GOOGLE MAPS HACKS

Tips and Tools for Geographic Searching and Remixing

O'REILLY®

Rich Gibson & Schuyler Erle

Google Maps Hacks™
by Rich Gibson and Schuyler Erle

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North,
Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Simon St.Laurent
Production Editor: Jamie Peppard
Copyeditor: Derek Di Matteo
Indexer: Nancy Crumpton

Cover Designer: Marcia Friedman
Interior Designer: David Futato
Cover Illustrator: Karen Montgomery
Illustrators: Robert Romano, Jessamyn Read, and Lesley Borash

Printing History:

January 2006: First Edition.

This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 0-596-10161-9
[M] [5/06]

HACK #60 Make Things Happen When the Map Moves

You can make your maps more interactive by responding to user-initiated events.

Balancing what to show and what not to show on a Google Map can be tricky. On one hand, for performanc reasons, you want to restrict the overlays displayed on the map to just the ones that would appear within the current view. On the other, if the user should pan or zoom the map, you might want the map's contents to change in response. Fortunately, the Google Maps API offers a way to make this happen automatically.

The Hack

Poor Clio's Side-by-Side Google Maps at *http://jamesedmunds.com/poorclio/googlemap11.cfm* offers a terrific example of how Google Maps hacks can respond to user actions. The site, built by James Edmunds, shows two Google Maps of the same location side by side. On the left, the regular map mode is shown, while the map on the right shows the same map in satellite mode. Figure 6-20 shows the side-by-side view of the area around 30th Street Station in Philadelphia, with the rail yards extending away to the north.

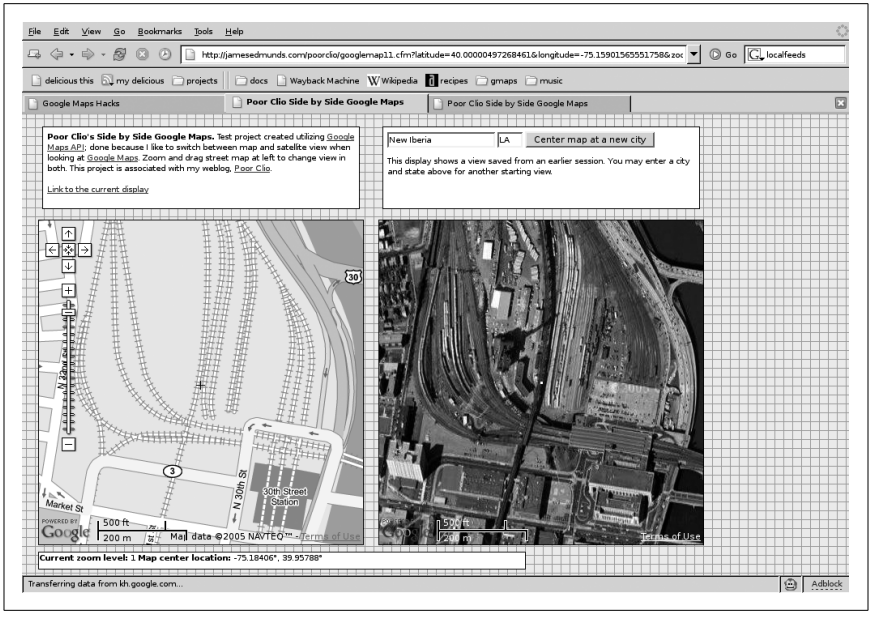

Figure 6-20. Two views of Philadelphia's 30th Street Station, side by side

What's novel about this site is that if you drag the map on the left with your mouse, double-click to recenter it, or zoom in or out, the satellite view on the right recenters and zooms to match. Until Google introduced hybrid mode maps, this was effectively one of the only ways to compare the satellite and map views. The secret to how it works lies in the Google Maps Event API.

The Code

If you've written or looked at JavaScript code that uses the Google Maps API to cause info windows to pop up when the user clicks on a marker, then you've probably seen a method call that looks like this:

```
GEvent.addListener(marker, "click", function( ) {
    marker.openInfoWindowHtml(html);
});
```

This method uses the `GEvent` class to register a click event on a marker object. When the marker is clicked, the anonymous function gets called, and the info window is opened. As it happens, the Google Maps API offers several other events for tracking and responding to user interactions.

Here's a snippet of code from the Poor Clio's Side-by-Side page:

```
function onMapMove( ) {
    smap.centerAndZoom(map.getCenterLatLng( ), smap.getZoomLevel( ));
    updateStatus( );
}

function onMapZoom(oldZoom, newZoom) {
    smap.centerAndZoom(map.getCenterLatLng( ), newZoom);
    updateStatus( );
}

GEvent.addListener(map, 'move', onMapMove);
GEvent.addListener(map, 'zoom', onMapZoom);
updateStatus( );
```

These dozen or so lines of JavaScript code handle virtually all of the user interactions needed to keep the two maps in sync. The first thing to note about this code is that, unlike most Google Maps API examples, you'll see *two* `GMap` objects in this code, one called `map` in the JavaScript, and the other called `smap`. Well, why not?

The first of the two functions shown above, `onMapMove( )`, takes the `GPoint` object representing the latitude and longitude of the center of the regular map, and the integer representing the current zoom level of the regular map, and passes them to the `centerAndZoom( )` method of the satellite map. When this happens, the satellite map automatically zooms and recenters to match

the center point and zoom level of the regular map. The `onMapZoom( )` function does basically the same thing, but it takes two arguments, representing the old and new zoom levels of the regular map, and uses the new zoom level of the regular map as the new zoom level of the satellite map. The two calls to `GEvent.addListener( )` hook these functions into the move and zoom events on the regular map, respectively, so that when the user changes the view of the regular map, the satellite map follows.

The calls to `updateStatus( )` interspersed through the code above cause the latitude, longitude, and zoom level display below the two maps to keep in sync as well. The code looks like this:

```
function updateStatus( )
{
    var point = map.getCenterLatLng( );
    var status =
        "<b>Current zoom level:</b>  " + map.getZoomLevel( ) +
        " <b>Map center location:</b> " +
        Math.round(point.x * 100000) / 100000 + "&deg;, " +
        Math.round(point.y * 100000) / 100000 + "&deg;";
    document.getElementById("status").innerHTML = status;
}
```

The use of `Math.round( )`, coupled with multiplication and division by a constant power of ten, is a standard trick for rounding a floating-point number to a given number of decimal places. The call to `document.getElementById( )` fetches the HTML Document Object Model element containing the status message, and the assignment to the `innerHTML` property updates the HTML inside with the latest center point and zoom.

This same kind of technique can hypothetically be used to have a "Link to this View" URL track the current view, although Poor Clio's doesn't do it that way. Assuming that you had an HTML `<a>` element with an `id` attribute set to `linkHere`, you could do something like the following to update the URL in the `href` attribute:

```
document.getElementById("linkHere").setAttribute( "href",
    "http://some.server.net/gmaps?ll
       =" + point.x + "+" + point.y + "&z=" + zoom );
```

There's one other trick that the Poor Clio's site uses to keep the maps in sync. Because the satellite half of this page doesn't have zoom and pan controls, the only way to move the satellite map independently is to drag it with the mouse. The site uses this line of JavaScript to keep such a thing from happening.

```
smap.disableDragging( );
```

The other way to solve the problem of the maps potentially getting out of sync, of course, would be to make the event handlers a bit more generic and

then add two more calls to `addEventListener()` to make the regular map follow movements on the satellite map as well.

The GEvent API

The key thing to note about the GEvent API is that all its methods are static, which is to say that they're called on the class itself, and not on an instance of the class. This means you say:

```
GEvent.addListener( ... );    // RIGHT
```

...as opposed to the following, which you should definitely *not* use:

```
var event = new GEvent();    // WRONG
event.addListener( ... );    // WRONG
```

We mention this because it may run contrary to your expectations, as most of the other classes in the Google Maps API require you to instantiate an object of that class before calling methods on it.

The GEvent API shouldn't be mistaken for the JavaScript event API, which works rather similarly, with its `onClick`, `onMouseOver`, `onChange` methods, and so on. In general, you'll want to use GEvent handlers for Google Maps API objects, and continue to use JavaScript event handlers for everything else. In particular, as of this writing, there isn't anything like an `onMouseOver` event for Google Maps marker objects, and more's the pity. Perhaps Google will fix this in a future version of its API.

Hacking the Hack

Most Google Maps hacks out there rely on an Update Map button (or the equivalent) to update the overlays on a map after the user zooms or pans. This definitely works, but it's not the most elegant solution for refreshing the map's contents, because it relies on the user to do something that could be done automatically in JavaScript.

However, before you run out and add event handlers to your Google Map, consider the following: the Google Maps API doesn't guarantee exactly *when* the registered event handlers will be called, just that they will be called at some point. If you try dragging the regular map around on the Poor Clio's site, you'll notice that there's sometimes a bit of lag, but (assuming you don't drag too quickly) the satellite map otherwise follows along pretty faithfully. This means that the `move` event on the regular map is getting triggered every so many seconds or milliseconds throughout the drag.

If your intention is to have one map track another, this is definitely what you want to have happen. On the other hand, if you're using the event handler to trigger a download of fresh data off the network, you probably don't want to have multiple overlapping data requests slowing everything down while the user is still trying to drag the map.

First, this means you want to use the `moveend` event with the GMap object, and not the `move` event. However, if you really want to be on the safe side, you might do well to combine this with a semaphore and a delay, in case the user does a bit of fine adjustment to recenter or zoom the map to exactly what they want to see. Here's what the code might look like:

```
var updateRequested = 0;

GEvent.addListener( map, "moveend", function( ) {
    updateRequested++;
    window.setTimeout( beginUpdate, 2500 )
});

function beginUpdate ( ) {
    if (--updateRequested > 0) return;
    // *now* check map.getCenterLatLng( ) etc.
    // then launch a new GXmlHttp request
    // and do the update handling when the request is complete
}
```

The use of the global `updateRequested` variable acts as a semaphore to keep multiple events from triggering multiple network updates that would then stomp all over each other. When a `moveend` event is triggered, `updateRequested` is incremented from zero to one, and the network update is scheduled to begin 2,500 milliseconds (i.e., 2.5 seconds) later. In the meantime, further `moveend` events will increment `updateRequested` by 1 each time. When `beginUpdate( )` gets called 2.5 seconds after the first `moveend` event, it will decrement `updateRequested` by 1, and then check to see if the semaphore is still greater than zero. If it is, then subsequent movements of the map must have occurred in the meantime, and the update is therefore deferred.

Finally, the last `beginUpdate( )` will trigger 2.5 seconds after last `moveend` event, at which point the user has stopped moving the map around, the `updateRequested` semaphore will be decremented back to zero and, then and only then, the update will occur. Choosing the right delay so that network updates are smooth and non-overlapping, but so that the user experience isn't too terribly degraded, is probably a matter that merits some experimentation.