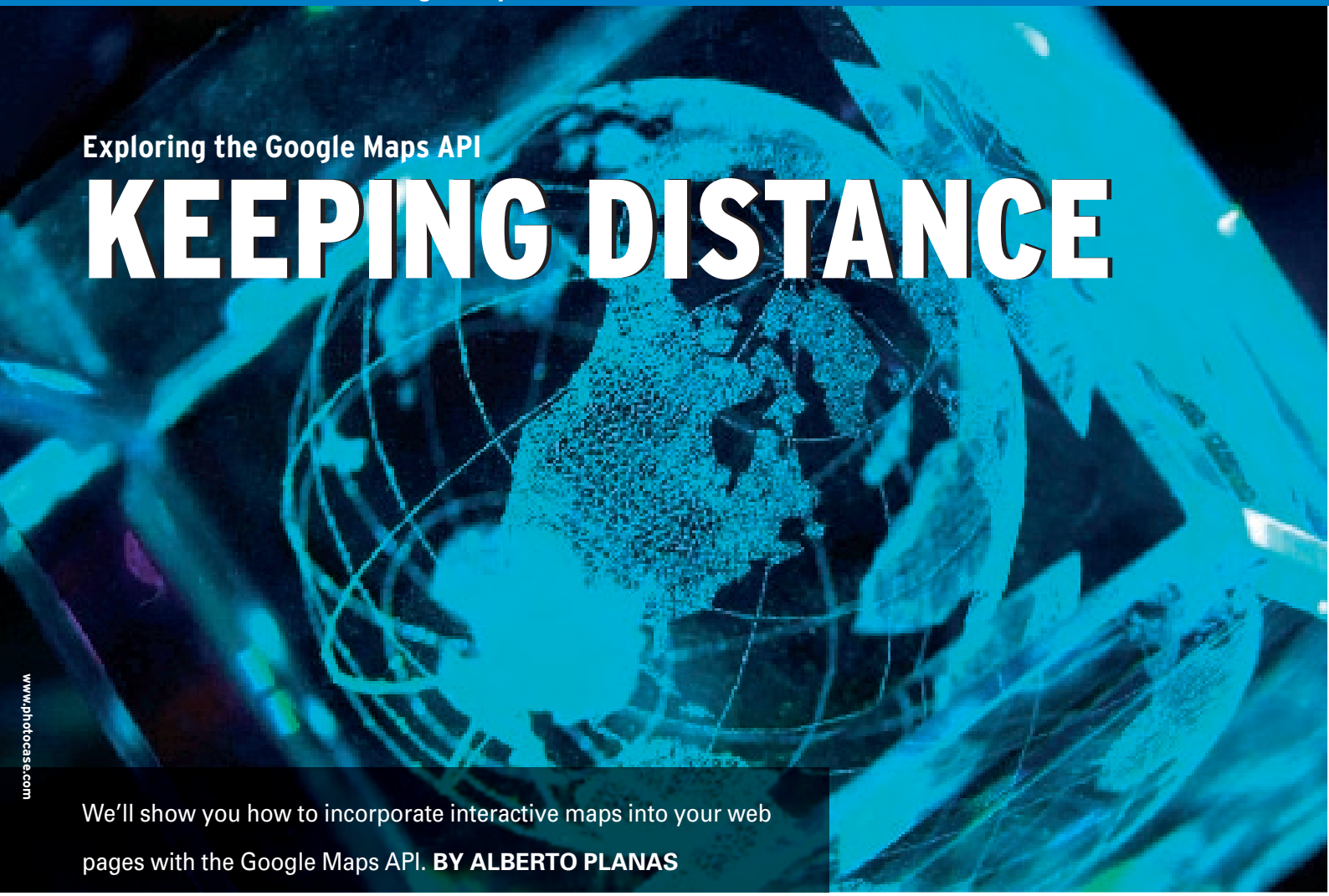

Exploring the Google Maps API

KEEPING DISTANCE

We'll show you how to incorporate interactive maps into your web pages with the Google Maps API. **BY ALBERTO PLANAS**

The engineers at Google have released a JavaScript(or ECMAScript[1])-based API that lets users create programs that incorporate maps and satellite pictures of the Earth. The Google Maps API even lets you embed a map-enhanced application in a website so that it will run from a browser without any additional server-side infrastructure. We tried out the Google Maps API on an example program that calculates the distance between points on a map.

The Key

The Google Maps API is available without charge as long as users follow some guidelines specified by Google [2]. Users must make any creations freely available to the public, not exceeding a specific number of queries each day, and not hiding the Google brand wherever it may be displayed. In order to use the API, you must obtain a *key* identifying yourself as a user. You can get a key from [3] as long as you have a Google account. Any Gmail account will be enough, but if even if you are not a Gmail user, you can create an account. Each key is associated with an access URL. If you do not have web space to host your Google Maps application, but you have an Apache server running locally, you can register the address *http://localhost*.

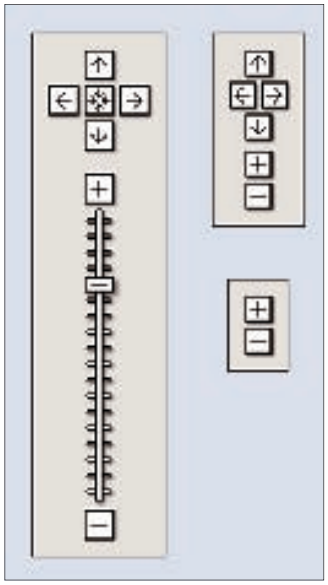

Figure 1: The big control on the left corresponds to a GLargeMapControl, the medium sized one on the right is a GSmallMapControl, and the small control is a GSmallZoomControl.

You must be careful when indicating the web address, because if you enter a misspelled or incomplete address, you won't be able to access the services offered by the Google Maps API. It is important to write the complete address, including ports and directories. For instance, in order to perform tests on my laptop – I have an Apache server installed on port 8080 – I created a directory named *maps*. This is where my website will be hosted. So, the address I use to generate the key is:

```
http://localhost:8080/maps/
```

You can create as many keys as required. Once the terms of use have been accepted, you will receive a long alphanumeric string, which will be stored in a file for subsequent use.

The Flight of the Phoenix

The official documentation of the API [4] recommends XHTML instead of ordinary HTML. The reason for this is the increased portability of XHTML documents. Listing 1 is an example web document that accesses the Google Maps API. The XHTML format is declared by means of the *DOCTYPE* located at Line 1. Line 2, and Lines 5 through 9, enable Internet Explorer to correctly display

Figure 2: Map of an area in Palo Alto.

Figure 3: Satellite picture of the same area.

some of the effects supplied by the Google Maps JavaScript library (particularly path lines, as explained later in this article).

To include the JavaScript file containing the API code, use a command similar to Line 10 in Listing 1. We have to substitute *...&key=XXXXX* for the key previously generated by Google. Curiously, all the XHTML in this first example remains confined between Lines 27 and 29. It is at that point when, after loading the HTML document, the browser must execute the JavaScript function *onLoad*, which is defined at Line 14. At Line 28 we find a * * sized at 500 x 500 pixels with a *map* identifier, which I will describe later in this article.

The *onLoad* function initializes the map. As I have mentioned, the entire API is written in JavaScript, and such code will run in the user's browser. Not all browsers supply the same JavaScript version and functionality, and this causes some incompatibilities. To make sure the program will run in the browsers supported by the API, we will call the *GBrowserIsCompatible* function (Line 15). If the user is using Firefox, Safari, Opera, or IE 5.5 (or higher) we will not have any kind of problems. In the next line, we create an object of the *GMap* type. We pass the * * object identified by *map* to the constructor. The HTML object will be used by *GMap* to insert a map of the size associated with its tag. *GMap* offers an interface that can be consulted in the official documents [3] (we will notice that there are several

Listing 1: test1.html

```
01 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
02 <html xmlns="http://www.w3.org/1999/xhtml" xmlns:v="urn:schemas-microsoft-com:vml">
03   <head>
04     <title>Ejemplo 1 - test1.html</title>
05     <style type="text/css">
06     v\:* {
07       behavior:url(#default#VML);
08     }
09     </style>
10     <script src="http://maps.google.com/maps?file=api&v=1&key=XXXXX" type="text/javascript"></script>
11     <script type="text/javascript">
12     //<![CDATA[
13 14     function onLoad() {
15       if (GBrowserIsCompatible()) {
16         var map = new GMap(document.getElementById("map"));
17         map.addControl(new GSmallMapControl());
18         map.addControl(new GMapTypeControl());
19         map.addControl(new GScaleControl());
20         map.centerAndZoom(new GPoint(-122.1419, 37.4419), 4);
21       }
22     }
23 24     //]]>
25     </script>
26 27   <body onload="onLoad()">
28     <div id="map" style="width: 500px; height: 500px"></div>
29   </body>
30 </html>
```

Figure 4: Superimposition of the map with the satellite picture. We can see the names of the streets on the real streets themselves.

constructors for this object), but already at Line 17, we can see that we can make calls to the *addControl()* method. We will use this method to add several controls to the map that will allow us to modify its behavior. We can scroll across the map simply by pressing the left mouse button and dragging the pointer around, but we can also use a component associated with the map that will also allow us to change the zoom level. This component is the one we add at Line 17 in Listing 1, and it corresponds to the one located in the upper right corner of Figure 1. We can play with the code, changing this line to:

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

In this way, we can include separate position and zoom controls. There are another two control types we can add: a map type selector and a scale measurement control, both in miles and kilometers. Of these two, the map type selector is the most important, and it is included in Line 18. There are three map types: normal, satellite, and hybrid (a superimposition of one on the other). An example of each type is shown in Figures 2 through 4. Up to this point, classic maps with street names and addresses are available only in the US, UK, and Japan.

At Line 20 we center the image, specifying the latitude, longitude, and zoom level. For this example we have chosen the coordinates of Palo Alto, California. We have to be careful when indicating the coordinates of the point where we want to center the map. First we must indicate the longitude and then the latitude. In general, we will find the coordinates of interesting areas in reverse order: latitude, longitude.

After this introduction, we will add the part of the program that calculates distances (Listing 2).

Sphere

Any point on the surface of Earth can be located using

Figure 5: Path of 0.77 Km drawn on the map. We can add and remove as many red marks as we want in order to draw a different path.

two numbers: latitude and longitude. But if we just know the coordinates of two points on the surface of a sphere, we will not be able to calculate the distance between them; we need another piece of information: the radius of the sphere. If we consider that the Earth has an average radius of about 6378.7 Km (the Earth is not a perfect sphere), we will be able to use spherical geometry to calculate distances based on latitude and longitude values [5]. One common approach converts both latitude and longitude from degrees to radians, and then applies the formula:

```
d =  6378.7 * acos(sin(lat1) ➥
* sin(lat2) + cos(lat1) ➥
* cos(lat2) * cos(lon2 - lon1))
```

This equation does not require further discussion, since its usage can be examined in the function *calcDistance()* in Listing 2 (see page 28).

Listing 2 is subtly different from the first example. The first difference is that, in the call to the *GMap* constructor, we are supplying an extra parameter (Line 21). This parameter is a single element array. We use it to indicate that we only want to display the satellite mode.

Google's API allows you to associate functions with a predefined list of events. In this way, we can alter the behavior of the system in response to a mouse click, a scroll command, or the addition of a mark. You'll find a complete list of supported events in the documentation. This example requires us to capture two events. One event is a scroll, after which the program must be able to redraw the map at the new location. The other event is a mouse click on the map, so that we can define the path we will use for the distance calculation. Google's API allows several ways of capturing events, but we will hook our functions in the following way:

```
Gevent.addListener➥
(map, 'click', function➥
(overlay, point) {
  map.recenterOrPanToLatLng➥
(point);
});
```

When using the *addListener()* method from the *GEvent* class, we must indicate the event we wish to capture and the function that will handle it. In this case, we are capturing the mouse click on the map (*click* event). The specific function for this event can receive two parameters: the *overlay* or mark where we have clicked and a point containing the exact coordinates of the click. At Line 25 we capture the *movend* event, which will be produced each time we finish a scroll on the map. The important event is captured at Line 35 of Listing 2. Adding our own mouse click manager, we will enable the user to place several marks, which will be stored in an array. If we click again on one of these marks, we must remove it both from the map and our array. In this simple way, we will be defining a path that we can trace with a blue line. Our distance calculation will be based on that path.

Both marks and lines are referenced as *overlays* in the official documents of Google Maps. Each overlay is an object superimposed on the map. Although we can define custom icons for the marks (again refer to the documents), we can also directly use the standard set provided by the API. A mark is created and positioned on the map as follows:

```
var m = new Gmarker➥
(new GPoint(lon1, lat1));
map.addOverlay(m);
```

We must create a mark located at some specific coordinates. Next we will add it to the map using the *addOverlay()* method of the *GMap* class. This is the same method we will use to draw the

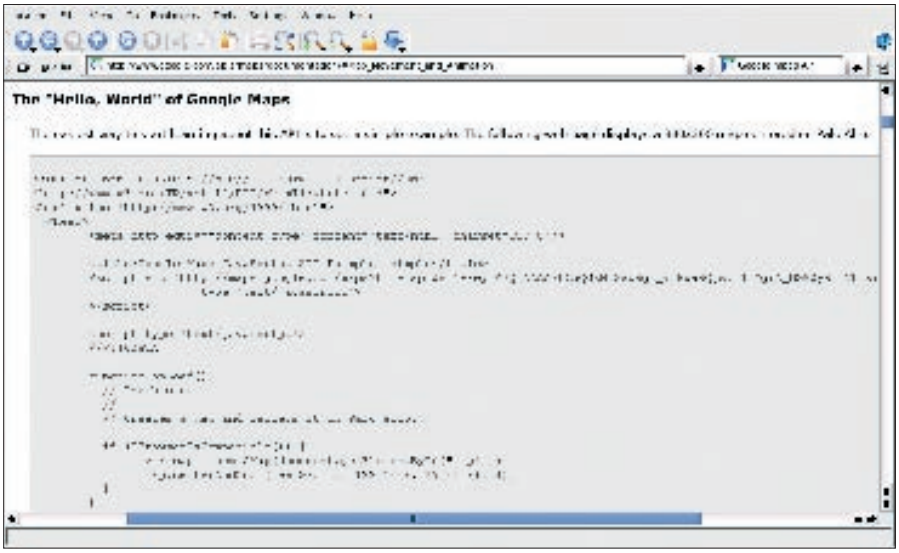

Figure 6: The Google Maps website provides several example scripts.

defined path, but instead of adding a *GMarker* object, we will use a *GPolyline*:

```
var p = new Array;
p.push(new GPoint(lon1, lat1));
p.push(new GPoint(lon2, lat2));
...
map.addOverlay➥
(new GPolyline(p));
```

This is just what the functions *addOverlay()* and *drawLine()* in our code do. An example of a path drawn in this way is the path shown in Figure 5. Once the path (or a part of it) is finished, we will be able to calculate distances using the aforementioned formula.

The Jury

Using no more than four objects and eight different methods, we have created a program capable of calculating the length of a path drawn interactively on a map by the user from a browser. The Google Maps API makes it easy to generate innovative and interesting applications that would otherwise require extensive programming and sound knowledge of mathematics and navigation.

The API supplies another group of objects for asynchronous XML data access through JavaScript (AJAX [6]). This set of objects enables us to store big amounts of terrestrial coordinates in a database and paint them efficiently from the client's browser. The users of this API have started to develop interesting applications [8], [9], [10].

You will also find objects that allow the generation of small signs, which are displayed when a click (or any other predefined event) is produced. These signs are very useful for associating comments with the map, such as a note about a monument or an explanation of a confusing intersection.

It is worth mentioning that the Google Maps API is still at beta stage; that is, it is subject to change even while you develop applications based on it. You can monitor the evolution of these changes, the addition of new features, and the exchange of applications and experiences from the Google Maps Discussion Group [7]. Please do not forget to release any applications you develop to the group so that we can all use them as a source of inspiration. ■

INFO

[1] ECMA-262 *http://www.ecma-international.org/publications/standards/Ecma-262.htm*

[2] Terms of use *http://www.google.com/apis/maps/terms.html*

[3] Google Maps API *http://www.google.com/apis/maps/*

[4] API Documents *http://www.google.com/apis/maps/documentation/*

[5] Distances *http://www.meridianworlddata.com/Distance-Calculation.asp*

[6] AJAX *http://en.wikipedia.org/wiki/AJAX*

[7] Google Maps Discussion Group *http://groups-beta.google.com/group/Google-Maps-API*

[8] Monuments in Paris *http://www.kahunablog.de/gmaps.php?map=paris*

[9] WikiMap *http://www.wikyblog.com/Map/Guest/Home*

[10] Traffic in the UK *http://www.gtraffic.info/*

Listing 2: distance.html

```
01  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
02  <html xmlns="http://www.w3.org/1999/xhtml" xmlns:v="urn:schemas-microsoft-com:vml">
03    <head>
04      <title>Calculating distances</title>
05      <style type="text/css">
06      v\:* {
07        behavior: url(#default#VML);
08      }
09      </style>
10      <script src="http://maps.google.com/maps?file=api&v=1&key=XXXXX" type="text/javascript"></script>
11      <script type="text/javascript">
12      //<![CDATA[
14      // Points of the path (GMaker)
15      var points = new Array;
16      // Last drawn line
17      var polyLine;
19      function onLoad() {
20        if (GBrowserIsCompatible()) {
21          var map = new GMap(document.getElementById("map"), [G_SATELLITE_TYPE]);
22          map.addControl(new GSmallMapControl());
23          map.addControl(new GScaleControl());
25          GEvent.addListener(map, 'moveend', function() {
26            var center = map.getCenterLatLng();
27            var latLngStr = '(' + center.y + ', ' + center.x + ')';
28            document.getElementById("latlong").innerHTML = latLngStr;
29          });
31          GEvent.addListener(map, 'click', function(overlay, point) {
32            if (overlay) {
33              removeOverlay(map, points, overlay);
34            } else if (point) {
35              addOverlay(map, points, new GMarker(point));
36            }
38            polyLine = drawLine(map, points, polyLine);
40            var distance = calcDistance(points);
41            document.getElementById("distance").innerHTML = distance + " Km"; 42. });
43          map.centerAndZoom(new GPoint(-4.48333, 36.66667), 4);
44        }
46        function drawLine(map, points, lastLine) {
47          var p = new Array();
48          for (var i = 0; i < points.length; i++) {
49            p.push(new GPoint(points[i].point.x, points[i].point.y));
50          }
51          var newLine = new GPolyline(p);
53          if (lastLine) {
54            map.removeOverlay(lastLine);
55          }
56          map.addOverlay(newLine);
58          return newLine;
59        }
61        function addOverlay(map, points, overlay) {
62          map.addOverlay(overlay);
63          points.push(overlay);
64        }
66        function removeOverlay(map, points, overlay) {
67          map.removeOverlay(overlay);
68          var oi = -1;
69          for (var i = 0; i < points.length; i++) {
70            if (points[i] == overlay) {
71              oi = i;
72              break;
73            }
74          }
75          points.splice(oi, 1);
76        }
78        function calcDistance(points) {
79          var distance = 0.0;
80          var p1 = points[0];
81          for (var i = 1; i < points.length; i++) {
82            var p2 = points[i];
83            var lat1 = p1.point.y * Math.PI / 180.0;
84            var lon1 = p1.point.x * Math.PI / 180.0;
85            var lat2 = p2.point.y * Math.PI / 180.0;
86            var lon2 = p2.point.x * Math.PI / 180.0;
87            distance += 6378.7 * Math.acos(Math.sin(lat1) * Math.sin(lat2) + Math.cos(lat1) * Math.cos(lat2) * Math.cos(lon2 - lon1));
88            p1 = p2;
89          }
91          return distance;
92        }
93      }
95      //]]>
96      </script>
98    <body onload="onLoad()">
99      <div id="map" style="width: 500px; height: 500px"></div>
100     <div id="latlong"></div>
101     <div id="distance"></div>
102   </body>
103 </html>
```