10
Maps in Action

In this chapter we will cover the following topics:

- Connecting a Twitter feed to a Google map
- Building an advanced interactive marker
- Adding multiple tweets into an InfoWindow bubble
- Customizing the look and feel of markers
- Final project: building a live itinerary

Introduction

In this chapter on mapping, we will tie in more deeply to our topic of data visualization. One of the most popular ways to visualize data these days is by using maps. In this chapter, we will explore a few ideas on how to integrate data into maps, using the Google Maps platform.

Connecting a Twitter feed to a Google map

This is the start of a very fun experiment with Google Maps. The goal of the task is to create a link between Twitter posts and a Google map. It will take us a few recipes to get to our final goal. By the end of this recipe, we will have a Google map. This Google map will be clickable in any area of the screen. When the user clicks on the map, they will connect to the Twitter API and search for tweets in that area that have the word "HTML5" in them. When the result comes back, it will pop a new marker onto the area that was clicked and add the most recent tweet on that topic originating from that location. At this stage, it would just be a marker with a rollover that shows us the actual tweet without more information.

Getting ready

If you haven't read through *Chapter 9*, *Using Google Maps*, you might find this chapter a little difficult, so I encourage you to read it before starting with this recipe. At this stage you should have a Google API set up (see the *Obtaining a Google API key* recipe in *Chapter 9*).

How to do it...

We will create new HTML and JavaScript files and call them `10.01.socielmap.html` and `10.01.socielmap.js`, respectively, and then perform the following steps:

1. Add the following code in the HTML file using your own API key:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Google Maps Markers and Events</title>
    <meta charset="utf-8" />
```

```
    <meta name="viewport" content="initial-scale=1.0, user-
scalable=no" />
    <style>
      html { height: 100% }
      body { height: 100%; margin: 0; padding: 0 }
      #map { height: 100%; width:100%; position:absolute; top:0px;
left:0px }
    </style>
  <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/
jquery.min.js"></script>
    <script src="http://maps.googleapis.com/maps/api/js?key=AIzaSy
AywwIFJPo67Yd4vZgPz4EUSVu10BLHroE&sensor=true">
    </script>
    <script src="./10.01.socielmap.js"></script>
  </head>
  <body onload="init();">
    <div id="map"></div>
  </body>
</html>
```

2. Let's move into the JavaScript file. As we have an `init()` function being called when the `onload` event triggers, we will place all of our code within a new `init` function.

```
function init(){
 //all code here
}
```

3. We will start by setting our center point of the map. So far, we were extremely focused on my home state, New York, so let's change our focus to Europe.

```
var BASE_CENTER = new google.maps.LatL
ng(48.516817734860105,13.005318750000015 );
```

4. Next, let's create a black-and-white style for our map so it's easier to focus on the markers we are about to create.

```
var aGray =  [
    {
      stylers: [{saturation: -100}]
    }
  ];

  var grayStyle = new google.maps.StyledMapType(aGray,{name:
"Black & White"});
```

5. Create a new Google map.

```
var map = new google.maps.Map(document.getElementById("map"),{
    center: BASE_CENTER,
    zoom: 6,
```

```
    mapTypeId: google.maps.MapTypeId.ROADMAP,
    disableDefaultUI: true,

  });
```

6. Set up the `grayStyle` styling object to be our default style.

```
map.mapTypes.set('grayStyle', grayStyle);
map.setMapTypeId('grayStyle');
```

7. Our next step is going to use Google's API to create a new `click` event for the map. When the map is clicked we want to trigger a `listener` function. When a click happens, we want to start our Twitter search as we will connect to the Twitter API and search for the submission of the keyword `html5` within a 50 kilometer radius from where our click on the map was. Let's create a new mouse event and start up the Twitter search.

```
google.maps.event.addListener(map, 'click', function(e) {
    //console.log(e.latLng);
      var searchKeyWord = 'html5';
      var geocode=e.latLng.lat() + "," + e.latLng.lng()+",50km";
      var searchLink = 'http://search.twitter.com/search.json?q='+
searchKeyWord+ '&geocode=' + geocode +"&result_type=recent&rpp=1";


      $.getJSON(searchLink, function(data) {
        showTweet(data.results[0],e.latLng);
    });

    });
```

8. When the Twitter search value is returned, it is time to show our new tweet; if no tweet is found, we will put in default content letting the user know that nothing could be found.

```
function showTweet(obj,latLng){
      if(!obj) obj = {text:'No tweet found in this area for this
topic'};
      console.log(obj);

      var marker = new google.maps.Marker({
            map: map,
            position: latLng,
            title:obj.text    });


  }
```

When you load the map again, you will find a map of Europe waiting to be clicked on. Each click will trigger a new Twitter search and will generate a new result based on the location you clicked on. To read the tweet, roll over the marker after it returns.

How it works...

We live in an age when data about almost anything increasingly overlaps with geolocation data and maps. It's almost impossible to write a book about data without talking about maps, and it's not possible to write a book about data visualization without at least opening the Pandora's box of the world of mapping and its possibilities, either.

Twitter lately has been trying more and more to capture the location of users. For most of the time, the location is still void. That having been said, Twitter always knows the base location of users based on their information, and more so when users tweet through their cell phones. As such, Twitter always has a rough idea of where users are when they send a message, and in the coming years this accuracy is expected to only get better. In the future, more and more Twitter results will have such an accurate location for users that we will be able to pinpoint them directly on a map.

After creating the map, the first step is that, as soon as a user clicks on any area of the map, we start building out a search query to be used on the Twitter API:

```
google.maps.event.addListener(map, 'click', function(e) {
    //console.log(e.latLng);
      var searchKeyWord = 'html5';
      var geocode=e.latLng.lat() + "," + e.latLng.lng()+",50km";
      var searchLink = 'http://search.twitter.com/search.json?q='+
searchKeyWord+ '&geocode=' + geocode +"&result_type=recent&rpp=1";
```

We don't cover all the possibilities with search, but instead we are focusing on two main points: the search query, in our case HTML5, and the location of the query. We get the location information directly from the event that is passed into a marker. We reformat the information from our Google returned event and format it into a string, adding to it the range; in our case we set this to 50 kilometers (you can choose `ml` for miles as well). As we are taking a look at the map of Europe now, I thought it would be appropriate to work in kilometers and not miles.

We want to get our search values back as **JavaScript Object Notation** (**JSON**) values. JSON is a very minimal shorthand way to pass object information as strings between servers. For the most part, you will usually work with automatic converters so you will be sending objects and getting objects, but under the hood there is a JSON encoder and decoder that will process the request.

If you don't know what JSON is, don't worry about it; it's all done in the background and it's not critical to understand how JSON works in order to work with it.

We want to get our data in JSON format; to do that we will send our URL parameters to the following URL:

```
http://search.twitter.com/search.json
```

Append to it our `q` values and `geocode` values. If you want to explore options and possibilities with the Twitter search API more deeply, visit the following page:

```
https://dev.twitter.com/docs/api/1/get/search
```

The next step is to send our information to this service and get our results back. To do that, we will use the `$.getJSON` function in jQuery. This function will take care of all our needs: sending our request, getting it back, and then decoding the information into a regular JavaScript object.

```
$.getJSON(searchLink, function(data) {
        showTweet(data.results[0],e.latLng);
    });
```

The two parameters we need to send in are the search link and the return function. In our case, we will grab our data and send it to an external function, `showTweet`. We will send only the first result from the data to return and the `e.latLng` object information we got from our click event.

Time to create the marker. In the `showTweet` function, our first task will be to check whether there is actually any data in the returned first element. If there is no value it means Twitter didn't find anything.

```
if(!obj) obj = {text:'No tweet found in this area for this topic'};
```

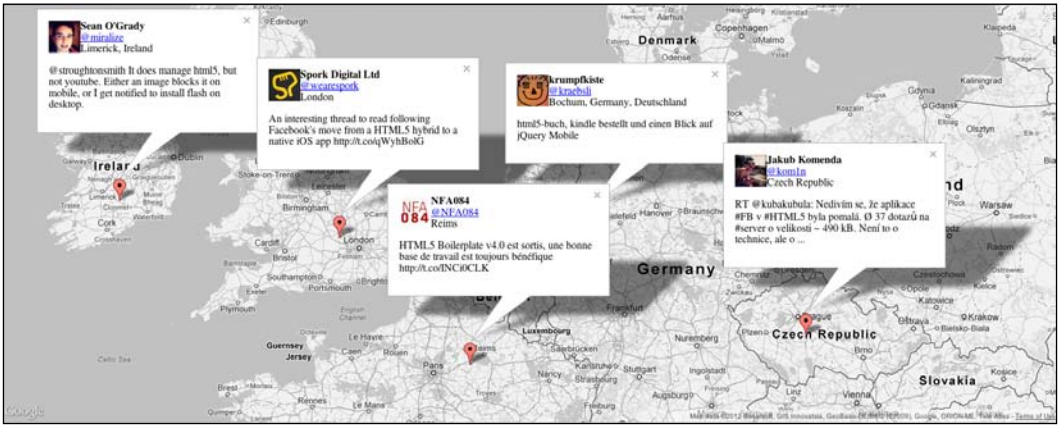
If no object was returned, we will create a new object with placeholder information to replace the regular result information. This is a great way to avoid complexities in your code: by building the exceptions into the regular user experience. We are done; all that is left is for us to create the marker.

```
var marker = new google.maps.Marker({
            map: map,
            position: latLng,
            title:obj.text    });
```

Although we got what we had our minds set on, the `latLng` information we have is for our click and not the exact location of our tweet. Currently there is a property called geo that is returned in the object returned from Twitter. At the time of writing this book, it always comes back empty. Currently it looks like it's a feature that is about to be released or is partially implemented, so by the time you read this book try to check and see if there is a value being returned from the `obj.geo` property and use it to make your point more accurate when this information is available.

Building an advanced interactive marker

The next step in our social map project is to add more details for our Twitter search result. We would like to open up an information panel automatically when the Twitter result comes in. In the process, we will create a subclass of Google Marker and extend it and add a new InfoWindow to enable us to add live HTML data right into our map.



Getting ready

It will be really hard to join in, if you haven't started from the start of this chapter. As this recipe is in continuation of the previous recipe, we will not create a new HTML file or a new JavaScript file but will instead continue from where we left off.

How to do it...

Grab your latest JavaScript file and let's continue to the next steps:

1. In the function `showTweet`, replace the new marker with a new `TwitterMarker` marker.

```
function showTweet(obj,latLng){
      if(!obj) obj = {text:'No tweet found in this area for this
topic'};
      console.log(obj);

      var marker = new TwitterMarker({
            map: map,
            position: latLng,
            tweet: obj,
            title:obj.text    });

  }
```

2. Now that we are not using the regular built-in marker, it's time for us to create our own marker. Let's start with the constructor.

```
function TwitterMarker(opt){
  var strTweet = this.buildTwitterHTML(opt.tweet);
  this.infoWindow = new google.maps.InfoWindow({
      maxWidth:300,
      content:strTweet
  });


  this.setValues(opt);
  this.infoWindow.open(this.map,this);
  google.maps.event.addListener(this, 'click', this.
onMarkerClick);
}
```

3. We will want to extend our new object from the `google.maps.Marker` marker so we can have all the features of the regular marker.

```
TwitterMarker.prototype = new google.maps.Marker();
```

4. Let's create a toggle button in our marker event listener. When the event is called, it will open or close our InfoWindow:

```
TwitterMarker.prototype.onMarkerClick=function(evt){
  this.isOpen=!this.isOpen;
  if(this.isOpen)
    this.infoWindow.close();
  else
    this.infoWindow.open(this.map,this);

}
```

5. It is time to create the Twitter message by creating an HTML string.

```
TwitterMarker.prototype.buildTwitterHTML = function(twt){
  var str;
  if(twt.from_user_name){
   str =       "<span><img style='float: left' src='"+twt.profile_
image_url+"' />"+
         "<b>" +twt.from_user_name + "</b><br/><a href ='http://
twitter.com/"
        + twt.from_user + "'>@"+twt.from_user+"</a><br/> "
        + twt.location + "</span>"
        + "<p>"+twt.text+"</p>";
  }else{
```

```
        str="The 50 Kilometer radius around this point did not message
    this value";
      }
      return str;
    }
```

If you reload your HTML file, you will find it interactive in any area of the world; if it can find a tweet, it will output it on the map within an InfoWindow.

How it works...

Although there are not many lines, a lot of logic is condensed into this code. Let's start by looking at our new marker. This is the first time in this book that we use inheritance. Inheritance, as its name implies, enables us to extend the features of an object in JavaScript without affecting the original object. In our case, we want to take all the features of the marker (methods, properties, and so on) and add to them some custom behaviors.

The inheritance is done in JavaScript by defining a prototype. Until now, we have used the prototype without talking about it much, but we used it mainly to create new methods. If we assign a full object to the prototype, all of the properties and methods of that object will be copied into our new object as well.

```
TwitterMarker.prototype = new google.maps.Marker();
```

Always start first with extending of the object you want to extend, before any other additions. This is because if you've placed any new prototype methods before this line of code, they will silently be deleted and thus will not work.

The `buildTwitterHTML` method takes in the Twitter object returned and converts some of its data into HTML. We use this method once per marker. When we create a new marker, we create a new `InfoWindow` object as well. We place an InfoWindow on top of the marker and showcase the tweet information.

```
function TwitterMarker(opt){
 var strTweet = this.buildTwitterHTML(opt.tweet)
 this.infoWindow = new google.maps.InfoWindow({
     maxWidth:300,
     content:strTweet
 });
```

We are setting the width as well, to avoid having a really big panel. We send our newly created `strTweet` string into the `infoWindow` object.

We wanted our marker to be a toggle button that controls the InfoWindow status. To do that we add a newly dynamically created property called `isOpen`. Even though we open up the InfoWindow earlier in the constructor, we don't set the value of `isOpen` there. We can fix that issue in the first action we perform in our click event listener of the marker.

```
TwitterMarker.prototype.onMarkerClick=function(evt){
  this.isOpen=!this.isOpen;
```

When the marker is clicked, we automatically change the state of the `isOpen` variable. As it was not set before, it will now be set to `true`. The `!` operator is a `boolean` operator that switches a value between `true` and `false`. Its actual meaning is `not`. In other words, we are saying:

```
this.isOpen = is not this.isOpen
```

The interesting fact here is that undefined (the value of a variable that wasn't defined) for the not (`!`) operator is the same as `false`, `null`, or even `0`. Any other value would be considered true. This way, each time the marker is clicked, the value of the variable `this.isOpen` switches. And *that* is the heart of the logic of our toggle button. All that is left is to decide whether to open or close the InfoWindow.

```
  if(this.isOpen)
    this.infoWindow.close();
  else
    this.infoWindow.open(this.map,this);
}
```

This leads us to our final step of figuring out what will be our Twitter text. We will edit this method in the next few recipes. You can play around with it and ideally personalize it to what you prefer it to look like. We have two possible outcomes: *no Twitter messages in the search area* and *Twitter messages in the search area*. If there is a message, we will use some of the returned object properties to build up an HTML outline that will be used inside the `infoWindow` object that is associated with this marker. If there isn't a result, we will create one.

```
TwitterMarker.prototype.buildTwitterHTML = function(twt){
  var str;
  if(twt.from_user_name){
   //build custom message
   /*notice we are validating based on checking if the twitter has a
property (any of the properties would work) */
  }else{
    //the error message
    str="The 50 Kilometer radius around this point did not message
this value";
  }
  return str;
}
```

There you go! Our social map is starting to be much more interesting. It still is missing some features. It would be really nice if we could see more than one message in the InfoWindow result. In the next recipe we will try to solve that issue.

Adding multiple tweets into an InfoWindow bubble

So far, in our interactive social map, we added markers in each location that we clicked on and opened up an InfoWindow with the tweet information. Our next step will be to enable multiple tweets to live inside our InfoWindow by adding a pagination system into our window.

Getting ready

To get through this recipe you must be knee deep into our holistic chapter. If you dropped in just now, it would be a good idea to go back to the start of this chapter as we are going to continue from where we left off in the previous recipe.

How to do it...

We are still in our JavaScript file and we will continue to add code and adjust our code to get multiple Twitter posts into our social map.

1. Let's start by changing our Twitter search to return up to 100 values per search. We are doing this because there is a limit on how many times we can call the Twitter API. So, we will try to grab as much as we can in one hit (this code should be around line 30).

```
var searchLink = 'http://search.twitter.com/search.
json?q='+ searchKeyWord+ '&geocode=' + geocode +"&result_
type=recent&rpp=100";
```

2. As we are now going to treat all the tweets that come back, we will need to change our references to send to our `TwitterMaker` marker the full array (changes highlighted in the code snippet).

```
  google.maps.event.addListener(map, 'click', function(e) {
    //console.log(e.latLng);
      var searchKeyWord = 'html5';
      var geocode=e.latLng.lat() + "," + e.latLng.lng()+",50km";
      var searchLink = 'http://search.twitter.com/search.
json?q='+ searchKeyWord+ '&geocode=' + geocode +"&result_
type=recent&rpp=100";


      $.getJSON(searchLink, function(data) {
        showTweet(data.results,e.latLng);
    });

    });


  function showTweet(a,latLng){
      if(!a) a = [{text:'No tweet found in this area for this
topic'}];
      //console.log(obj);

      var marker = new TwitterMarker({
            map: map,
            position: latLng,
            tweet: a,
            title:a[0].text    });

  }
}
```

3. We want to update the `TwitterMarker` constructor to include our array and quick information on it, such as the total tweets and the current tweet we are in. We will need a way to identify our object, and as such, we will give it an ID as well (more on that in a few steps).

```
function TwitterMarker(opt){
  this.count = opt.tweet.length;
  this.crnt = 0;
  this.id = TwitterMarker.aMarkers.push(this);
  this.aTweets = opt.tweet;
  var strTweet = this.buildTwitterHTML(opt.tweet[0])
  this.infoWindow = new google.maps.InfoWindow({
      maxWidth:300,
      content:strTweet
  });


  this.setValues(opt);
  this.infoWindow.open(this.map,this);
  google.maps.event.addListener(this, 'click', this.
onMarkerClick);
}
```

4. We want to store, in a static array that can be accessed from any place in our code, all the markers created. To do that, we will add a new status array to our `TwitterMarker` class:

```
TwitterMarker.prototype = new google.maps.Marker();
TwitterMarker.aMarkers= [];
```

5. In the `buildTwitterHTML` method, we want to add in back/next links that will be visible to users from InfoWindow:

```
TwitterMarker.prototype.buildTwitterHTML = function(twt){
  var str;
  if(twt.from_user_name){
   str =       "<span><img style='float: left' src='"+twt.profile_
image_url+"' />"+
         "<b>" +twt.from_user_name + "</b><br/><a href ='http://
twitter.com/"
        + twt.from_user + "'>@"+twt.from_user+"</a><br/> "
        + twt.location + "</span>"
        + "<p>"+twt.text+"</p>";

    if(this.count>1){
      str+="<span style='absolute; bottom: 0;
          right: 0px; width:80px'>";
```

```
      if(this.crnt!=0) str+="<a href='javascript:TwitterMarker.
aMarkers["+(this.id-1)+"].prev();'>&lt;</a> ";
      str+= (this.crnt+1) + " of " + this.count;
      if(this.crnt<(this.count-1)) str+= "<a
href='javascript:TwitterMarker.aMarkers["+(this.id-1)+"].
next();'>&gt;</a> ";
      str+= "</span>"
    }
  }else{
    str="The 50 Kilometer radius around this point did not message
this value";
  }
  return str;
}
```

6. Let's now add the `next` and `prev` methods.

```
TwitterMarker.prototype.next =function(){
  this.infoWindow.close();
  this.infoWindow.content = this.buildTwitterHTML(this.
aTweets[++this.crnt]);
  this.infoWindow.open(this.map,this);
  return false;
}

TwitterMarker.prototype.prev =function(){
  this.infoWindow.close();
  this.infoWindow.content = this.buildTwitterHTML(this.aTweets[--
this.crnt]);
  this.infoWindow.open(this.map,this);
  return false;
}
```

Load the HTML file, and you should find a working InfoWindow that can accommodate up to 100 tweets per click.

How it works...

Our first change was to change the number of results coming back from the Twitter search API. This change forced us to change the references in our code from referring directly to the first object returned to focus on the full results object and sending it to our `TwitterMarker` constructor. This change created a few smaller changes in the flow of the information within the constructor as well.

Our goals are to create two buttons that will update our InfoWindow. This is an issue as we need a two-way connection between our marker and its InfoWindow. Until now, all our communication with the InfoWindow was one way. The easiest way for us to solve this problem and bypass the Google interface is to create a static array that will store all markers and refer to our static marker when we trigger buttons inside the `InfoWindow` object. All we need to do is add a variable direction to our class name.

```
TwitterMarker.aMarkers= [];
```

By adding our variable directly into the `TwitterMarker` class, we can now refer to it directly at any point and it will not get duplicated in our objects (as it's not part of the prototype). Now that we have an array, it's time for us to go back into our `TwitterMarker` constructor and send a new reference to this array each time we create a new `TwitterMarker` object. Another benefit we get out of doing this is that we automatically get a unique identifier (ID) for each marker as the returned number will always be a unique number for our needs.

```
this.id = TwitterMarker.aMarkers.push(this);
```

In this one line of code, we perform all of the tasks we talked about in the previous paragraph. The array `push` method returns the new length of the array.

Now that we have a way to refer to our marker and have got an identifier, it's time for us to go back into the `buildTwitterHTML` method and add into the rendered HTML two `href` buttons that will trigger the right marker when the next/previous selections are clicked.

Before we delve into that, we want to check and validate that we have more than one Twitter message that came back; if there is none, there is no point in adding the new logic, and we would be introducing a bug if we had next/previous logic for an item that has only one item.

```
if(this.count>1){

}
```

By the following `if` statement, we figure out whether we are currently in the first Twitter message, and if not we shall add the back button:

```
if(this.crnt!=0) str+="<a href='javascript:TwitterMarker.
aMarkers["+(this.id-1)+"].prev();'>&lt;</a> ";
```

This might look like a huge mess but, if we ignore the HTML and focus on the actual JavaScript that will be triggered when the button is pressed, this is what we will get:

```
TwitterMarker.aMarkers["+(this.id-1)+"].prev();
```

The `this.id-1` parameter will be replaced with the actual current number:

As this is rendered into a string to be parsed as HTML, the value that will be integrated into the HTML will be hardcoded. Let's see this in a real case to make it clear. The first array ID would be `0`, and as such the `prev` button would look like this code statement:

```
TwitterMarker.aMarkers[0].prev();
```

Now the logic is starting to reveal itself. By grabbing the marker from the array that is our current element, all that is left for us to do is trigger the `prev` method and let it take over.

The same logic happens for our other end. The only condition is that we are not in the last Twitter result and if not we call the `next` method:

```
if(this.crnt<(this.count-1)) str+= "<a href='javascript:TwitterMarker.
aMarkers["+(this.id-1)+"].next();'>&gt;</a> ";
```

There you have it! The core of our logic is in place.

If we wanted we could have created our InfoWindow by wrapping a `<div>` tag with a unique ID and just called it and made direct updates to our content (try doing that by yourself as that would be a better solution). Instead, we are working with the limitations of the InfoWindow. As we cannot update the full bucket container while it's open, we need to close it to update it and then open it again. Thus our logic in both the `next` and `prev` methods is similar; both have a limitation on the change of the actual value that is being rendered.

```
TwitterMarker.prototype.next =function(){
  this.infoWindow.close();
  this.infoWindow.content = this.buildTwitterHTML(this.aTweets[++this.
crnt]);
  this.infoWindow.open(this.map,this);
  return false;
}

TwitterMarker.prototype.prev =function(){
  this.infoWindow.close();
  this.infoWindow.content = this.buildTwitterHTML(this.aTweets[--this.
crnt]);
  this.infoWindow.open(this.map,this);
  return false;
}
```

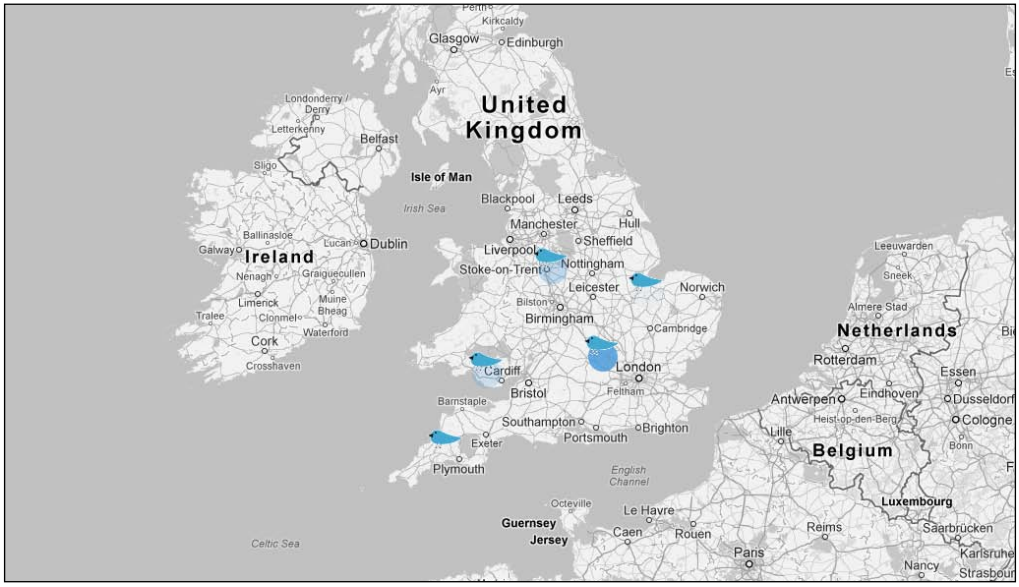
All the logic is the same and limited to the highlighted code snippet. If you aren't familiar with this shortcut, the `++` and `--` operators when set before a variable, enable us to add/subtract 1 from it and update it before its value is sent on. Thus in one line, we can both change the number in the variable and send that newly created number to continue its tasks.

In the case of the `next` method, we want to grab the next tweet, while for the `prev` method, we want to grab the previous tweet.

Customizing the look and feel of markers

This will be our last recipe for social mapping. In this recipe, we will revisit our marker itself and give it a facelift. As our marker represents Twitter messages in a clicked area, we will update our marker to look like a Twitter bird (hand made). We will not stop there; after updating our graphic, we will add another graphical layer to shadow our Twitter marker. It will be a shadow, and its opacity will range from zero to full, depending on the number of tweets (a maximum of hundred tweets).

The best way to understand our goal is by checking out the following screenshot:



Note how some tweets have no visible circle outline, while others have a very dark one (that is based on how many tweets are there).

Getting ready

To complete this task you need to first complete all the previous recipes in this chapter.

How to do it...

We will jump right into the JavaScript file and continue from where we left off in the previous recipe.

1. Update the `showTweet` function.

```
function showTweet(a,latLng){
      if(!a) a = [{text:'No tweet found in this area for this
topic'}];
      //console.log(obj);

      var marker = new TwitterMarker({
            map: map,
            position: latLng,
            tweet: a,
            title:a[0].text,
            icon:"img/bird.png"    });

  }
```

2. Create an instance of the `MarkerCounter` object in the `TweeterMarker` constructor.

```
function TwitterMarker(opt){
  this.count = opt.tweet.length;
  this.mc = new MarkerCounter(opt);
  this.crnt = 0;
...
```

3. Create the `MarkerCounter` constructor.

```
function MarkerCounter(opt) {
    this.radius = 15;
    this.opacity = (opt.tweet.length) /100;
    this.opt = opt;
    this.setMap(opt.map);
  }
```

4. Create subclass, `MarkerCounter`, for the `google.maps.OverlayView` object.

```
MarkerCounter.prototype = new google.maps.OverlayView();
```

5. Create an `onAdd` method. It will be called automatically when an element is added into the map. In this method, we will finish up all the preparatory work for the drawing but won't draw the elements.

```
MarkerCounter.prototype.onAdd = function() {
  var div = document.createElement('div');
    div.style.border = "none";
    div.style.borderWidth = "0px";
```

```
    div.style.position = "absolute";


  this.canvas = document.createElement("CANVAS");
    this.canvas.width = this.radius*2;
  this.canvas.height = this.radius*2;

    this.context = this.canvas.getContext("2d");
  div.appendChild(this.canvas);
  this.div_ = div;


    var panes = this.getPanes();
       panes.overlayLayer.appendChild(div);

 }
```

6. Last but not least, it's time to override the `draw` method and draw into the new canvas element created in the previous step and position the `div` element.

```
MarkerCounter.prototype.draw = function() {
   var radius = this.radius;
   var context = this.context;
 context.clearRect(0,0,radius*2,radius*2);

 context.fillStyle = "rgba(73,154,219,"+this.opacity+")";
 context.beginPath();
   context.arc(radius,radius, radius, 0, Math.PI*2, true);
 context.closePath();
 context.fill();
 var projection = this.getProjection();
 var point = projection.fromLatLngToDivPixel(this.opt.position);

   this.div_.style.left = (point.x - radius) + 'px';
   this.div_.style.top = (point.y - radius) + 'px';

 };
```

When you run the application, you will find that now our markers look like those of Twitter and the larger the number of tweets that originate from a location, the more opaque the egg under our Twitter bird will be.

How it works...

The first step is to swap the graphic that is the default graphic for the marker. As we are extending the regular marker, we have all of its default features and behaviors. One of these features is the ability to swap the icon. To do that, we pass in one of our object parameters as the icon and its path.

```
var marker = new TwitterMarker({
            map: map,
            position: latLng,
            tweet: a,
            title:a[0].text,
            icon:"img/bird.png"    });
```

You might be wondering how this actually works, as we are not actually doing anything to the icon parameter in our code. It's very simple. If you take a deeper look at the `TwitterMaker` constructor, you will find the following line:

```
this.setValues(opt);
```

Passing the `setValues` method to the `opt` object is our way of letting the marker continue and rendering our marker with the information we just got into our constructor. All the things that can be done in a regular marker can be done in ours as well.

At this stage we have our Twitter bird as our graphic interface for our marker. Unfortunately, this is as far as we can go with customizing our marker; next, we will need to add another visual layer. As we want to create a visual layer that behaves like a marker just visually (as it will be part of the marker), we will need to create a subclass for the `google.maps.OverlayView` object.

Similar to the marker logic, when we are ready to render our element, we want to call the method `setMap` (for the marker it was a different method but the same idea).

```
function MarkerCounter(opt) {
    this.radius = 15;
    this.opacity = (opt.tweet.length) /100;
    this.opt = opt;
    this.setMap(opt.map);
  }

  MarkerCounter.prototype = new google.maps.OverlayView();
```

In our constructor, we are only storing very basic global information, such as our target opacity, radius, and the `options` object. We can store any information we want here. The most important element of information that we will need is the position (latitude and longitude). We will send that information into our marker, and it will be inside our `opt` object.

The `google.maps.OverlayView` object has an `onAdd` method. It's just like a listener, but in addition, we will override this method and add our processing/preparation work when the element is added into the map.

```
MarkerCounter.prototype.onAdd = function() {
  var div = document.createElement('div');
    div.style.border = "none";
    div.style.borderWidth = "0px";
    div.style.position = "absolute";


  this.canvas = document.createElement("CANVAS");
    this.canvas.width = this.radius*2;
  this.canvas.height = this.radius*2;

    this.context = this.canvas.getContext("2d");
  div.appendChild(this.canvas);
  this.div_ = div;
    var panes = this.getPanes();
    panes.overlayLayer.appendChild(div);
  }
```

Most of the logic here should look familiar. We start by creating a new `div` element. We set its CSS attributes to make absolute the position of the `div` element so we can move it around easily. We follow this with creating a canvas element and setting its width and height to be two times the radius of our circle. We add the canvas into our `div` element. Last but not least it's time for us to add our `div` element into the map. We will do that by accessing the `getPanes` method. This method will return all the visual layers this element can contain. In our case, we will go right to our overlay layer and add our `div` element to it. We do this inside the `onAdd` method rather than doing it earlier because the overlay will not be rendered and we will not have access to the last two lines in the previous code.

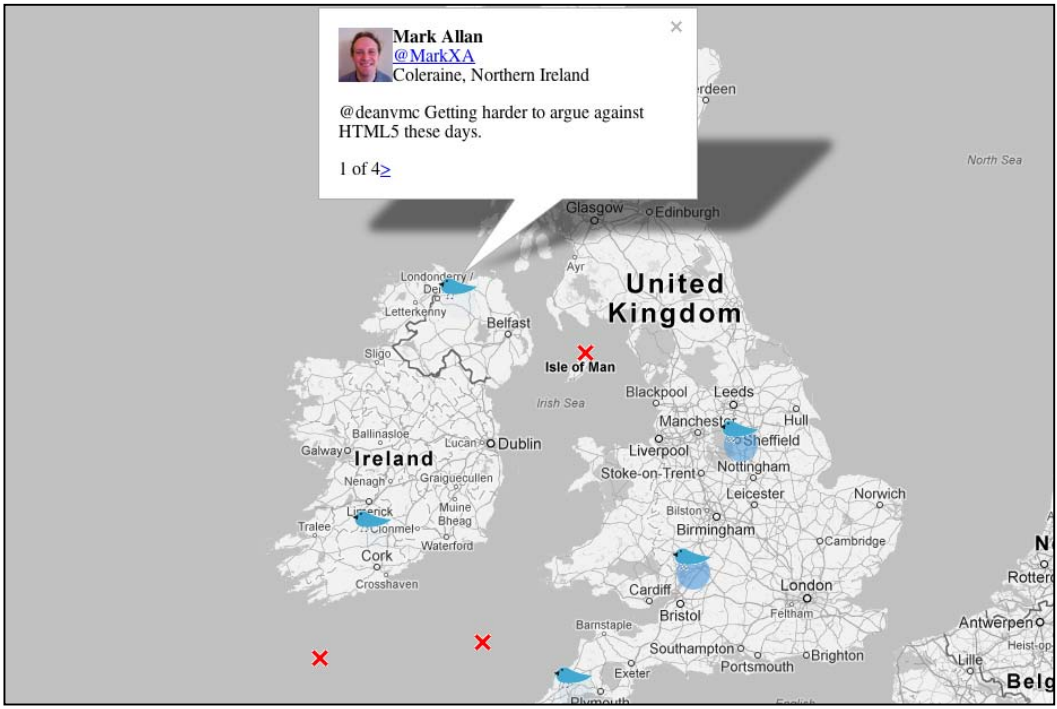
Just as we overrode the `onAdd` method, we do the same for the `draw` method. This is our last critical step. For the most part, all the work in this method will be very familiar as we have played a lot with canvas in this book. So, let's explore the new steps to find where we want to position our overlay.

```
var projection = this.getProjection();
   var point = projection.fromLatLngToDivPixel(this.opt.position);

    this.div_.style.left = (point.x - radius) + 'px';
    this.div_.style.top = (point.y - radius) + 'px';
```

In the first line in the preceding code, we get the projection. The projection is the relative point of view of our overlay. Through this projection, we can extract the actual point in pixels. We call the `projection.fromLatLngToDivPixel` method, send to it a latitude/longitude object, and get back a point (`x`, `y` values). All that is left is to update the style of our `div` element and position it according to this information (not forgetting to subtract our radius size so our element is exactly in the middle of the actual point that was clicked).

Until now, we have treated our `TwitterMarker` constructor as if there are always tweets somewhere in the world, but the reality is that sometimes there will not be anything, and right now we are creating both a visualization that won't work and a marker that won't visualize it. Let's override our behaviors and put up an alternative marker if there is no result and skip all of our customizations.



Let's sort it out. We start by removing our original error logic from the `showTweet` method. Instead, we will just update the `text` attribute but will not create a new array.

```
function showTweet(a,latLng){
      var marker = new TwitterMarker({
            map: map,
            position: latLng,
            tweet: a,
```

```
            title:a.length? a[0].text : 'No tweet found in this area
            for this topic' ,
            icon:"img/bird.png"    });

    }
```

In case you are not familiar with the ternary operator, it's a very condensed way of creating an `if...else` statement within code. The core logic of it is as follows:

```
condition?true outcome:false outcome;
```

The outcome is then sent back, and we can capture it right into our variable as we are doing in this case.

The next area we want to change is the `TwitterMarker` constructor.

```
function TwitterMarker(opt){
  if(!opt.tweet || !opt.tweet.length){
    opt.icon = "img/x.png";
  }else{

    this.count = opt.tweet.length;
    this.mc = new MarkerCounter(opt);
    this.crnt = 0;
    this.id = TwitterMarker.aMarkers.push(this);
    this.aTweets = opt.tweet;
    var strTweet = this.buildTwitterHTML(opt.tweet[0])
    this.infoWindow = new google.maps.InfoWindow({
        maxWidth:300,
        content:strTweet
    });

    this.infoWindow.open(this.map,this);
    google.maps.event.addListener(this, 'click', this.onMarkerClick);

  }
  this.setValues(opt);
}
```

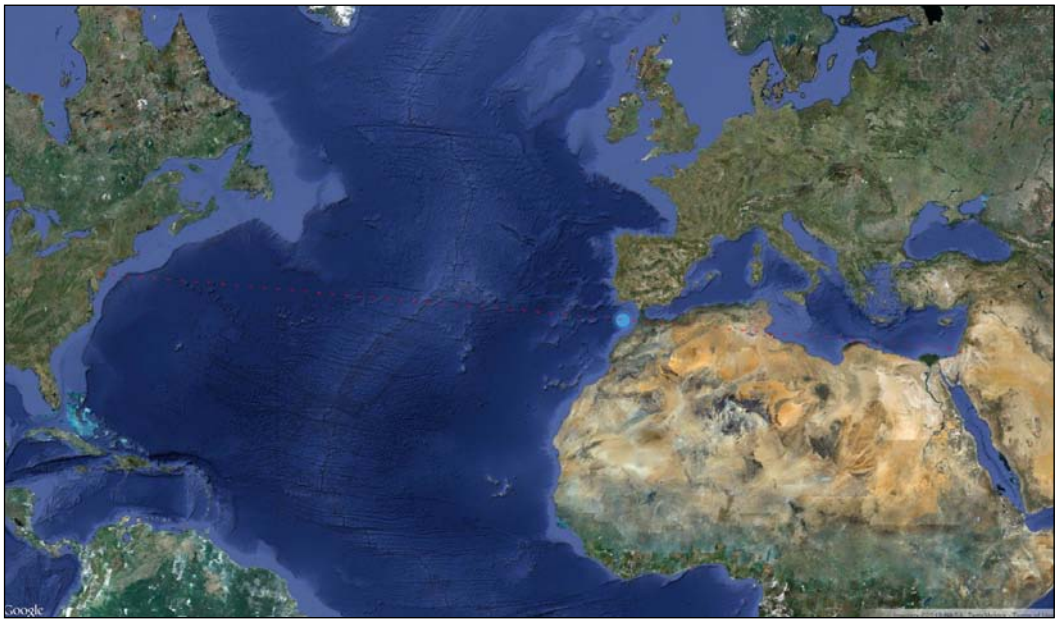
The main changes here are that we start our application by first checking if there are any tweets. If no tweets are around, we update the icon graphic to a new **X** icon. If we do have a result, all remains the same. We extracted the `setValues` method to be called out of the `if...else` conditions as we need to call it in any case.

There you go! We've completed our social map. There is much more you can do with this project. A couple of examples could be making it easier to change the search term, and comparing between two search results (that could be very interesting and easy). I would be interested to see around the world the number of times Flash versus HTML5 are mentioned, so if you get to it send me an e-mail.

Final project: building a live itinerary

Although the natural next step from our previous sample would be just to add an extra feature to our already growing social map (which we have built throughout this chapter), we are taking a direction shift.

In our final recipe, we will build an interactive Google map that will animate with the travel information of a close friend of mine in South America while I was working on this book. To build this application, we will animate the map by adding drawings and moving markers; we will integrate with an external feed of travel information and integrate animations and text snippets that will describe the journey. In the following screenshot, you can see a very small snapshot of the plain path:



Getting ready

Many of the elements we will be working with in this recipe will be based on work we did throughout all of the chapters. As such, it will not be easy to just jump right in if you haven't gone through the journey together with us. There are no prerequisites. We will start from scratch, but we will not focus on things we have learned already.

As the user "travels" around the world map when there is a message for the user in the data source, the map will fade out and the message will be displayed before the user can continue traveling the world:

How to do it...

In this recipe we will be creating two files: an HTML file and a JavaScript file. Let's look into them, starting with the HTML file:

1. Create the HTML file.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Google Maps Markers and Events</title>
    <meta charset="utf-8" />
    <meta name="viewport" content="initial-scale=1.0, user-
scalable=no" />
    <link href='http://fonts.googleapis.com/css?family=Yellowtail'
rel='stylesheet' type='text/css'>
    <style>
      html { height: 100% }
      body { height: 100%; margin: 0; padding: 0 }
      #map { height: 100%; width:100%; position:absolute; top:0px;
left:0px }

      .overlay {
        background: #000000 scroll;
        height: 100%;
        left: 0;
        opacity: 0;
```

```
            position: absolute;
            top: 0;
            width: 100%;
            z-index: 50;
        }
        .overlayBox {
            left: -9999em;
            opacity: 0;
            position: absolute;
            z-index: 51;
            text-align:center;
            font-size:32px;
            color:#ffffff;
            font-family: 'Yellowtail', cursive;
        }
        </style>
      <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/
jquery.min.js"></script>
        <script src="http://maps.googleapis.com/maps/api/js?key=AIzaSy
Bp8gVrtxUC2Ynjwqox7I0dxrqjtCYim-8&sensor=false"></script>
        <script src="https://www.google.com/jsapi"></script>

        <script src="./10.05.travel.js"></script>
      </head>
      <body>
        <div id="map"></div>
      </body>
    </html>
```

2. Time to move to the JavaScript file, `10.05.travel.js`. We will start by initiating the visualization library and storing a global map variable.

```
 google.load('visualization', '1.0');
google.setOnLoadCallback(init);
```

3. var map; When the `init` function is triggered, the map is loaded and it triggers the loading of a Google spreadsheet in which we will store all of my friend's travel information.

```
function init() {
  var BASE_CENTER = new google.maps.LatL
ng(48.516817734860105,13.005318750000015 );

  map = new google.maps.Map(document.getElementById("map"),{
    center: BASE_CENTER,
    mapTypeId: google.maps.MapTypeId.SATELLITE,
    disableDefaultUI: true,

  });
```

```
  var query = new google.visualization.Query(
      'https://spreadsheets.google.com/tq?key=0Aldzs55s0XbDdERJVlY
yWFJISFN3cjlqU1JnTGpOdHc');
    query.send(onTripDataReady);

}
```

4. When the document is loaded, it will trigger the `onTripDataReady` listener. When that happens, we will want to create a new `GoogleMapTraveler` object (a custom class for managing our experience).

```
function onTripDataReady(response){
  var gmt = new GoogleMapTraveler(response.g.D,map);
}
```

5. The constructor method of the `GoogleMapTraveler` object will prepare our variables, create a new `Animator` object, a `Traveler` object and a new `google.maps.Polyline` object, and will trigger the creation of the first travel point by calling the `nextPathPoint` method.

```
function GoogleMapTraveler(aData,map){
  this.latLong; //will be used to store current location
  this.zoomLevel; //to store current zoom level
  this.currentIndex=0;
  this.data = aData; //locations
  this.map = map;

  //this.setPosition(0,2);
  this.animator = new Animator(30);

  this.pathPoints = [this.getPosition(0,1)]; //start with two
points at same place.

  var lineSymbol = {
        path: 'M 0,-1 0,1',
        strokeOpacity: .6,
        scale: 2
      };

    this.lines = new google.maps.Polyline({
        path: this.pathPoints,
        strokeOpacity: 0,
        strokeColor: "#FF0000",
        icons: [{
          icon: lineSymbol,
          offset: '0',
          repeat: '20px'
```

```
        }],
        map: map
      });

  this.traveler = new Traveler(this.map,this.getPosition(0,1));
  this.nextPathPoint(1);

}
```

6. The `getPosition` method is a very smart, small method that enables us to create a new `google.maps.LatLng` object each time it's called and to create a point based on an average of points or based on one item.

```
  GoogleMapTraveler.prototype.getPosition = function
(index,amount){
  var lat=0;
  var lng=0;
  for(var i=0; i<amount; i++){
    lat+= parseFloat(this.data[index+i].c[0].v);
    lng+= parseFloat(this.data[index+i].c[1].v);

  }
  var ll=new google.maps.LatLng(
            lat/amount,
            lng/amount);
  return ll;
}
```

7. We want to be able to set the position of our traveler, and as such we will want to create a `setPosition` method as well.

```
GoogleMapTraveler.prototype.setPosition = function(index,amount){
  this.currentFocus = index;

  var lat=0;
  var lng=0;
  for(var i=0; i<amount; i++){
    lat+= parseFloat(this.data[index+i].c[0].v);
    lng+= parseFloat(this.data[index+i].c[1].v);

  }
  var ll=new google.maps.LatLng(
            lat/amount,
            lng/amount);

  if(this.data[index].c[2])this.map.setZoom(this.
data[index].c[2].v);
  this.map.setCenter(ll);

}
```

8. In the heart of our application is the ability to automatically move from one step to the next. This logic is applied using our `Animator` object in combination with the `nextPathPoint` method:

```
GoogleMapTraveler.prototype.nextPathPoint = function(index){
  this.setPosition(index-1,2);
  this.pathPoints.push(this.getPosition(index-1,1)); //add last
point again
  var currentPoint = this.pathPoints[this.pathPoints.length-1];
  var point = this.getPosition(index,1);

  //console.log(index,currentPoint,point,this.
getPosition(index,1));
  this.animator.add(currentPoint,"Za",currentPoint.Za,point.Za,1);
  this.animator.add(currentPoint,"Ya",currentPoint.Ya,point.Ya,1);
  this.animator.add(this.traveler.ll,"Za",this.traveler.
ll.Za,point.Za,2,0.75);
  this.animator.add(this.traveler.ll,"Ya",this.traveler.
ll.Ya,point.Ya,2,0.75);

  this.animator.onUpdate = this.bind(this,this.renderLine);
  this.animator.onComplete = this.bind(this,this.
  showOverlayCopy);//show copy after getting to destination
}
```

9. There are two callbacks that are triggered through our `Animator` object (they're highlighted in the preceding code snippet). It is time to create the logic that updates the information on our `onUpdate` callback. Let's take a peek at the `renderLine` method.

```
GoogleMapTraveler.prototype.renderLine = function(){
  this.lines.setPath(this.pathPoints);
  if(this.traveler.isReady)this.traveler.refreshPosition();
}
```

10. In the next step, when the animation is complete, it triggers our overlay logic. The overlay logic is very simple; if there is text in the Google document, in the fifth column, we will darken the screen and type it out. If there is no text, we will skip this step and go right to the next step that is in the `hideOverlayCopy` method that triggers the next travel point (the next line in the spreadsheet).

11. Our previous method of the `GoogleMapTraveler` object is the `bind` method. We already created this `bind` method in the *Moving to an OOP perspective* recipe in *Chapter 6*, *Bringing Static Things to Life*; as such, we will not elaborate on it further.

```
GoogleMapTraveler.prototype.bind = function(scope, fun){
   return function () {
        fun.apply(scope, arguments);
    };
}
```

12. Create the `Traveler` class. The `Traveler` class will be based on the work we did in the *Customizing the look and feel of markers* recipe in this chapter, only this time it will be an animating marker.

```
function Traveler(map,ll) {
  this.ll = ll;
    this.radius = 15;
    this.innerRadius = 10;
    this.glowDirection = -1;
    this.setMap(map);
    this.isReady = false;

  }

  Traveler.prototype = new google.maps.OverlayView();

  Traveler.prototype.onAdd = function() {
  this.div = document.createElement("DIV");
  this.canvasBG = document.createElement("CANVAS");
    this.canvasBG.width = this.radius*2;
  this.canvasBG.height = this.radius*2;
  this.canvasFG = document.createElement("CANVAS");
    this.canvasFG.width = this.radius*2;
  this.canvasFG.height = this.radius*2;

  this.div.style.border = "none";
  this.div.style.borderWidth = "0px";
  this.div.style.position = "absolute";

  this.canvasBG.style.position = "absolute";
  this.canvasFG.style.position = "absolute";


  this.div.appendChild(this.canvasBG);
  this.div.appendChild(this.canvasFG);


    this.contextBG = this.canvasBG.getContext("2d");
    this.contextFG = this.canvasFG.getContext("2d");

  var panes = this.getPanes();
    panes.overlayLayer.appendChild(this.div);

  }

  Traveler.prototype.draw = function() {
    var radius = this.radius;
```

```
    var context = this.contextBG;

  context.fillStyle = "rgba(73,154,219,.4)";
  context.beginPath();
    context.arc(radius,radius, radius, 0, Math.PI*2, true);
  context.closePath();
  context.fill();

  context = this.contextFG;
  context.fillStyle = "rgb(73,154,219)";
  context.beginPath();
    context.arc(radius,radius, this.innerRadius, 0, Math.PI*2,
true);
  context.closePath();
  context.fill();

    var projection = this.getProjection();

    this.updatePosition(this.ll);
    this.canvasBG.style.opacity = 1;
    this.glowUpdate(this);
    setInterval(this.glowUpdate,100,this);
    this.isReady = true;

  };

  Traveler.prototype.refreshPosition=function(){
    this.updatePosition(this.ll);
  }

  Traveler.prototype.updatePosition=function(latlng){
    var radius = this.radius;
    var projection = this.getProjection();
  var point = projection.fromLatLngToDivPixel(latlng);
    this.div.style.left = (point.x - radius) + 'px';
    this.div.style.top = (point.y - radius) + 'px';
  }

  Traveler.prototype.glowUpdate=function(scope){ //endless loop
    scope.canvasBG.style.opacity = parseFloat(scope.canvasBG.
style.opacity) + scope.glowDirection*.04;
    if(scope.glowDirection==1 && scope.canvasBG.style.opacity>=1)
scope.glowDirection=-1;
    if(scope.glowDirection==-1 && scope.canvasBG.style.
opacity<=0.1) scope.glowDirection=1;
  }
```

13. We will grab the `Animator` class created in the *Animating independent layers* recipe in *Chapter 6*, *Bringing Static Things to Life*, and tweak it (changes are highlighted in the code snippet).

```
function Animator(refreshRate){
  this.onUpdate = function(){};
  this.onComplete = function(){};
  this.animQue = [];
  this.refreshRate = refreshRate || 35; //if nothing set 35 FPS
  this.interval = 0;
}

Animator.prototype.add = function(obj,property,
from,to,time,delay){
  obj[property] = from;

  this.animQue.push({obj:obj,
            p:property,
            crt:from,
            to:to,
            stepSize: (to-from)/(time*1000/this.refreshRate),
            delay:delay*1000 || 0});

  if(!this.interval){ //only start interval if not running already
    this.interval = setInterval(this._animate,this.
refreshRate,this);
  }

}


Animator.prototype._animate = function(scope){
  var obj;
  var data;

  for(var i=0; i<scope.animQue.length; i++){
      data = scope.animQue[i];

      if(data.delay>0){
        data.delay-=scope.refreshRate;
      }else{
        obj = data.obj;
     if((data.stepSize>0 && data.crt<data.to) ||
        (data.stepSize<0 && data.crt>data.to)){

          data.crt = data.crt + data.stepSize;
          obj[data.p] = data.crt;
```

```
            }else{
              obj[data.p] = data.to;
              scope.animQue.splice(i,1);
              --i;
            }
          }

      }
    scope.onUpdate();
      if(  scope.animQue.length==0){
        clearInterval(scope.interval);
        scope.interval = 0; //reset interval variable
     scope.onComplete();
      }
    }
```

When you load the HTML file, you will find a fullscreen map that is getting its directions from a spreadsheet. It will animate and show you the paths my friend took as he traveled from Israel to South America and back.

How it works...

There are many components in this example, but we will focus mainly on the new steps that we haven't covered in any other part of this book.

The first new thing we meet is right in our HTML and CSS:

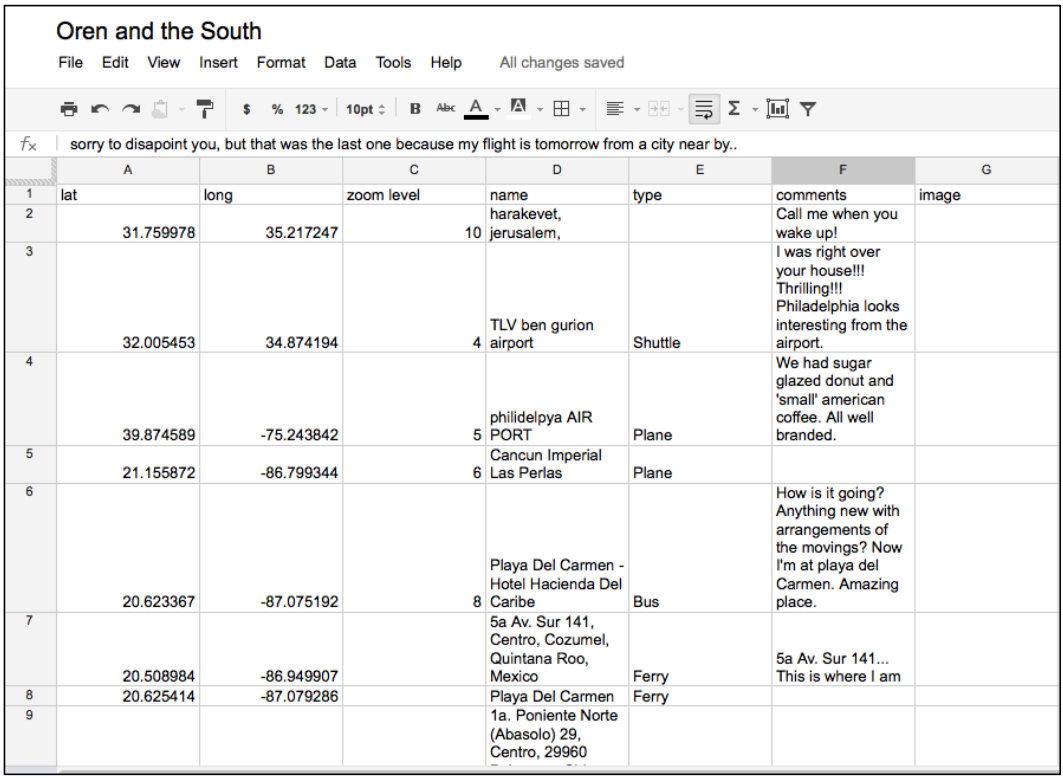
```
<link href='http://fonts.googleapis.com/css?family=Yellowtail'
rel='stylesheet' type='text/css'>
```

We picked a font from the Google font library at `http://www.google.com/webfonts` and integrated it into the text overlays.

```
.overlayBox {
       ...
        font-family: 'Yellowtail', cursive;
    }
```

It is time to travel into our JavaScript file, which we start by loading in the Google Visualization Library. It's the same library we were working with in *Chapter 8*, *Playing with Google Charts*. Once it's loaded, the `init` function is triggered. The `init` function starts our map up and starts loading in the spreadsheet.

In the *Changing data source to Google spreadsheet* recipe in *Chapter 8*, *Playing with Google Charts*, we worked with Google spreadsheets for the first time. There you learned all the steps involved with preparing and adding a Google chart into the Google visualization. In our case, we created a chart that contains line by line all the areas through which my friend traveled.



Oren and the South

File Edit View Insert Format Data Tools Help All changes saved

sorry to disapoint you, but that was the last one because my flight is tomorrow from a city near by..

	A	B	C	D	E	F	G
1	lat	long	zoom level	name	type	comments	image
2	31.759978	35.217247	10	harakevet, jerusalem,		Call me when you wake up!	
3	32.005453	34.874194	4	TLV ben gurion airport	Shuttle	I was right over your house!!! Thrilling!!! Philadelphia looks interesting from the airport.	
4	39.874589	-75.243842	5	philidelpya AIR PORT	Plane	We had sugar glazed donut and 'small' american coffee. All well branded.	
5	21.155872	-86.799344	6	Cancun Imperial Las Perlas	Plane		
6	20.623367	-87.075192	8	Playa Del Carmen - Hotel Hacienda Del Caribe	Bus	How is it going? Anything new with arrangements of the movings? Now I'm at playa del Carmen. Amazing place.	
7	20.508984	-86.949907		5a Av. Sur 141, Centro, Cozumel, Quintana Roo, Mexico	Ferry	5a Av. Sur 141... This is where I am	
8	20.625414	-87.079286		Playa Del Carmen	Ferry		
9				1a. Poniente Norte (Abasolo) 29, Centro, 29960			

The exception in this case is that we don't want to feed our URL into a Google chart, but instead we want to work with it directly. To do that we will use one of Google's API interfaces, the `google.visualization.Query` object:

```
var query = new google.visualization.Query(
      'https://spreadsheets.google.com/tq?key=0Aldzs55s0XbDdERJVlYyWFJ
ISFN3cjlqU1JnTGpOdHc');
    query.send(onTripDataReady);
```

The next step is to create our `GoogleMapTraveler` object. The Google map traveler is a new way for us to work with Google Maps. It doesn't extend any built-in feature of Google maps but is instead a hub for all the other ideas we created in the past. It is used as a manager hub for the marker, called Traveler, that we will create soon and the `google.maps.Polyline` object that enables us to draw lines on the map.

Instead of having a very static line appearance, let's create a reveal effect for new lines that are added to the Google map. To achieve that, we need a way to update the polyline every few milliseconds to create an animation. From the get go, I know the start point and the destination point as I get that information from the Google spreadsheet created earlier.

The idea is very simple even though in a very complex ecosystem. The idea is to have an array that will store all the latitude/longitude points. This would then be fed into the `this.line` object every time we wanted to update our screen.

The heart of the logic in this application is stored within this line of code:

```
this.nextPathPoint(1);
```

It will start a recursive journey throughout all of the points in our chart.

There's more...

Let's take a deeper look at the logic behind the `GoogleMapTraveler.prototype.nextPathPoint` method. The first thing we do in this function is to set our map view.

```
this.setPosition(index-1,2);
```

The `setPosition` method does a few things that are all related to repositioning our map and our zoom level based on the data in the current index that is sent. It's a bit smarter than that as it takes in a second parameter that enables it to average out two points. As one travels between two points, it would be best if our map is at the center of the two points. That is done by sending in `2` as the second parameter. The internal logic of the `setPosition` method is simple. It will loop through as many items as it needs to, to average out the right location.

Next, we add a new point to our `this.pathPoints` array. We start by duplicating the same point that is already in the array, as we want our new second point to start from the starting point. This way, we can update the last value in our array each time, until it reaches the end goal (of the real next point).

```
this.pathPoints.push(this.getPosition(index-1,1)); //add last point
again
```

We create a few helper variables. One will point to the new object we just created and pushed into our `pathPoints` array. And the second is the point that we want to reach at the end of our animation.

```
var currentPoint = this.pathPoints[this.pathPoints.length-1];
var point = this.getPosition(index,1);
```

The first variable is not a new object but a reference to the last point created, and the second line is a new object.

Our next step will be to start and animate the values of our `currentPoint` until it reaches the values in the `point` object and to update our traveler latitude/longitude information until it reaches its destination as well. We give a delay of 0.75 seconds to our second animation to keep things more interesting.

```
this.animator.add(currentPoint,"Za",currentPoint.Za,point.Za,1);
this.animator.add(currentPoint,"Ya",currentPoint.Ya,point.Ya,1);
this.animator.add(this.traveler.ll,"Za",this.traveler.ll.Za,point.
Za,2,0.75);
this.animator.add(this.traveler.ll,"Ya",this.traveler.ll.Ya,point.
Ya,2,0.75);
```

Before we end this method, we want to actually animate our lines. Right now, we are animating two objects that are not visual. To start animating our visual elements, we will listen to updates till the time we complete the animations.

```
this.animator.onUpdate = this.bind(this,this.renderLine);
  this.animator.onComplete = this.bind(this,this.showOverlayCopy);//
show copy after getting to destination
```

Each time the animation happens, we update the values of our visual elements in the `renderLine` method.

To avoid getting runtime errors, we added to the traveler marker an `isReady` Boolean to indicate to us when our element is ready to be drawn into.

```
this.lines.setPath(this.pathPoints);
if(this.traveler.isReady)this.traveler.refreshPosition();
```

When the animation completes, we move to the `showOverlayCopy` method, where we take over the screen and animate the copy in the same strategy as we've done before. This time around, when we are done with this phase, we will trigger our initial function again and start the cycle all over with an updated index.

```
GoogleMapTraveler.prototype.hideOverlayCopy = function(){
  //update index now that we are done with initial element
  this.currentIndex++;
  …

  //as long as the slide is not over go to the next.
  if(this.data.length>this.currentIndex+1)this.nextPathPoint(this.
currentIndex+1);
}
```

That covers the heart of our build. It's time for us to talk briefly about the two other classes that will help create this application.

Understanding the Traveler marker

We will not dig deeply into this class, as for the most part, it's based on the work we did in the previous recipe, *Customizing the look and feel of markers*. The biggest difference is that we added internal animation to our element and an `updatePosition` method that enables us to move our marker around whenever we want to move it.

```
Traveler.prototype.updatePosition=function(latlng){
   var radius = this.radius;
   var projection = this.getProjection();
 var point = projection.fromLatLngToDivPixel(latlng);
   this.div.style.left = (point.x - radius) + 'px';
   this.div.style.top = (point.y - radius) + 'px';
 }
```

This method gets a latitude and longitude and updates the marker's position.

As we are animating the actual `ll` object of this object in the main class, we added a second method, `refreshPosition`, which is called each time the animations are updated.

```
Traveler.prototype.refreshPosition=function(){
    this.updatePosition(this.ll);
  }
```

There is more to explore and find in this class, but I'll leave that for you to have some fun.

Updating the Animator object

We made two major updates to our `Animator` class, which was originally created in the *Animating independent layers* recipe in *Chapter 6*, *Bringing Static Things to Life*. The first change was integrating callback methods. The idea of a callback is very similar to events. Callbacks enable us to call a function when something happens. This way of working enables us to only have one listener at a time. To do this, we start by creating the two following variables that are our callback functions:

```
function Animator(refreshRate){
  this.onUpdate = function(){};
  this.onComplete = function(){};
```

We then trigger both functions in the `Animator` class in their relevant location (on update or on complete). In our `GoogleMapTraveler` object, we override the default functions with functions that are internal to the `GoogleMapTraveler` object.

Our second and last major update to the `Animator` object is that we added smarter, more detailed logic to enable our animator to animate both to positive and negative areas. Our original animation didn't accommodate animating latitude/longitude values, and as such we tweaked the core animation logic.

This covers the major new things we explored in this recipe. This recipe is jam-packed with many other small things we picked up throughout the chapters. I truly hope you've enjoyed this journey with me, as this is the end of our book. Please feel free to share with me your work and insight. You can find me at `http://02geek.com` and my e-mail is `ben@02geek.com`.