Google Maps Developer Tutorial

Brenn Berliner, 2008

Overview

This document describes the procedure for setting up an interactive Google Maps instance with clickable country polygons. Refer to the Global Health map (http://depts.washington.edu/deptgh/map/) to view one such application in action. LST staff can also provide additional information by request if you would like further documentation of a working setup.

The steps involved are roughly the following:

1. Download .shp border data
2. Convert .shp data into .kml
3. Encode .kml into Google's format
4. Store encoded points/lines in database
5. Create PHP script to access data
6. Add HTML elements to page
7. Initialize map in Javascript
8. Register function callbacks
9. Create click handler function
10. Create XML parser function

Data Source

The first step is to find the appropriate border data. Typically this will be available on the web in the form of shape files (*.shp). In our case we used a freely available archive called world_adm0.zip which contains border data for most of the world's countries. Note that you will also need to have two metadata files associated with the .shp, a .dbf and a .shx; these should be included by default. All of this information should still apply if you are working with U.S. States or other geographic areas.

Next, the border data file must be converted from .shp format into KML. This is a form of XML specifically designed for GIS use. The conversion can be accomplished using the free program Shp2Kml (http://shp2kml.sourceforge.net/). Simply run the application and provide it with your .shp file. The default settings on subsequent tabs should be fine; save the .kml file when done. Note that the generated file will often exist as one extremely long line of code, so you may wish to insert line breaks manually to improve the readability of the contents.

Encoding

In order for border data to be recognized by the Google Maps API, it needs to be encoded in a certain format. The algorithm has been documented online courtesy of Mark McClure (http://facstaff.unca.edu/mcmcclur/GoogleMaps/EncodePolyline/), and it has also been ported to several different languages for practical use. We have written a small PHP wrapper class for the algorithm which handles automatic database inserts of encoded points/levels, though this is not strictly necessary.

Although you should not need to modify the algorithm itself, depending on the amount of data you are

serving, it may be useful to specify the level of detail. This can improve performance at the cost of aesthetic appearance. In the PHP version, changes can be made by adjusting the variable $verySmall. The default is 0.00001, representing the minimum latitude/longitude difference for a line to be created between two points. It is recommended that you change this parameter in increments of 0.000005 until you find a setting that is acceptable for your project.

Database

The output of the encoding process produces two strings, representing encoded points and their respective levels. The points translate to latitude/longitude coordinates on the map, while the levels define which points will appear at various levels of zoom. In other words, as you navigate around the map, the resolution of the displayed points will increase or decrease according to the zoom setting and the visible area. In terms of data storage, you should place this information in three columns: the two described above along with a country ID or name. You could optionally include other data here as well if not using a relational table configuration.

PHP/HTML

The next step is to create a server-side script in PHP capable of querying the above database and returning point/level data. Typically this is done within an AJAX framework, as we will discuss below. The returned content should be in XML format, so remember to set the content-type header properly. You can use the traditional SQL query and while loop here. Specifically, the return format should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<Document>
<Placemark name="country_name" id="country_id">
<polygon points="encoded_points" levels="encoded_levels" />
</Placemark>
</Document>
```

Note that each set of point/level pairs corresponds to an individual polygon. This may or may not comprise an entire country. For example, Canada contains many islands, each of which is stored as an individual entry since the polygons need to be able to close. Make sure your client-side code accounts for both possibilities. In general, every instance of the <Placemark> tag represents one row in the database.

The HTML component is quite simple. Just include something like this in your page:

```
<div id="map"></div>
```

Javascript

The majority of any Google Maps application is of course written in Javascript. Start by registering your site's domain to receive an API key, if you haven't already done so. Then refer to the Google Maps API to set up the basic map framework, attaching it to the div created earlier and loading the desired controls. Ideally this should all be in a function that gets called in your body onload(). At a minimum, you will need to set up callbacks to handle loading of data and clicking on polygons. Here is a basic constructor:

```
if (GBrowserIsCompatible()) {
  map2 = new GMap2(document.getElementById('map'));
  GEvent.addListener(map2, "click", handleClick);
  GDownloadUrl("download.php", parseOverlays);
}
```

Next, create the handleClick callback function. There is much more you could potentially do here to customize your map's behavior, but this example should cover the basics of capturing click events, forwarding them to an individual country page if an overlay was clicked, or ignoring them if not.

```
function handleClick(overlay) {
  if(overlay) { //clicked on an overlay
    if(overlay instanceof GPolygon) { //clicked on a polygon
      location.href = 'country.php?id=' + overlay.extra.id; //redirect browser
    }
  }
}
```

Parser

The XML parser is the meat of the application, translating placemark data into GPolygon objects and adding them to the map. As you can see, this function makes heavy use of JSON notation in setting object attributes. Many of these can be removed if they're not needed in your particular case. Use the included comments to identify the different sections of the code and their purpose.

```
function parseOverlays(data, responseCode) {
  var xml = GXml.parse(data); //parse xml
  var placemarks = xml.documentElement.getElementsByTagName("Placemark");
  var i, j, boundaries, obj, poly, polys, c;

  //loop through placemarks
  for(i=0; i<placemarks.length; i++) {
    polys = placemarks[i].getElementsByTagName("polygon");
    boundaries = new Array(0);

    //loop through borders
    for(j=0; j<polys.length; j++) {
      obj = {
        color: "#000000",
        weight: 1,
        opacity: 1,
        points: polys[j].getAttribute("points"),
        levels: polys[j].getAttribute("levels"),
        numLevels: 18,
        zoomFactor: 2
      };
      boundaries.push(obj);
    }

    //create actual polygon
    poly = GPolygon.fromEncoded({
      polylines: boundaries,
      color: "#222167",
      opacity: 0.4,
      fill: true,
      outline: true
    });

    //add additional country info to poly object
    c = poly.getBounds().getCenter();
    poly.extra = {
      name: placemarks[i].getAttribute("name"),
      id: parseInt(placemarks[i].getAttribute("id")),
```

```
      center: c,
      pxcenter: map2.fromLatLngToContainerPixel(c)
    };

    map2.addOverlay(poly); //add polygon to map
    boundaries = obj = polys = poly = null;
  }
  xml = placemarks = null;
}
```

Enhancements

There are a variety of other things you could do to improve the functionality of the map. Here are some examples that might be inserted into the parseOverlays function. Note that these utilize the jQuery library, although they can be easily rewritten in standard Javascript. First, tooltips:

```
//catch poly hover events
GEvent.addListener(poly, "mouseover", function() {
  $("#mapbox").append($("<div></div>").addClass('maptooltip').css({
    left: this.extra.pxcenter.x,
    top: this.extra.pxcenter.y
  }).text(this.extra.name));
});
GEvent.addListener(poly, "mouseout", function() {
  $(".maptooltip").remove();
});
```

If you want a loading icon that will disappear when the map is ready, add this line to the HTML page:

```
<div id="maploader"><img src="img/loading.gif" border="0" /></div>
```

Then add the following Javascript line to the end of the parseOverlays function:

```
$("#maploader").css("visibility", "hidden");
```

Optimization

We have found that certain issues can arise when loading large quantities of data onto the map, particularly in Internet Explorer. To avoid memory leaks in which the map eventually consumes more and more system resources each time it is rendered, put the following line in the body tag of your HTML:

```
onunload="map2=null; GUnload();"
```

Depending on how many polygons are being created, you might also receive “slow script” warnings as the parseOverlays function is running. These can be very confusing to the user as well as causing IE to hang on some occasions. The solution is to split up the AJAX calls, modifying the PHP script to return only some of the rows from the database each time. You will need to keep track of which page of results you are on with a global Javascript variable. Placing the following code near the end of parseOverlays does the trick via timeouts, with the extra advantage of rendering polygons in stages, providing better visual feedback to the user:

```
if (placemarks.length > 0) {
  atpage++;
  var args = "\"download.php?pg="+atpage+"\"";
```

```
  setTimeout('GDownloadUrl('+args+', parseOverlays);', 1);
}
```

Integration

Once you have a working map instance configured, you may wish to incorporate hooks into other related data sources maintained by your department. For instance, in the case of the Global Health map, it was desirable to integrate faculty expertise information into the map application as a whole. Specifically, we elected to use an existing departmental XML feed to populate a set of faculty data tables on a nightly basis. While the particulars of such a setup will depend on how you choose to make this information accessible, we will cover one potential option here.

To begin, you will need to create a small application capable of producing the source XML feed. This can be done in PHP or any other scripting language; if your department uses an enterprise database system, it may be able to handle this procedure for you automatically. Essentially, you just need to ensure that the output feed contains all of the desired fields that your map application will expect. Once these have been selected, it is essential to standardize the tag hierarchy so that your parser can locate the appropriate data within the feed. The feed itself can either be live or can be updated programmatically at fixed intervals, as long as the URL remains consistent. An pseudo-code example of how this might look appears below.

```
<Department>
      <Person>
            <First>John</First>
            <Last>Doe</Last>
            <Field>Value</Field>
      </Person>
      <Person>
            <First>Jane</First>
            <Last>Smith</Last>
            <Field>Value</Field>
      </Person>
</Department>
```

Next, utilize PHP's built-in XML processing functions to create a script that can access individual data fields within the published feed. Typically this is done via a series of nested loops. After reading in the file and instantiating the parser, you can proceed to loop through the individual faculty records; the innermost loop would then handle the individual fields associated with each faculty member. The resulting data should be saved into an array or other data structure as it is read in. Note that any validation of field contents should also be handled here. Lastly, when the end of the feed is reached, the script must be able to actually update your database, keeping in mind the distinction between adding new records and updating existing ones.

Finally, this process will have to be scheduled so it can run on its own. In most cases you can use a cron job for this purpose, provided you have the necessary system privileges. Simply invoke the crontab command to edit (-e) or view (-l) the list of scheduled jobs. The syntax allows for five fields pertaining to time, the first two of which represent the minute and hour, respectively. For example, the following line will execute script.php every night at 3:30 AM. It is important to make sure the path to PHP is accurate for your system. After the script has populated your database with the appropriate records, you can create the desired associations; for instance, the country ID and person ID could be linked by the use of a foreign key relationship.

```
30 3 * * * /usr/local/bin/php /path/to/your/script.php
```