has its reasons, which may or may not include seeing what projects are where which are the most popular, and which may be violating the terms of service. Google is not the only one th 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 also provide you with a very basic Òstarter mapÓ to help familiarize you with the fundamental concepts required to integrate a map into your website. WeÕll begin by dissecting and working with this starter code so you can gain a basic understanding of whatÕs happening.

If you start off using GoogleÕs sample, your key is already embedded in the JavaScript. Alternatively, you canÑas with all listingsÑgrab the source code from the bookÕs website at http://googlemapsbook.com and insert your own key by hand.

Either way, save the code to a file called index.php. Your key is that long string of characters following key=. (Our key, in the case of this bookÕs website, is ABQIAAAA33EjxkLYsh9SEveh_MphphQP1yR2bHJW2Brl_bW_l0KXsyt8cxTKO5Zz-UKoJ6IepTlZRxN8hfTRgw

Examining the Sample Map

Once you have the file in Listing 2-1 uploaded to your webspace, check it out in a browser. And ta-da, a map in action!

Listing 2-1. The Google Maps API Starter Code

```
<!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&amp;v=2&amp;key=ABQIAAAA
33EjxkLYsh9SEveh_MphphQP1yR2bHJW2Brl_bW_l0KXsyt8cxTKO5Zz-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>
```

In Listing 2-1, the container holding the map is a standard XHTML web page. A lot of the listing here is just boilerplateÑstandard 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 GoogleÕs server, and your key is passed as a parameter:

```
<script src="http://maps.google.com/maps?file=api&amp;v=2&amp;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 it 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 onto 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, as you can probably imagine, is a pretty important class, that we'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'll be introducing 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 (http://www.google.com/apis/maps/documentation/) directly online.

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've chosen the Golden Gate Bridge in San Francisco, California (Figure 2-2). It's a large landmark and is visible in the satellite imagery provided on Google Maps (http://maps.google.com). 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

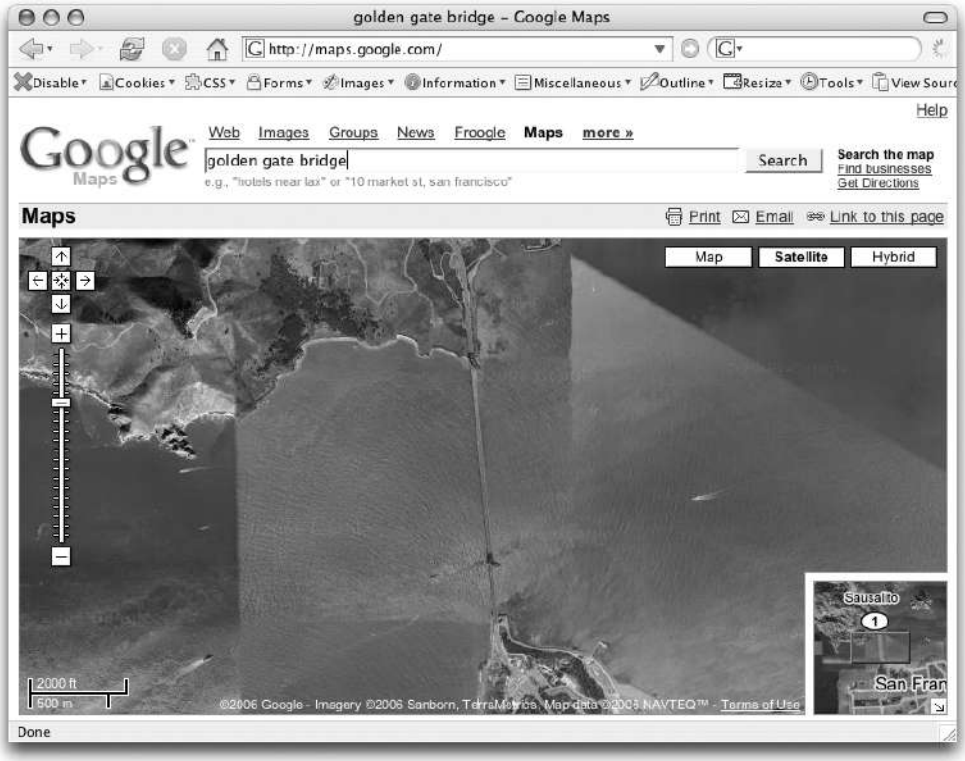


Figure 2-2.The Golden Gate Bridge satellite imagery from Google Maps

---

Caution If you use Google Maps to search for landmarks, the Link to This Page URL wonÕt immediately contain the latitude and longitude variable but instead have a parameter containing the search terms. To a 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.

ItÕs 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 youÕll 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 Ð90 (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 Ð180 (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, itÕs 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, itÕs 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 out not only increases loading speeds, as the browser can cache the JavaScript independently of the XHTML, but their removal also helps prevent the messy and unreadable code that results from mixing XHTML with other programming languages. Your eyes and your text editor will love you if they donÕt have to deal with mixed XHTML and JavaScript at the same time.

In this case, youÕll actually take it one step further and also separate the marker data file from the map functions file. This will allow you to easily convert the static data file to a dynamically generated file in later chapters, without the need to touch any of the processing JavaScript.

To accommodate these changes, weÕve separated the web applicationÕs JavaScript functions, data, and XHTML, putting them in separate files called `index.php` for the XHTML portion of the page, `map_functions.js` for the behavioral JavaScript code, and `map_data.php` for the data to plot on the map. Listing 2-2 shows the revised version of the `index.php` file.

Listing 2-2. Extrapolated index.php 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=
ABQIAAAAfAb2RNhzPaf0W1mtifapBRI9caN7296ZHDcvjSpGbL7PxwkwBS
ZidcfOwy4q2EZpjEJx3rc4Lt5Kg" type="text/javascript"></script>
    <script src="map_data.php" type="text/javascript"></script>
    <script src="map_functions.js" type="text/javascript"></script>
</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 are two extra `script` elements inside the `head`. Rather than referencing the external API, these reference localÑon the serverÑJavaScript files called `map_data.php` and `map_functions.js`. For now, youÕll leave the `map_data.php` file empty, but it will be used later in the chapter when we demonstrate how to map an existing list of markers. The important thing to note here is that it must be referenced first, before the `map_functions.js` file, so that the data is ÒavailableÓ to the code in the `map_functions.js` file. Listing 2-3 shows the revised `map_functions.js` file.

Listing 2-3. Extrapolated map_functions.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 wonÕt need to hunt down a `setCenter()` call thatÕs buried somewhere within the code.
- The initialization JavaScript has been moved out of the `body` of the XHTML and into the `map_functions.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 `index.php` file will remain exactly as it is in Listing 2-2, and you will need to add code only to the `map_functions.js` and `map_data.php` files to introduce the new features to your map.

Caution ItÕs important to see the difference between `init` and `init()`. When you add the parentheses after the function name, it means Òexecute it.Ó Without the parentheses, it means Ògive me a reference to When you assign a function to an event handler such as `document.onload`, you want to be very careful that you donÕt include the parentheses. Otherwise, all youÕve assigned to the handler is the functionÕs return value, probably `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 JavaScriptÕs 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 APIÕs objects, which helps keep browsers happy. The function is `GUnload()`, and to take advantage of it is a simple matter of hooking it onto the `body.onunload` event, as in Listing 2-4.

Listing 2-4. Calling GUnload() in map_functions.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;
```

ThereÕs no obvious reward for doing this, but itÕs 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, itÕs a quiet way to improve the experience for all your visitors.

Basic 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 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 mapÕs current center
- GSmallZoomControl, the two-button zoom control used in driving-direction pop-ups
- GMapTypeControl, which lets the visitor toggle between Map, Satellite, and Hybrid types

---

Tip If youÕre interested in making your own custom controls, you can do so by extending the GControl class and implementing its various functions. We may discuss this on the bgiken.com/gmaps/book.com blog, so be sure to check it out.

In all cases, itÕs a matter of instantiating the control object, and then adding it to the map with the GMap2 objectÕs addControl() method. For example, hereÕs how to add the small map control, which you can see as part of the next example in Listing 2-5:

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

You use an identical process to add all the controls: simply pass in a new instance of the controlÕs class.

Note What does *instantiating* mean? In object-oriented programming, a class is like a blueprint for a type of entity that can be created in memory. When you put new in front of a class name, JavaScript takes the blueprint and actually creates a usable copy (an *instance*) of the object. There's only one GLatLng class, but you can instantiate as many GLatLng *objects* as you need.

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 the 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

Does it seem like overkill? It's less scary than it sounds. An updated map_functions.js is presented in Listing 2-5, with the new lines 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, we 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.

Opening Info Windows

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. To know when to open the info window, however, 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

Detecting Marker Clicks

JavaScript is primarily an event-driven language. The `init()` function that you've been using since Listing 2-3 is hooked onto 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.

For example, if you had 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

Chapter 3 will discuss the info window in more detail. The method we'll demonstrate here is `openInfoWindowHtml()`. Although you can open info windows over arbitrary locations on the

map, here youÕll 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 youÕll need to combine this code with the addListener() function:

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

Finally, youÕll wrap up all the code for generating a pin, an event, and an info window into a single function, called addMarker(), 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.

A List of Points

In Listing 2-3, we introduced the variables centerLongitude and centerLatitude. Global variables like 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, similar to how a dictionary matches words to definitions. Compare these two lines:

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

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

The object, however, is slightly more interesting. 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, if 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'
    }
];
```

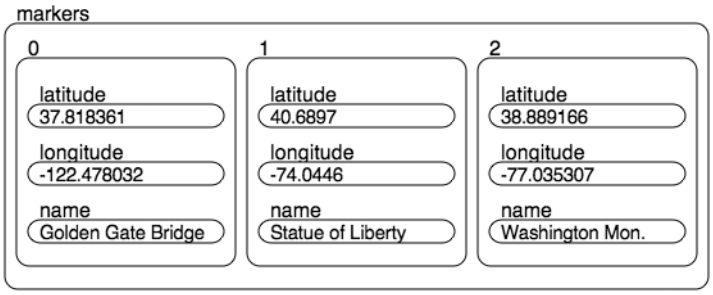


Figure 2-5.A series of objects stored inside an array

As youÕll see in the next section, JavaScript provides some terrific methods for working with data in this type of format.

Note In this book, youÕll see primarily MySQL used for storing data permanently. Some people however have proposed the exact format in Figure 2-5 as an alternative to XML, calling it JSON, for JavaScript Obje Notation. While there are some advantages, JSONÕs plethora of punctuation can be intimidating to a les technical person. You can find more information on JSON at http://json.org . WeÕll still be using a lot of JSON-like structures for communicating data from the server to the browser.

Iterating

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

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

However, JavaScript also provides a much classier way of setting this up. ItÕs called a `for in` loop. Watch for the difference:

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

Wow. It automatically gives you back every index that exists in an array or object, without needing to increment anything manually, or ever test boundaries. Clearly, youÕll want to use a `for in` loop to cycle over the array of points.

Until now, the `map_data.php` file has been empty and youÕve been dealing mainly with the `map_functions.js` file. To show a list of markers, you need to include the list, so this is where `map_data.php` comes in. For this chapter, youÕre not going to actually use any PHP, but the intention is that you can populate that file from database queries or some other data store. WeÕve named the file with the PHP extension so you can reuse the same base code in later chapters without the need to edit everything and start over. For now, pretend the PHP file is like any other normal JavaScript file and create your list of markers there. As an example, populate your `map_data.php` file with the structure from Listing 2-7.

To get that structure plotted, itÕs just a matter of wrapping the marker-creation code in a `for in` loop, as shown in Listing 2-8.

Listing 2-8. map_functions.js Modified to Use the Markers from map_data.php

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

function addMarker(longitude, latitude, 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(id in markers) {
            addMarker(markers[id].latitude, markers[id].longitude, markers[id].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 complete, youÕve made an incredible amount of progress! YouÕve 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 bandÕs upcoming gigs?

The possibilities are endless, but itÕs really just the beginning. In the next chapter, youÕll be expanding 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 geocodingÑturning street addresses into latitudes and longitudesÑ 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. Most, if not all, of the best Google Maps mashups 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 mashups in a number of ways. For instance, you might offer some additional detail for each marker using the info window bubbles introduced in Chapter 2, or use something more elaborate such as filtering the markers based on search criteria. Google Maps, Google's public mapping site (http://maps.google.com/) is a mashup of business addresses and a map to visually display where the businesses are located. It provides the required interactivity by allowing you to search for specific businesses, and listing 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.

In this chapter, we'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.

Going on a Treasure Hunt

To help you learn about some of the interactive features of the Google Maps API, youÕre going to go on a treasure hunt and create a map of all the treasures you find. The treasures in this case are geocaches, those 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 will discuss in the next chapter), or *geocaching* as the activity is commonly referred to, it is a global Òhide-and-seekÓ 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 their 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 an item in the cache for something of their own.

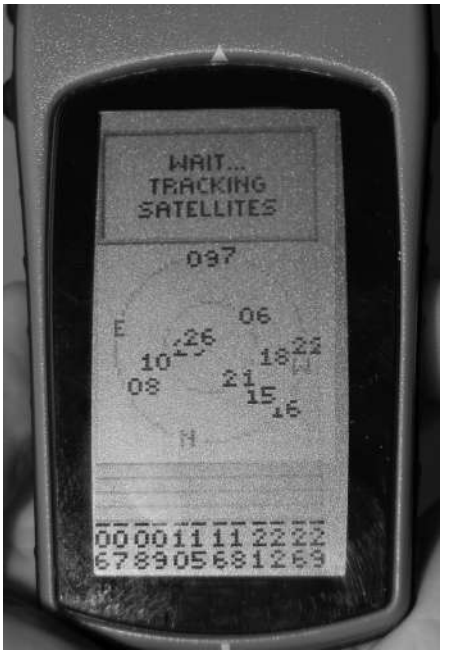


Figure 3-1.A common handheld GPS device used by geocachers to locate hidden geocaches

---

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

As you create your interactive geocache treasure map, youÕll 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.
- Save the latitude, longitude, and additional information in the form to your server using the GXmlHttp Asynchronous JavaScript and XML (Ajax) object on the client side and PHP on the server.
- Retrieve the existing markers and their additional information from the server using Ajax and PHP.
- Re-create the map upon loading by inserting new markers from a server-side list, each with an info window to display its information.

For this chapter, weÕre not going to discuss any CSS styling of the map and its contents; weÕll leave all that up to you.

Creating the Map and Marking Points

YouÕll begin the map for this chapter from the same set of files introduced in Chapter 2, which include the following:

- index.php to hold the XHTML of the page
- map_functions.js to hold the JavaScript functionality
- map_data.php to create a JavaScript array and objects representing each location on the map

Additionally, youÕll create a file called storeMarker.php to save information back to the server and another file called retrieveMarkers.php to retrieve XML using Ajax, but weÕll get to those later.

Starting the Map

To start, copy the index.php file from Listing 2-2 and the map_functions.js file from Listing 2-3 into a new directory for this chapter. Also, create an empty map_data.php file and empty store Marker.php and retrieveMarkers.php files.

While building the map for this chapter and other projects, youÕll be adding auxiliary functions to the map_functions.js file. 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 map_functions.js file. It will also ensure youÕre targeting the same map object. Also, you may want to add some of the control objects introduced in Chapter 2, such as GMapTypeControl Listing 3-1 highlights the map variable and additional controls.

Listing 3-1. Highlights for map_functions.js

```
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 GMap2TypeControl());
        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). For this example, the starting GLatLngis not important, so feel free to change it to some other location if you wish.

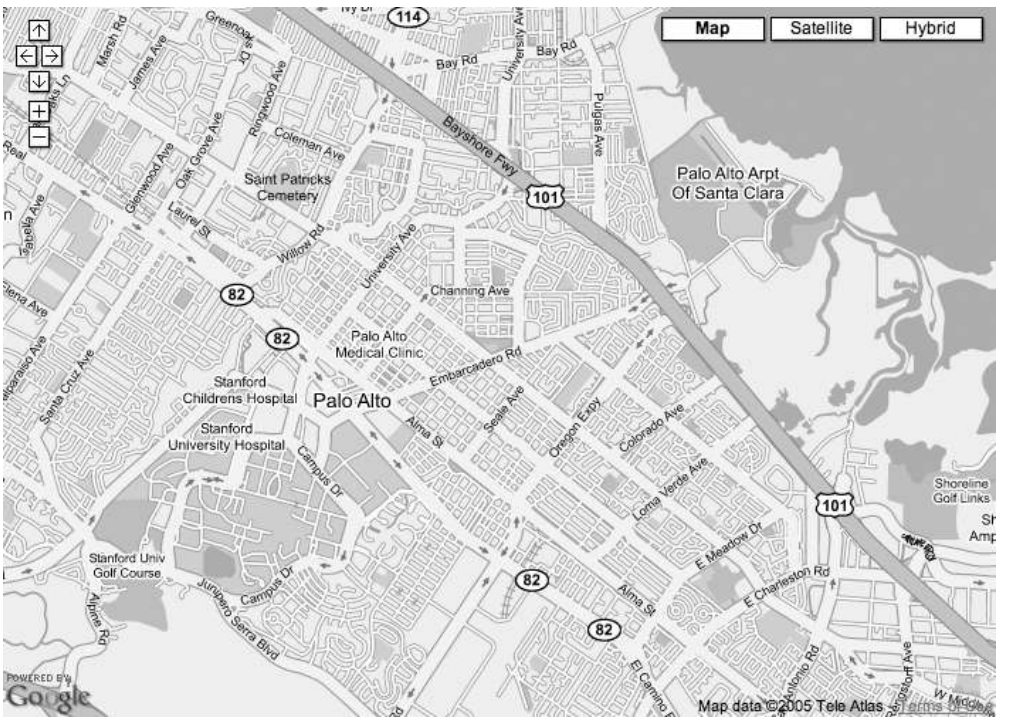


Figure 3-2. Starting map with controls centered on Palo Alto, California

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 the map, not moving it or dragging it around.

Tip If you refer to the Google Maps API documentation in Appendix B, 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 a `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.

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

Given the source `map` and the event `click`, this example will trigger the function to run any code you wish to implement.

Take a look at the modification to the `init()` function in Listing 3-2 to see how easy it is to add this event listener to your existing code and use it to create markers the same way you did in Chapter 2. The difference is that in Chapter 2, you used `new GLatLng()` to create the latitude and longitude location for the markers, whereas here, instead of creating a `new GLatLng`, you can 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 `overlay` variable is the overlay where the clicked location resides if you clicked on a marker or another overlay object.

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 GMap2TypeControl());
        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);
            });
        }
}
```

Ta-da! Now, with a slight code addition and one simple click, anyone worldwide could visit your map page and 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 GXmlHttpobject or the GDownloadUrlobject, which weÕll discuss in the ÒUsing GoogleÕs Ajax ObjectÓ section later in this chapter.

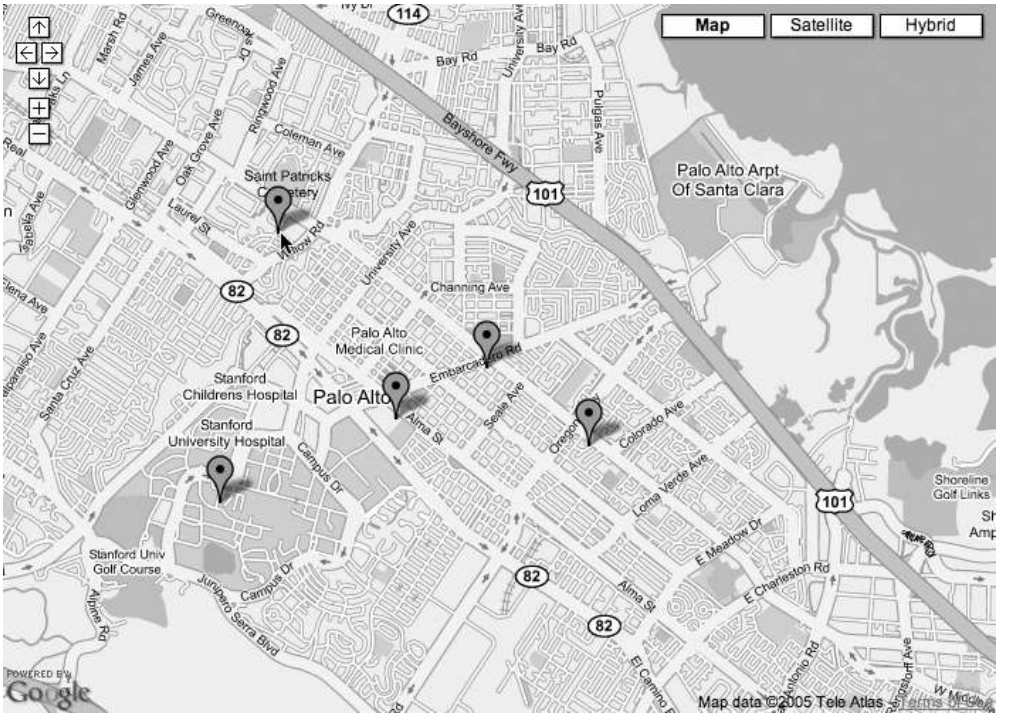


Figure 3-3.New markers created by clicking on the map

RETRIEVING THE LATITUDE AND LONGITUDE FROM A MAP CLICK

When you click on a Google map, the `latlng` variable passed into the event listenerÕs 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 canÕt 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 was found at each location. To provide this extra information, letÕs 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, letÕs put the form in an info window indicating where the visitor clicked.

As introduced in Chapter 2, the info window is the cartoon-like bubble that often appears when you click map markers (Figure 3-4). It is used by Google Maps to allow you to enter the To Here or From Here information for 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.

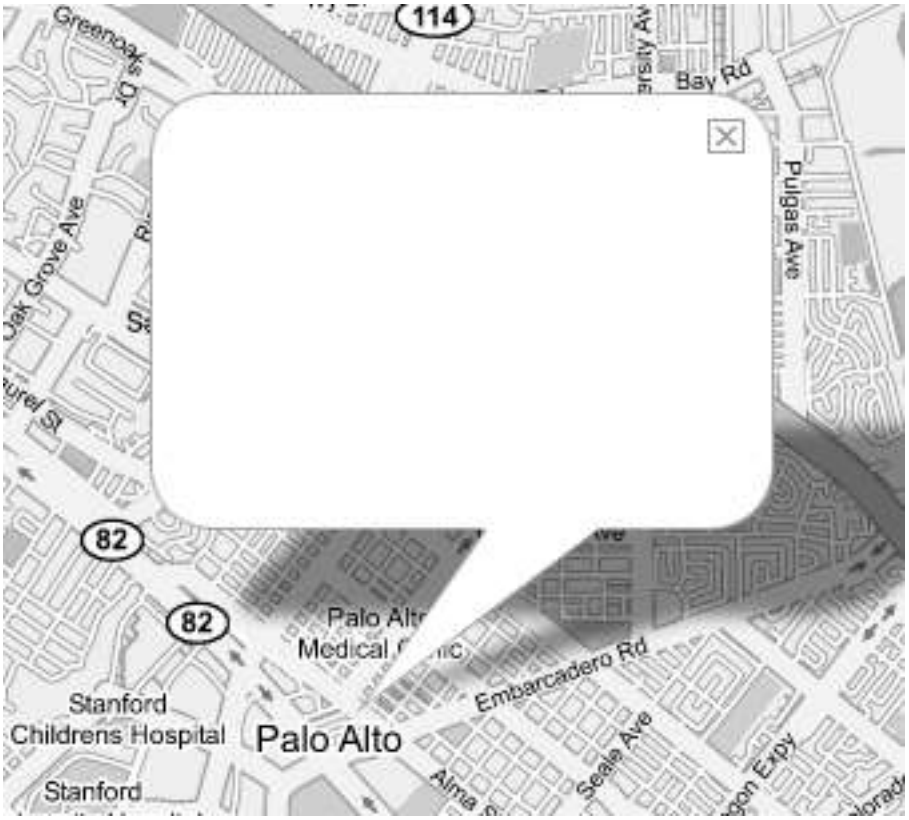


Figure 3-4. An empty info window

YouÕre 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 theyÕve found.

Note When we introduce the `GXmlHttp` object in the ÒUsing GoogleÕs Ajax ObjectÓ section later in this chapter, weÕll explain how to save the content of the info window to your server.

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, youÕll 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 Listing 3-2 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, youÕll see an info window pop up with its base pointing at the position you just clicked with the content ÒYou clicked here!Ó (Figure 3-5).

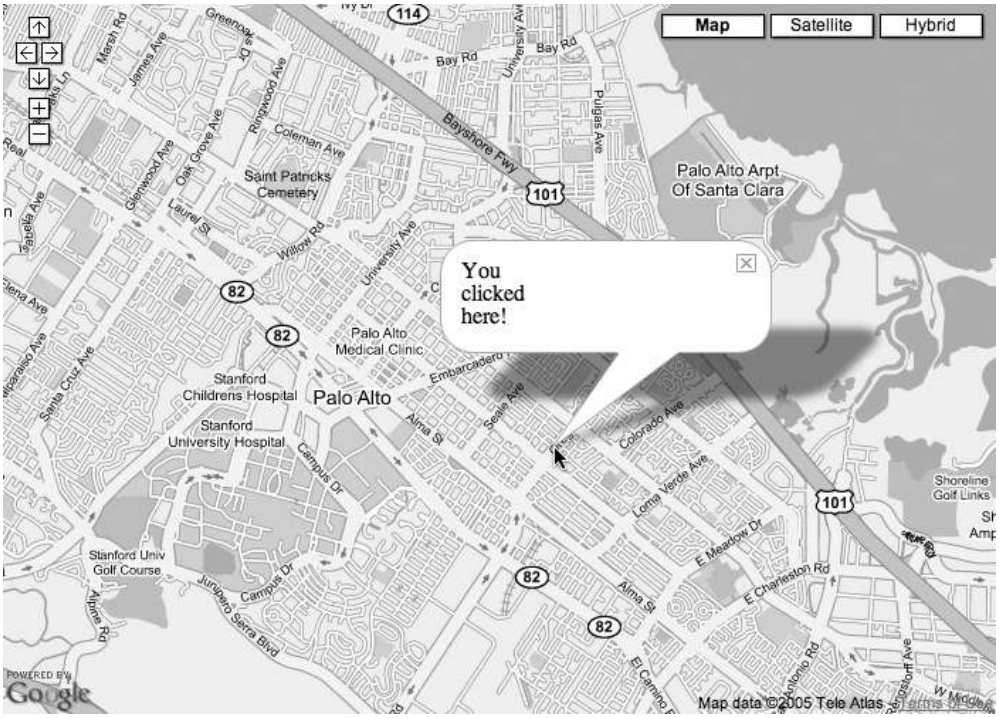


Figure 3-5. An info window created when clicking the map

Embedding a Form into the Info Window

When geocachers want to create a new marker, youÕll first prompt them to enter some information about their treasure. YouÕll want to know the geocacheÕs 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, youÕll 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 Document Object Model (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="storeMarker(); return false;">
    <fieldset style="width:150px;">
        <legend>New Marker</legend>
        <label for="found">Found</label>
        <input type="text" id="found" style="width:100%;"/>
        <label for="left">Left</label>
        <input type="text" id="left" style="width:100%;"/>
        <input type="submit" value="Save"/>
        <input type="hidden" id="longitude"/>
        <input type="hidden" id="latitude"/>
    </fieldset>
</form>
```

Note You may notice the form in Listing 3-3 has an `onsubmit` event attribute that calls a `storeMarker()` JavaScript function. The `storeMarker()` 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 `storeMarker()` 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() {storeMarker(); 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" style="width:100%;"/>'
                + '<label for="left">Left</label>'
                + '<input type="text" id="left" style="width:100%;"/>'
                + '<input type="submit" value="Save"/>'
                + '<input type="hidden" id="longitude" value="' +          lng  + '"/>'
                + '<input type="hidden" id="latitude" value="' +           lat  + '"/>'
                + '</fieldset>';

        map.openInfoWindow (latlng,inputForm);
});
```

Caution When creating the DOM form element, you need to use the setAttribute() method to define things like name, action, target, and method, but once you venture beyond these basic four, you may begin to notice inconsistencies. For example, 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 website 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 (Figure 3-6).

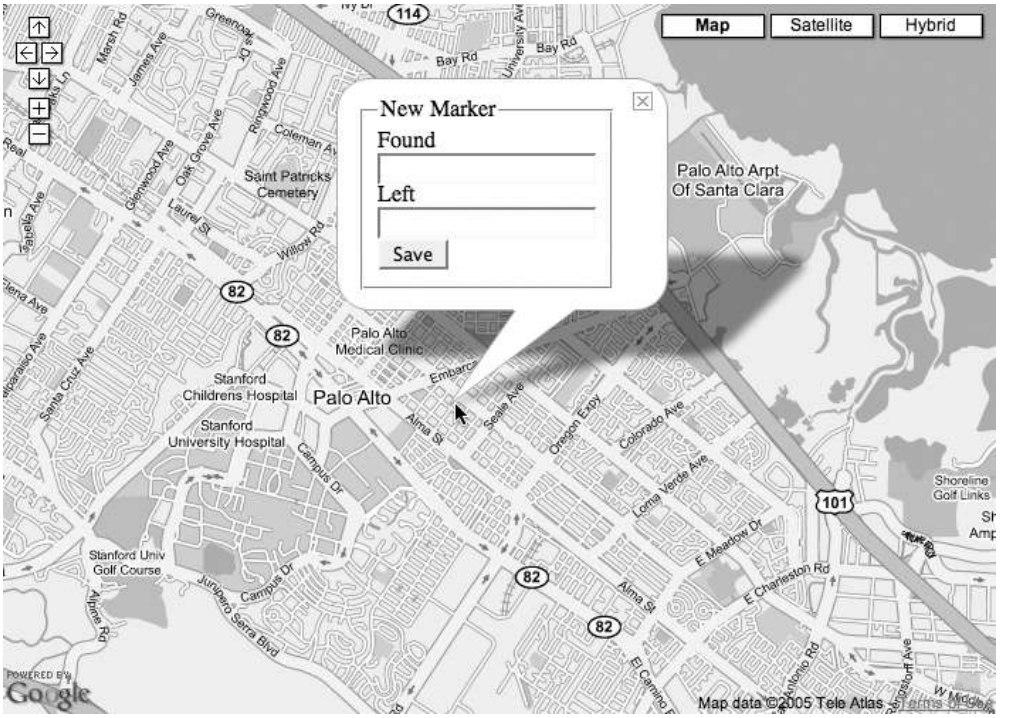


Figure 3-6. The info window with an embedded form

In Listing 3-4, the latitude and longitude elements of the form have been pre-populated 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 re-create the marker in the exact position when you retrieve the data from the server. Also, once the information has been 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, youÕll also 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 (Figure 3-7), and then clicking elsewhere on the map without clicking the `Save` button. YouÕll 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

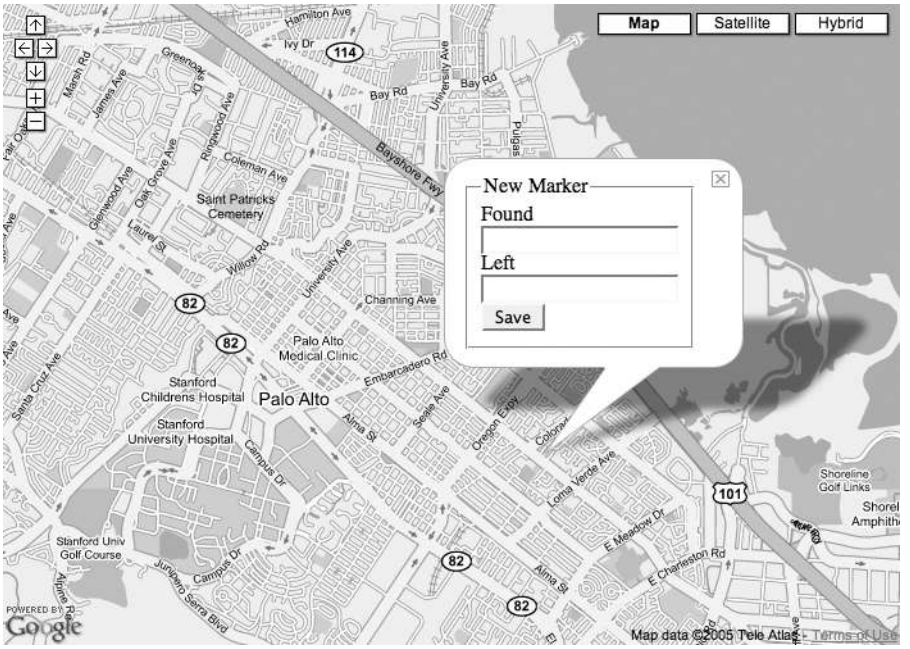


Figure 3-8.New info window that has lost the previously supplied information

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 PHP GoogleMapsUtility class in Listing 7-11, which has a variety of useful methods to help you create your tiles. The tile script for the custom tile method (shown later in Listing 7-13) uses the methods of the GoogleMapsUtility class 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'll discuss further in Chapter 9, when we talk about types of map projections.

Listing 7-11. The GoogleMapUtility Class Methods for Tile Construction

```
<?php

class GoogleMapUtility {
    //The Google Maps all use tiles 256x256
    const TILE_SIZE = 256;
    /**
     * Convert from a pixel location to a geographical location.
     **/
    public static function fromXYToLatLng($point,$zoom) {
        $mapWidth = (1 << ($zoom)) * GoogleMapUtility::TILE_SIZE;

        return new Point(
            (int)($normalised->x * $mapWidth),
            (int)($normalised->y * $mapWidth)
        );
    }

    /**
     * Calculate the pixel offset within a specific tile
     * for the given latitude and longitude.
     **/
    public static function getPixelOffsetInTile($lat,$lng,$zoom) {
        $pixelCoords = GoogleMapUtility::toZoomedPixelCoords(
            $lat, $lng, $zoom
        );
        return new Point(
            $pixelCoords->x % GoogleMapUtility::TILE_SIZE,
            $pixelCoords->y % GoogleMapUtility::TILE_SIZE
        );
    }

    /**
     * Determine the geographical bounding box for the specified tile index
     * and zoom level.
     **/
    public static function getTileRect($x,$y,$zoom) {
        $tilesAtThisZoom = 1 << $zoom;
```

```
        $lngWidth = 360.0 / $tilesAtThisZoom;
        $lng = -180 + ($x * $lngWidth);

        $latHeightMerc = 1.0 / $tilesAtThisZoom;
        $topLatMerc = $y * $latHeightMerc;
        $bottomLatMerc = $topLatMerc + $latHeightMerc;

        $bottomLat = (180 / M_PI) * ((2 * atan(exp(M_PI *
            (1 - (2 * $bottomLatMerc))))) - (M_PI / 2));
        $topLat = (180 / M_PI) * ((2 * atan(exp(M_PI *
            (1 - (2 * $topLatMerc))))) - (M_PI / 2));

        $latHeight = $topLat - $bottomLat;

        return new Boundary($lng, $bottomLat, $lngWidth, $latHeight);
    }

    /**
     * Convert from latitude and longitude to Mercator coordinates.
     **/
    public static function toMercatorCoords($lat, $lng) {
        if ($lng > 180) {
            $lng -= 360;
        }

        $lng /= 360;
        $lat = asinh(tan(deg2rad($lat)))/M_PI/2;
        return new Point($lng, $lat);
    }

    /**
     * Normalize the Mercator coordinates.
     **/
    public static function toNormalisedMercatorCoords($point) {
        $point->x += 0.5;
        $point->y = abs($point->y-0.5);
        return $point;
    }

    /**
     * Calculate the pixel location of a  latitude and longitude point
     * on the overall map at a  specified zoom level.
     **/
    public static function toZoomedPixelCoords($lat, $lng, $zoom) {
        $normalised = GoogleMapUtility::toNormalisedMercatorCoords(
            GoogleMapUtility::toMercatorCoords($lat, $lng)
        );
```

```
            $scale = (1 << ($zoom)) * GoogleMapUtility::TILE_SIZE;
            return new Point(
                  (int) ($normalised->x * $scale),
                  (int)($normalised->y * $scale)
            );
      }
}

/**
* Object to represent a  coordinate point (x,y).
**/
class Point {
      public $x,$y;
      function __construct($x,$y) {
            $this->x = $x;
            $this->y = $y;
      }

      function __toString() {
            return "({$this->x},{$this->y})";
      }
}

/**
* Object to represent a  boundary point (x,y) and (width,height)
**/
class Boundary {
      public $x,$y,$width,$height;
      function __construct($x,$y,$width,$height) {
            $this->x = $x;
            $this->y = $y;
            $this->width = $width;
            $this->height = $height;
      }
      function __toString() {
            return "({$this->x},{$this->y},{$this->width},{$this->height})";
      }
}

?>
```

Using the GoogleMapsUtility 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 youÕll see soon), each tile request:

```
var tileURL = "server.php?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 GoogleMapsUtility::getTileRect method, as demonstrated in the server-side PHP script for the custom tiles in Listing 7-13 (also coming up soon). 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.

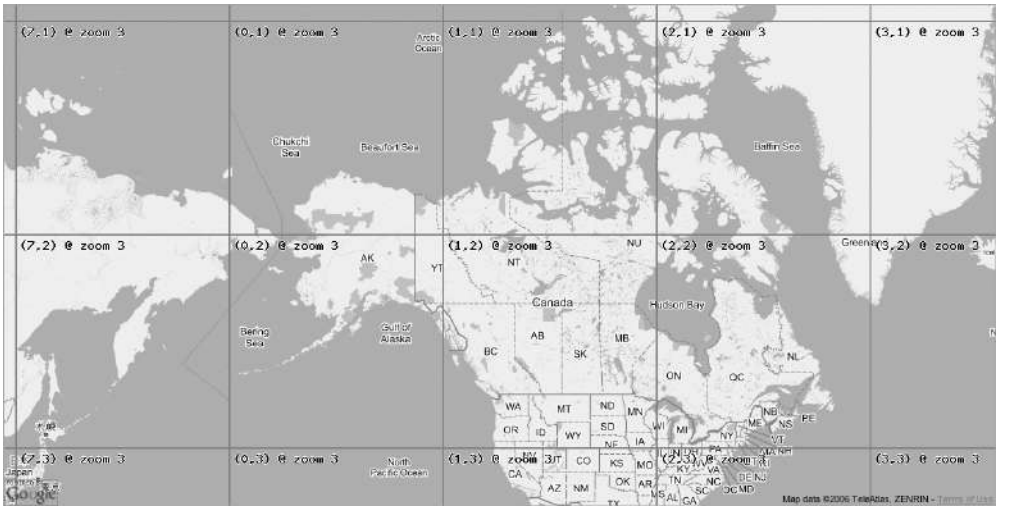


Figure 7-8. Google tile numbering scheme

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://googlemapsbook.com/chapter7/ServerCustomTiles/). In Chapter 9, youÕll 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, weÕve drawn a colored circle outlined in white with each color representing the height of the tower, as shown in Figure 7-9.

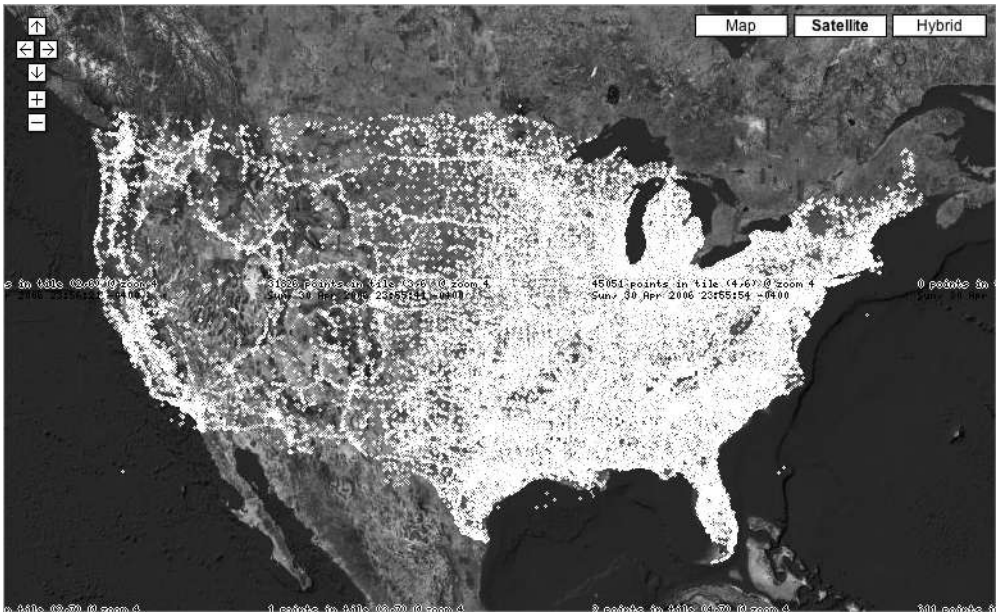


Figure 7-9. The finalized custom tile map in satellite mode

For testing purposes, each tile is also labeled with the date/time tile number and the number of points in that tile. If you look at the online example, youÕll notice that the tiles render very quickly. 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 15 seconds to render, as there were almost 50,000 points per tiles in the United States. If the data on your map is continually changing, you may want to consider running a script to create all the tiles before publishing your map to the Web so your first visitors donÕt experience a lag when the tiles are first created.

Listing 7-12. Client-Side JavaScript for the Custom Tile Method

```
var map;
var centerLatitude = 49.224773;
var centerLongitude = -122.991943;
var startZoom = 6;

//create the tile layer object
var detailLayer = new GTileLayer(new GCopyrightCollection(''));

//method to retrieve the URL of the tile
detailLayer.getTileUrl = function(tile, zoom){
    //pass the x and y position as well as the zoom
    var tileURL = "server.php?x="+tile.x+"&y="+tile.y+"&zoom="+zoom;
    return tileURL;
};
```

```
detailLayer.isPng = function() {
    //the example uses GIFs
    return false;
}

//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. Server-Side PHP for the Custom Tile Method

```
<?php

//include the helper calculations
require('GoogleMapUtility.php');


//this script may require additional memory and time
set_time_limit(0);
ini_set('memory_limit',8388608*10);

//create an array of the size for each marker at each zoom level
$markerSizes = array(1,1,1,1,2,2,3,3,4,4,5,5,6,6,7,7,8,8,9,9,10,10,11,11,12,12);

//get the lat/lng bounds of this tile from the utility function
//return a  bounds object with width,height,x,y
$rect = GoogleMapUtility::getTileRect(
    (int)$_GET['x'],
    (int)$_GET['y'],
    (int)$_GET['zoom']
);
```

```
//create a  unique file name for this tile
$file = 'tiles/c'.md5(
      serialize($markerSizes).
      serialize($rect).'|'.
      $_GET['x'].'|'.
      $_GET['y'].'|'.
      $_GET['zoom']).
      '.gif';

//check if the file already exists
if(!file_exists($file)) {

      //create a  new image
      $im = imagecreate(GoogleMapUtility::TILE_SIZE,GoogleMapUtility::TILE_SIZE);
      $trans = imagecolorallocate($im,0,0,255);
      imagefill($im,0,0,$trans);
      imagecolortransparent($im, $trans);
      $black = imagecolorallocate($im,0,0,0);
      $white = imagecolorallocate($im,255,255,255);

      //set up some colors for the markers.
      //each marker will have a  color based on the height of the tower
      $darkRed = imagecolorallocate($im,150,0,0);
      $red = imagecolorallocate($im,250,0,0);
      $darkGreen = imagecolorallocate($im,0,150,0);
      $green = imagecolorallocate($im,0,250,0);
      $darkBlue = imagecolorallocate($im,0,0,150);
      $blue = imagecolorallocate($im,0,0,250);
      $orange = imagecolorallocate($im,250,150,0);

      //init some vars
      $extend = 0;
      $z = (int)$_GET['zoom'];
      $swlat=$rect->y + $extend;
      $swlng=$rect->x+ $extend;
      $nelat=$swlat+$rect->height + $extend;
      $nelng=$swlng+$rect->width + $extend;

      //connect to the database
      require($_SERVER['DOCUMENT_ROOT'] . '/db_credentials.php');
      $conn = mysql_connect("localhost", $db_name, $db_pass);
      mysql_select_db("googlemapsbook", $conn);

       /*
       * Retrieve the points within the boundary of the map.
       * For the FCC data, all the points are within the US so we
       * don't need to worry about the meridian problem.
       */
```

```
$result = mysql_query(
    "SELECT
        longitude as lng,latitude as lat,struc_height,struc_elevation
    FROM
        fcc_towers
    WHERE
        (longitude > $swlng AND longitude < $nelng)
        AND (latitude <= $nelat AND latitude >= $swlat)
    ORDER BY
        lat"
    , $conn);

//get the number of points in this tile
$count = mysql_num_rows($result);


$filled=array();
$row = mysql_fetch_assoc($result);
while($row)
{
    //get the x,y coordinate of the marker in the tile
    $point = GoogleMapUtility::getPixelOffsetInTile($row['lat'],$row['lng'],$z);

    //check if the marker was already drawn there
    if($filled["{$point->x},{$point->y}"]<2) {

        //pick a  color based on the structure's height
        if($row['struc_height']<=20) $c = $darkRed;
        elseif($row['struc_height']<=40) $c = $red;
        elseif($row['struc_height']<=80) $c = $darkGreen;
        elseif($row['struc_height']<=120) $c = $green;
        elseif($row['struc_height']<=200) $c = $darkBlue;
        else $c = $blue;

        //if there is aready a  point there, make it orange
        if($filled["{$point->x},{$point->y}"]==1) $c=$orange;

        //get the size
        $size = $markerSizes[$z];

        //draw the marker
        if($z<2) imagesetpixel($im, $point->x, $point->y, $c );
        elseif($z<12) {
            imagefilledellipse($im, $point->x, $point->y, $size, $size, $c );
            imageellipse($im, $point->x, $point->y, $size, $size, $white );
        } else {
            imageellipse($im, $point->x, $point->y, $size-1, $size-1, $c );
            imageellipse($im, $point->x, $point->y, $size-2, $size-2, $c );
```

```
                imageellipse($im, $point->x, $point->y, $size+1, $size+1, $black );
                imageellipse($im, $point->x, $point->y, $size, $size, $white );
            }

            //record that we drew the marker
            $filled["{$point->x},{$point->y}"]++;
        }

        $row = mysql_fetch_assoc($result);
    }

    //write some info about the tile to the image for testing
    imagestring($im,1,-1,0,
        "$count points in tile ({$_GET['x']},{$_GET['y']}) @ zoom $z ",$white);
    imagestring($im,1,0,1,
        "$count points in tile ({$_GET['x']},{$_GET['y']}) @ zoom $z ",$white);
    imagestring($im,1,0,-1,
        "$count points in tile ({$_GET['x']},{$_GET['y']}) @ zoom $z ",$white);
    imagestring($im,1,1,0,
        "$count points in tile ({$_GET['x']},{$_GET['y']}) @ zoom $z ",$white);
    imagestring($im,1,0,0,
        "$count points in tile ({$_GET['x']},{$_GET['y']}) @ zoom $z ",$black);
    imagestring($im,1,0,9,date('r'),$black);

    //output the new image to the file system and then send it to the browser
    header('content-type:image/gif;');
    imagegif($im,$file);
    echo file_get_contents($file);

} else {

    //output the existing image to the browser
    header('content-type:image/gif;');
    echo file_get_contents($file);

}

?>
```

Tip Another benefit of using the tile layer is that it bypasses the cross-domain scripting restrictions on the browser. Each tile is actually an image and nothing more than GET parameters specify which tile the browser is requesting, and the browser can load any image from any site, as it is not considered maliciousÑitÕs just an image.

BUT WHAT ABOUT INFO WINDOWS?

Using tiles to display your ÒmarkersÓ is relatively easy, and you can simulate most of the features of the GMarker object with the exception of info windows. You canÕt 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(
        "your_server_side_script.php?"
        + "lat=" + point.lat()
        + "&lng=" + point.lng()
        + "&z=" + map.getZoom(),
        function(data, responseCode) {
            map.openInfoWindow(point,document.createTextNode(data));
        });
});
```

The trick is figuring out what was actually clicked. When your users click your map, youÕll need to send the locationÕs latitude and longitude back to the server and have it determine what information is relative to that point. If something was clicked, you can then send the appropriate information back across the Ajax request and create an info window directly on the map. From the clientÕs 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 just a little too big for the mapÑsomewhere between 100 to 300 pointsÑ you donÕt necessarily need to make new requests to retrieve your information. You can achieve good results using solutions similar to those weÕve outlined for the server side, but store the data set in the browserÕs 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 methods weÕll 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, weÕll recommend a couple other optimizations to improve your usersÕ experience.

Client-Side Boundary Method

With the server-side boundary method, you used the server to check if a point was inside the boundary of the map. Doing so on the server side required that you write the calculation manually into your script. Using the Google Maps API provides a much simpler solution, as 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()` methods returns `true` if the supplied point is within the geographical coordinates defined by the rectangular boundary.

Listing 7-14 (http://googlemapsbook.com/chapter7/ClientBounds/) shows the working example of the boundary method implemented in JavaScript.

Listing 7-14. JavaScript for the Client-Side Boundary Method

```
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() {
    map.clearOverlays();
    var mapBounds = map.getBounds();

    //loop through each of the points from the global points object
    for (k in points) {
        var latlng = new GLatLng(points[k].lat,points[k].lng);
        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 JSONobject resembles the object discussed earlier in the chapter:

```
var points = {
    p1:{lat:-53,lng:-74},
    p2:{lat:-51.4,lng:59.51},
    p3:{lat:-45.2,lng:-168.43},
    p4:{lat:-41.19,lng:-174.46},
    p5:{lat:-36.3,lng:60},
    p6:{lat:-35.15,lng:-149.08},
    p7:{lat:-34.5,lng:56.11},

    ... etc ...

    p300:{lat:-33.24,lng:70.4},
}
```

This object was loaded into the browser using another script tag, in the same way you loaded the data into the map in Chapter 2. Now, rather than creating a new request to the server, the points object contains all the points, so you only need to loop through points and determine if the current point is within the current boundary. Listing 7-14 uses the current boundary of the map from map.getBounds().

Client-Side Closest to a Common Point Method

As with the boundary method, the client-side closest to a common point method is similar to the server-side closest to common point method, but you can use the Google Maps API to accomplish the same goal on the client side if you don't have too many points. 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 GLatLngclass 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 7-15 (http://googlemapsbook.com/chapter7/ClientClosest/), you can see the client-side JavaScript is very similar to the server-side PHP in Listing 7-5. The main difference (besides not sending a request to the server) is the use of point.distanceFrom() rather than

the surfaceDistance() PHP function. Also for the example, the boundary of the data is outlined using the Rectangle object, similar to the one discussed earlier.

Listing 7-15. 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 in points) {
        distanceList[p] = {};
        distanceList[p].glatlng = new GLatLng(points[k].lat,points[k].lng);
        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 calcu late which points to group. In Listing 7-16 (http://googlemapsbook.com/chapter7/ClientClus ter/), we use a grid method similar to the one we used with the server-side clustering example. The bi ggest difference here is your grid cells will be larger and not as fine-grained, so you donÕt slow down the JavaScript on slower computers. 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

Listing 7-16. JavaScript for Client-Side Clustering

```
var map;
var centerLatitude = 42;
var centerLongitude = -72;
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 in points) {
    var latlng = new GLatLng(points[k].lat,points[k].lng);

    //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";
}
```

Further 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 weÕve presented so far, you may have noticed that your maps ÒflashÓ between redraws and requests. This occurs because the JavaScript removes all the points and then draws them all again. If you donÕt 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 ÒrememberÓ which points are currently on the map and remove only those that arenÕt in the new list. Using the same object, you can also add only those that arenÕt in the old list. Listing 7-17 (http://googlemapsbook.com/chapter7/TrackingPoints/) shows the client-side boundary method from Listing 7-14 modified to keep track of points to remove the flashing between redraws.

Listing 7-17. Modified Client-Side Boundary JavaScript That Remembers Which Markers Are on the Map

```
var map;
var centerLatitude = 49.224773;
var centerLongitude = -122.991943;
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 in points) {
            var latlng = new GLatLng(points[k].lat,points[k].lng);

            //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, itÕs our experience that most users ÒdragÓ 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 around 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 alr eady waiting before they get there.

Additionally, you could also extend the requested bounds beyond the edge of the viewport to include whatÕs 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, weÕve presented a few optimization methods, for both your server and the browser, that 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 donÕt base your decision oncoolnessalone. Creating your own tiles, as in the custom tile method described in this chapter, is pretty neat, but doesnÕt serve well for data that is generated from filtered searches, since each tile will always be different. Also, when using a feature like 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 youÕre looping through large amounts of data and see if you can reduce it further. Just because your web application works doesnÕt mean itÕs working as well as it could. The better optimized your map, the happier your users will be and the better experience theyÕll have.

At the same time youÕre 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, youÕll see some of the possible things Googlemay add, but no guarantees!

CHAPTER 8

WhatÕs 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, itÕs 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, split 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 allowed the maps to support as many detail levels as the satellite photography (or your custom overlay) warrants.

So far, weÕve shown you a lot of really neat techniques and tricks for getting data into your application and onto a map. In the following chapters, weÕll 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 theyÕre likely possibilities, given the demand and interest in them. As developers like yourself push the API further, the demand for new capabilitiesÑsuch as the free geocoderÑbecomes louder, and when Google consents, we get more toys and more fun.

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, youÕll notice a growing interest in the routing system built into http://maps.google.com, as shown in Figure 8-1.

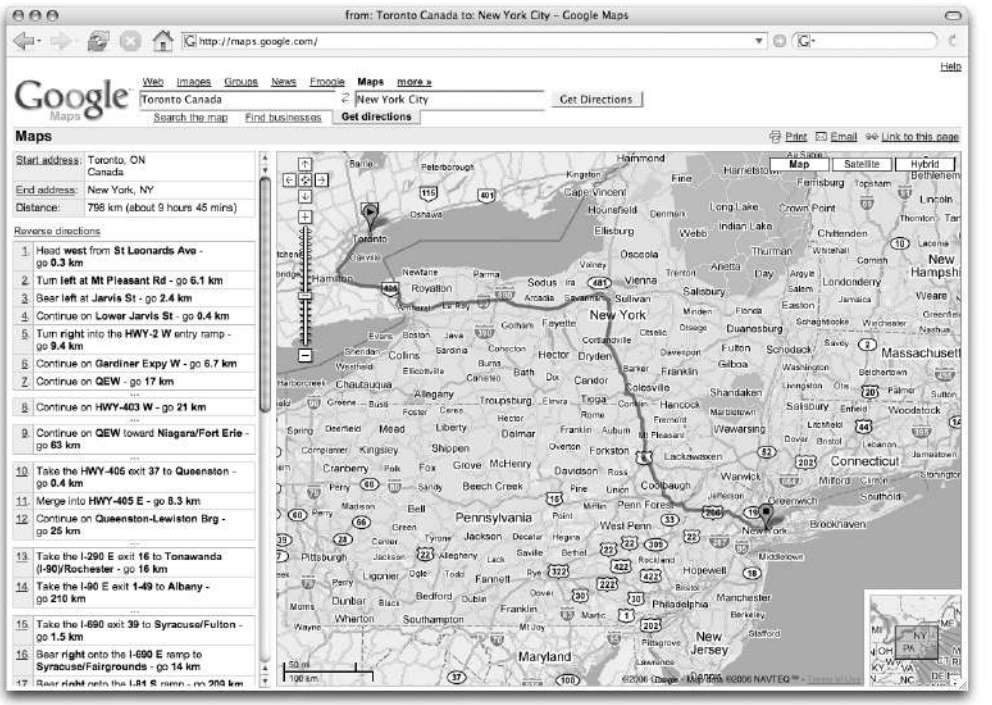


Figure 8-1.Google Maps with a route from Toronto to New York

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 GoogleÕs 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 US Census BureauÕs 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 youÕve 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 GoogleÕs 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 location, as shown in Figure 8-2.

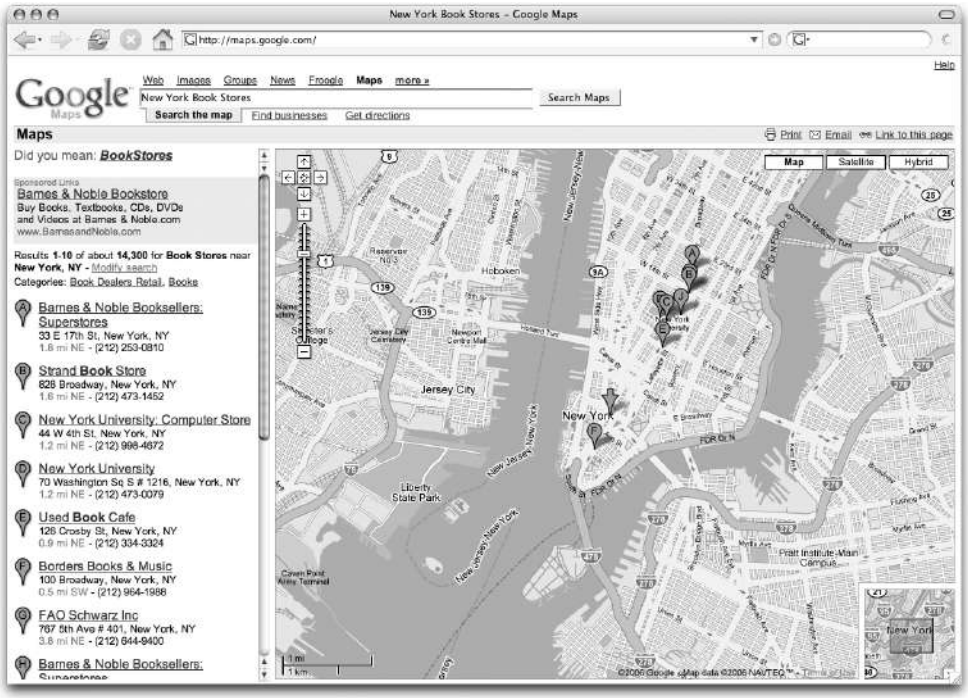


Figure 8-2. Google Maps search for ÒNew York Book StoresÓ

If Google chose to integrate its search database into the Google Maps API, GoogleÕs 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 Òbookstores in New York.Ó The possibility of supplementing your mapÕs proprietary data with GoogleÕs 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 ÒCoffee shops within one mile of StoreLatLngÓ query.

Tip Though not built into the Google Maps API, using GoogleÕs search database is actually possible no by combining some additional Google APIs such as the Google AJAX Search API and maps. For an exam 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 ÒNew York BookstoresÓ 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.

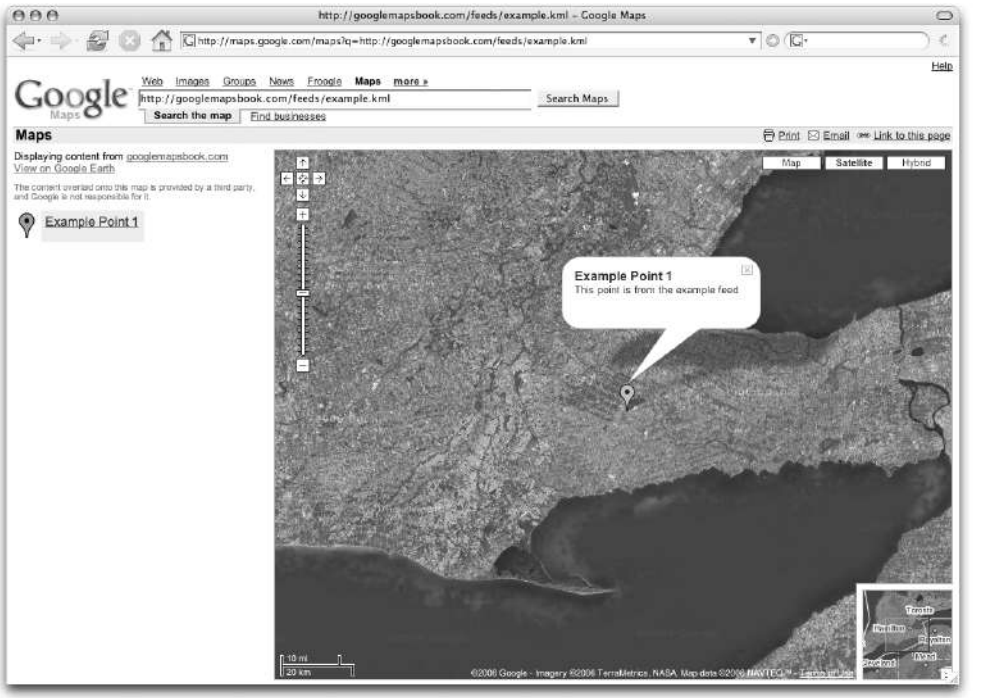


Figure 8-3. Sample KML file in a map

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 hometown. 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 shouldnÕt we expect that it will begin to offer other complementary layers? The data for things like 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, GoogleÕs 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 Òmap-in-a-boxÓ 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 things 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 youÕre 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 browserÕs ability to quickly scale images or move around large numbers of on-screen objects. Other mapping tools such as the Yahoo Mapping API offer alternative Flash clients that can benefit from the performance optimizations of that system. Though Google doesnÕt 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 .

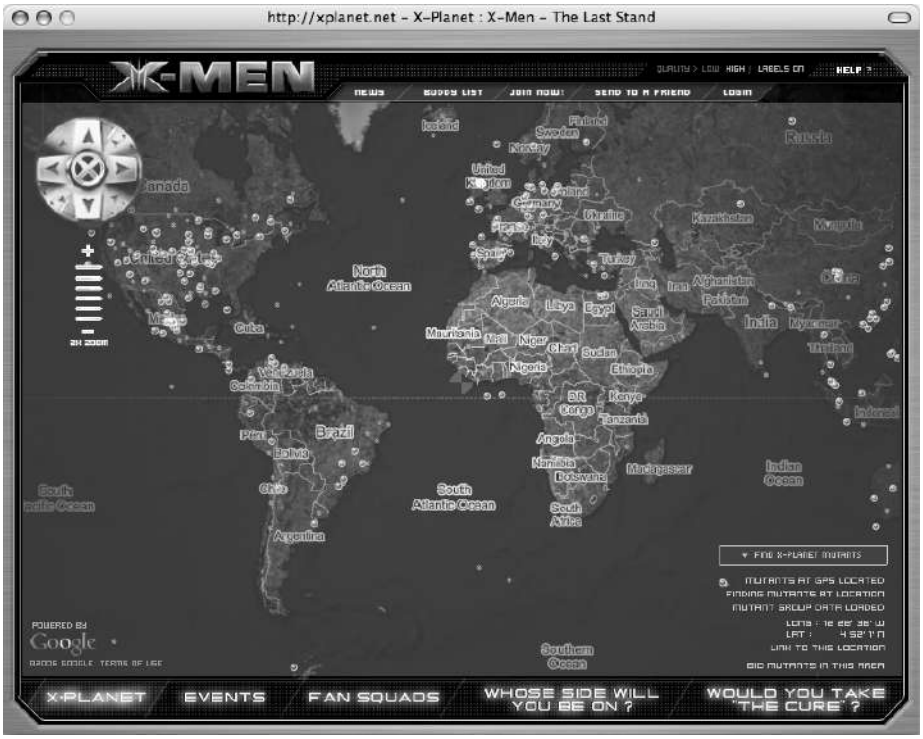


Figure 8-5. The X-Men Flash-based Google map[1]

With the growing competition from Yahoo! Maps and Windows Live Local, Google may come to offer additional options such as a Flash API, or even a next-generation one based on Scalable Vector Graphics (SVG) or some other technology that can bring the browser experience closer to that of Google Earth.

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 newly released 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 advancing mapping techniques. In the next part of the book, weÕll 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, weÕll cover some mathematics you may need in a professional map. Finally, in Chapter 11, weÕll show you how to build your own geocoder from scratch, using a raw data set.

---

1. X-Men and XPlanet.net copyright Marvel, Fox and their related entities.

PART 3

Advanced Map Features and Methods

CHAPTER 9

Advanced Tips and Tricks

Beyond what youÕve seen so far, the Google Maps API has a number of features that are often overlooked. Here, youÕll 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 acts 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, letÕs take a quick look at debugging within the Google Maps API. With the Google Maps API version 1, the debuggerÕs best friend was `alert()`. But as they say, ÒOnly a Lert uses alert to debug,Ó and if youÕve ever accidentally ÒalertedÓ something in a loop, you know what they mean! With Google Maps API version 2, you now have access to the wonderfully simple, yet wonderfully useful, `GLog` class. Now `GLog.write()` is the ÒnewÓ `alert()`, but it creates a floating log window, as shown in Figure 9-1, to hold all your debugging messages.

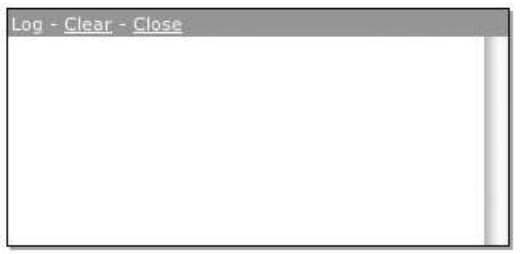


Figure 9-1. Empty GLog window

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 desktop-like 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 mapÕs 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.

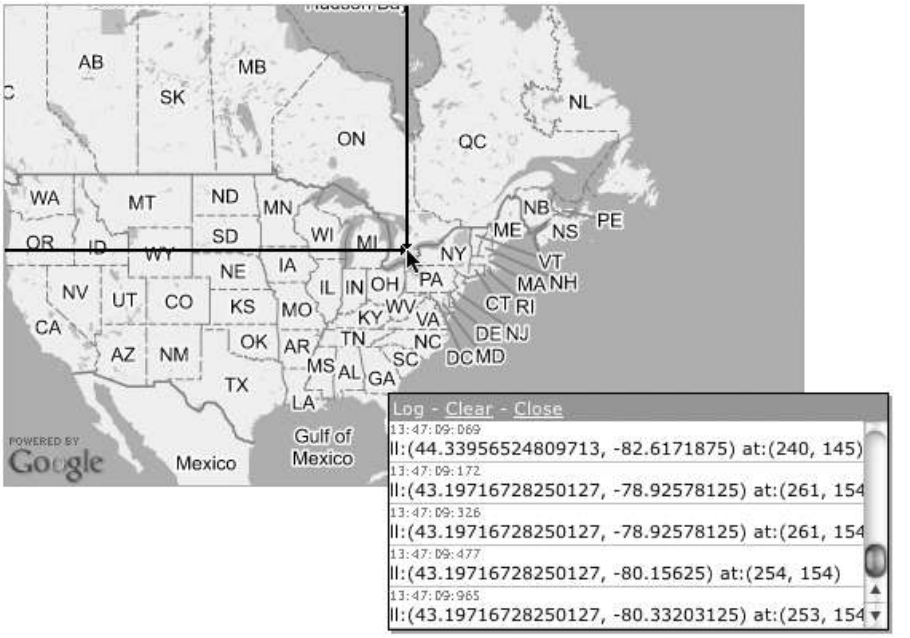


Figure 9-3. Tracking the mouse movement relative to the map container

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, introduced in Chapter 3, 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've 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);
});
```

If you adjust the zoom level of your map, you'll get a log entry that looks something like Figure 9-4.

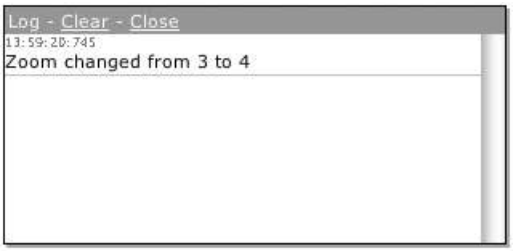


Figure 9-4. GLog entry after changing zoom levels using the zoom control

Notice in Figure 9-4 how the old and new zoom levels are specified. From elsewhere in your web application, you can force the `zoomend` event to execute by calling

```
GEvent.trigger(map,'zoomend');
```

Executing this method will cause the `zoomend` event to run as normal. The problem is that you'll get `undefined` values for both `oldLevel` and `newLevel`, as shown in Figure 9-5.

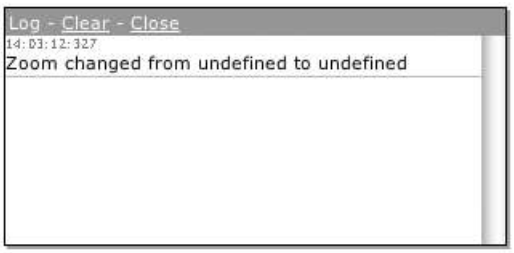


Figure 9-5. GLog entries after triggering zoomend using GEvent.trigger(map,'zoomend')

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 changed the zoom level of the map some other way, the zoom level wouldn'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 GoogleÕs 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 did 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.

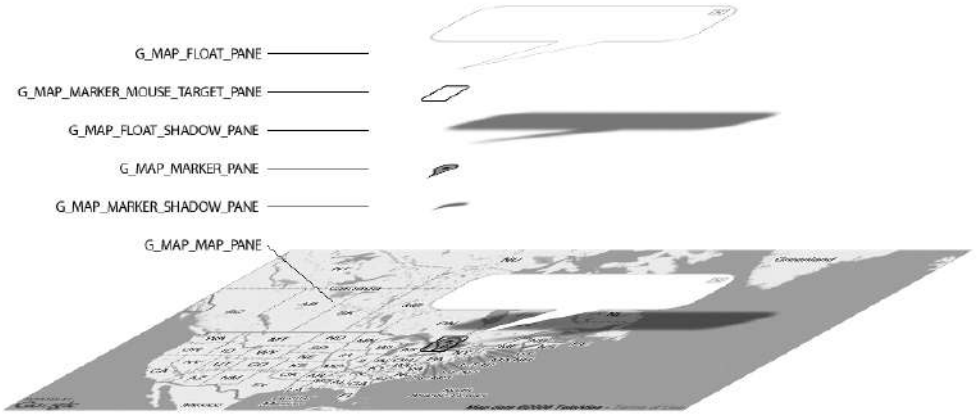


Figure 9-6. GMapPane constants layering

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 like 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 easyGOverlayexample, letÕs 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 tool tip GOverlay, as listed 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 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`, itÕs 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 donÕt accidentally overwrite a property of the GOverlayclass, if Google has used`html` as a property for the GOverlayclass.

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 appropr iate pane using map.getPane(). Also, you probably want to assign the `map`variable to an internal variable so youÕll 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. ItÕs 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 a width was 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 youÕll 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 youÕll 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, 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 (Listing 9-5).

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. Along with version 1's GSmallMapControl, GLargeMapControl, GSmallZoomControl, and GMapTypeControl, the controls GScaleControl and GOverviewMapControl (which shows a little overview window in the corner of the screen) were introduced in version 2 of the API. 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://google mapsbook.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 mapÕs 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, youÕre 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: Anchor to the top-right corner
- G_ANCHOR_TOP_LEFT: Anchor to the top-left corner
- G_ANCHOR_BOTTOM_RIGHT: Anchor to the bottom-right corner
- G_ANCHOR_BOTTOM_LEFT: 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 GInfoWindowmethods, as shown in Listing 9-4.

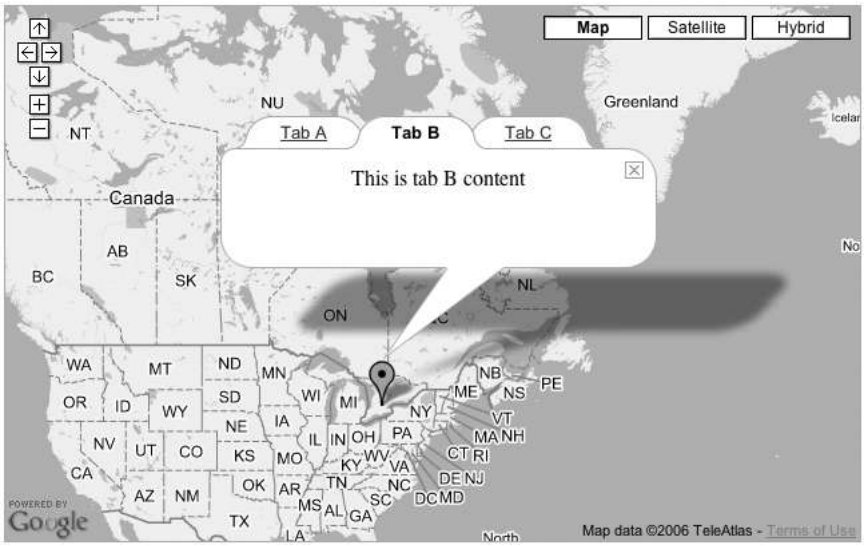


Figure 9-11.A tabbed info window

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 in r ight away:

```
marker.openInfoWindowTabsHtml(infoTabs,{
      selectedTab:1,
      maxWidth:300
});
```

or in 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 work ing, 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.

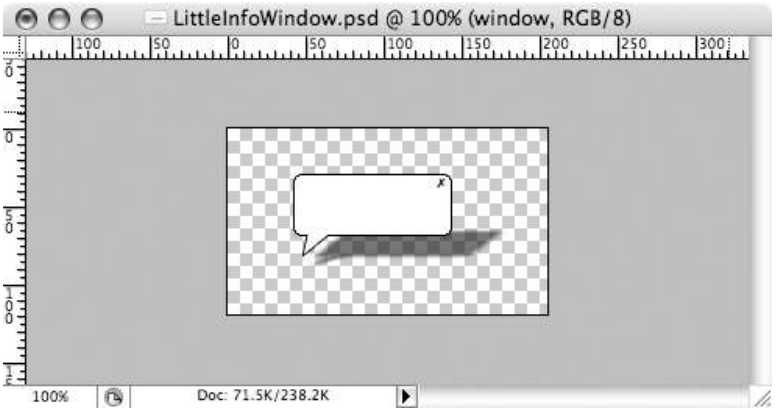


Figure 9-13.The info window art file

The finalized framework for the LittleInfoWindow overlay in Listing 9-5 is almost identi cal to the ToolTip overlay you created earlier in Listing 9-3, 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 make 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 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 Òdrawn Ó 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 allowed the browser to set the offset properties so you could 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 thediv .

Building the Wrapper

Now that you have the size of the content box, the rest is pretty straightforward. First, style the content 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 GoogleÕs 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 the 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's info window is when it opens off-screen, the map pans until the window is visible on-screen. 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://googlemapsbook.com/chapter9/CustomInfoWindow/ , 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 GoogleÕs 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, youÕll 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), which shows the earth using outlines and colored objects, similar to a printed map you might purchase for driving directions
- Satellite, which shows the map using satellite photos of the earth taken from space
- Hybrid, which 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, youÕll 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 is returned from GMapType.getName(true) and is used in the GOverviewMapControl. The default is the same as the name supplied in the GMapType arguments.
urlArg	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	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 weÕve 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% 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 is about half the area of the United States. This is because the Google API uses the Mercator projection.

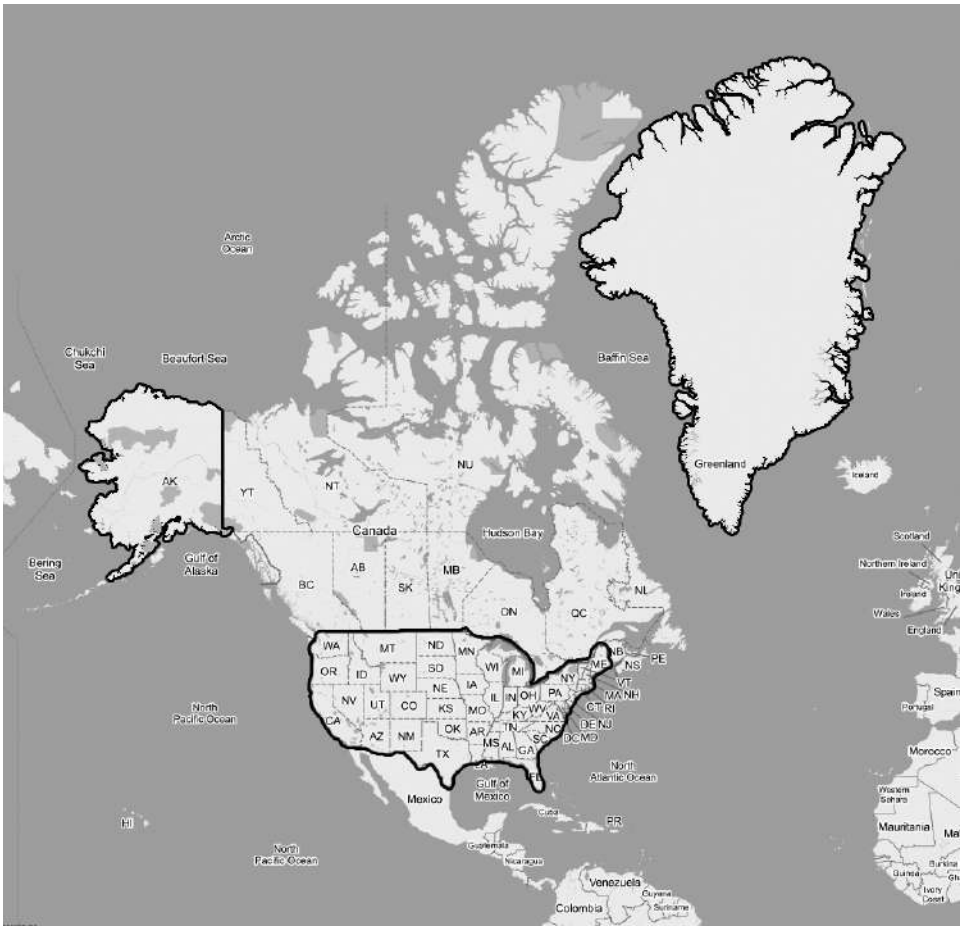


Figure 9-15.Comparing the United States and Greenland on a Mercator projection

Understanding Projection Types

Without going deep into mathematical theories and discussions, map projections can generally be divided into three categoriesÑplanar, conic, and cylindricalÑbut 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.

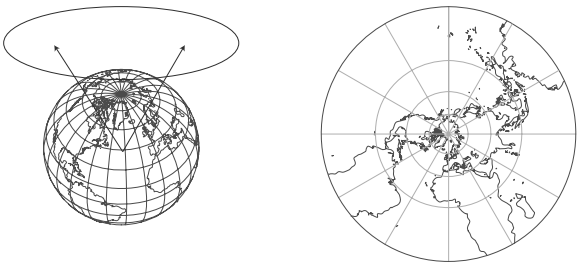


Figure 9-16.Creating a planar projection

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.

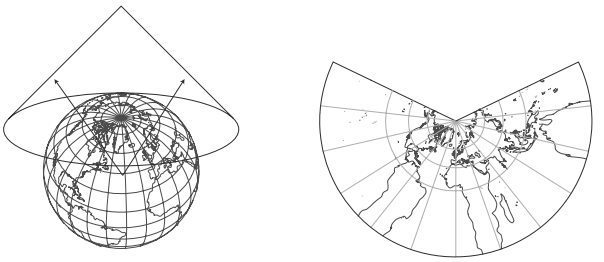


Figure 9-17.Creating a conic projection

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 on to 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 canÕt be represented accurately.

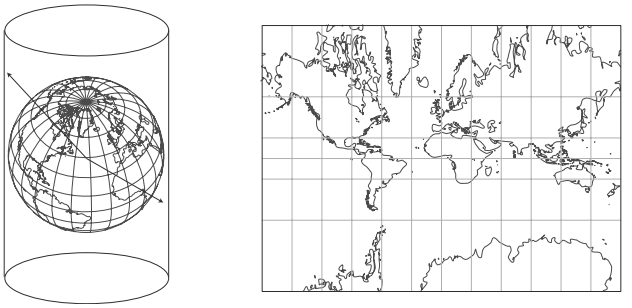


Figure 9-18.Creating a cylindrical projection

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 represents the longitude and represents the latitude:

$$x = \check{S}_{0}$$

$$y = \ln \tan \frac{1}{4} + \frac{1}{2}$$

$$= \frac{1}{2} \ln \frac{1+\sin}{1\check{S}\sin}$$

$$= \sinh^{\check{S}1}(\tan)$$

$$= \tanh^{\check{S}1}(\sin)$$

$$= \ln(\tan + \sec)$$

The equations preserve more realistic shapes, as shown in Figure 9-19.

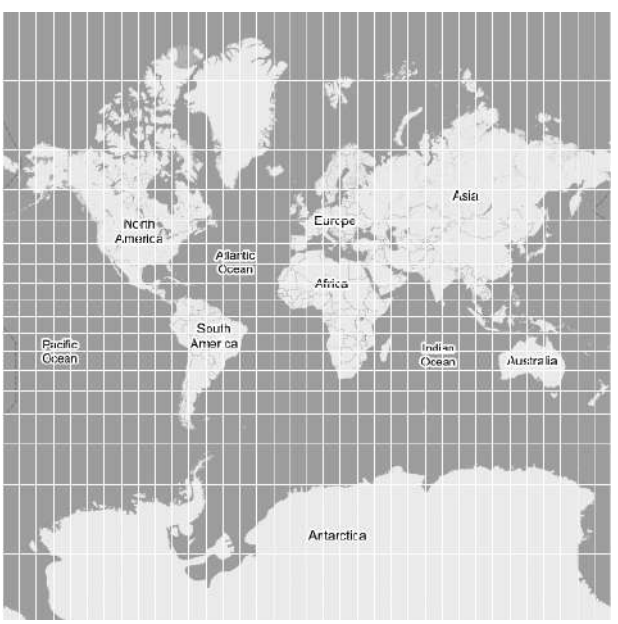


Figure 9-19.Latitude and longitude lines of the Google Maps APIÕs Mercator projection

The downside with Mercator projections, as you saw in Figure 9-15, is that areas farther away from the equator are greatly exaggerated and the poles themselves canÕt 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 donÕt 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 the latitude, longitude from the GLatLng object, and zoom level,returns the X and Y pixel coordinates of the location relative to the bounding div of the map.
fromPixelToLatLng (pixel,zoom,unbounded)	GLatLng	Reverse offromLatLngToPixel. Given the pixel coordinates and zoom, returns the geographical latitude and longitude on 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	Returns true if the tile index is within a valid range 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, 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 youÕll use in the ÒThe Blue Marble Map: Putting it All TogetherÓ section later in the chapter to create a map using the NASA Visible Earth images as tiles. 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 GoogleÕs 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.

Listing 9-6. Equidistant Cylindrical GProjection

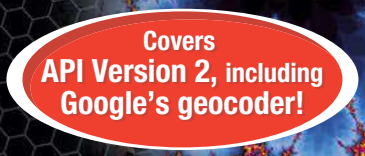
```
EquidistantCylindricalProjection = new GProjection();

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;
```

THE EXPERT'S VOICE® IN WEB DEVELOPMENT



Beginning Google Maps Applications with PHP and Ajax

From Novice to Professional

Build awesome web-based mapping applications with this powerful API!

Michael Purvis, Jeffrey Sambells, and Cameron Turner

Foreword by Mike Pegg, Founder of the Google Maps Mania Blog

Apress®