Google Earth and JavaScript

Duncan Temple Lang, University of California at Davis

Table of Contents

..... 1
Getting Started 1
JavaScript 5
Connecting to an R plot 7

See http://code.google.com/apis/earth/documentation/reference/index.html

This is a brief introduction to putting Google Earth in a Web page. We'll describe the basic mechanism and then try to build more interesting and complex pages. Along the way we will discuss

1. HTML,
2. CSS
3. JavaScript in HTML,
4. the computational model for JavaScript
5. Google Earth API
6. event-driven or asynchronous programming

We cover the introduction to KML and the facilities in R to create KML in a separate document. This focuses on HTML, JavaScript and the GE API.

There is a wealth of widgets that we can include in an HTML page. See, for example, the Yahoo UI.

In what follows, R is no longer running. Eventually, we might mention how we can have R embedded in Firefox and so can have JavaScript access both R and Google Earth.

Getting Started

The first thing is to create an HTML document and insert a Google Earth panel into it. Basically, we create an HTML document. We have an HTML node at the root and within this there are 2 parts - a head and a body. The head is where we put

1. meta-information such as the title (to display in the browser's frame) and other details such as the author, keywords, etc.
2. JavaScript code, typically variable and function definitions.
3. Cascading Style Sheet (CSS) content to control the appearance and layout of elements

The **<body>** is where we put the content that is display.

To get Google Earth to display, we add JavaScript code to the **<head>**. We put the code inside **<script>** nodes. Often we put the code in comments also to ensure it doesn't mess up the HTML content, i.e. with < and > in the JavaScript code being confused for HTML markup.

```
<script type="text/javascript"><!--
  /* JavaScript code */
 var x = 1;
--></script>
```

Often, I like to put the JavaScript code in a separate file and have the browser read that.

```
  <script rel="text/javascript" src="geInit.js"></script>
```

Note

When things don't (appear to) work, make certain to use the Error Console within your browser, e.g. Tools->Error Console in Firefox.

1. JavaScript expressions end with ;
2. we have to declare variables with var varName;
3. we invoke functions as foo(arg1, arg2, ..., argn) as in R, but there are no named arguments.
4. arrays are created as var a = [1, 2, 6, 8];
5. we create objects with new ClassName(arg1, arg2, ...)
6. we invoke methods on objects with obj.methodName(arg1, arg2, ...)
7. we define functions as function funcName(param1, param2, ---, paramN) { body ....}

We also include a CSS file which controls the appearance of the different elements. Note how the body is laid out with color, margins and font. The text of the **<H1>** elements (the top-level section headers) are colored blue.

To coordinate where the GE display is located in the display, we put a ** ** element with an `id` attribute corresponding to the one used in the JavaScript code that loads GE, i.e. in the call to google.earth.createInstance().

We specify a class for the div which will control its appearance via the CSS rules. However, we also specify the style explicitly as an attribute, so that one wins out over the CSS specification.

Note that we put regular text into our document before the ** ** element. We have a section title in a **<H1>**. There are six such sections, i.e. **<H1>**, **<H2>**, ... **<H6>**. Paragraphs are marked via **<p>**. Links to URLs or internal anchors are marked up with **<a>**, e.g. <a href="http://www.targetURL.org/path/to/doc".

So our HTML document looks like

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html> <head>
 <title>Basic Google Earth in Browser example</title>
    <!-- *** Replace the key below below with your own API key, available at http:
    <script src="http://www.google.com/jsapi?key=ABQIAAAAwbkbZLyhsmTCWXbTcjbgbRSzH
    <script rel="text/javascript" src="geInit.js"></script>
    <link rel="stylesheet" href="basic.css"></link>
    <script><!-- var x = 1; --></script>
</head>


<body onload='init()' id='body'>

<h1>Getting Started with Google Earth in a browser</h1>

<p>
 This is a brief example of displaying Google Earth as part of a web
 page. We can do lots of interesting things with the layout, but for
 now we display it after this paragraph.
</p>

<!-- Note that we put a class here and set the style in CSS, but we
    specify the style explicitly here.  -->
 <div id='myGE' class="ge" style='border: 3px solid silver; height: 600px; width:


<hr>
<address><a href="http://www.stat.ucdavis.edu/~duncan">Duncan Temple Lang</a>
<a href=mailto:duncan@wald.ucdavis.edu>&lt;duncan@wald.ucdavis.edu&gt;</a></addres
<!-- hhmts start -->
Last modified: Mon Mar  8 03:11:33 PST 2010
<!-- hhmts end -->
</body> </html>
```

Our JavaScript file looks like

```
google.load("earth", "1");

var ge = null;

function init() {
  google.earth.createInstance("myGE", initCallback, failureCallback);
}

function initCallback(object) {
```

```
  ge = object;
  ge.getWindow().setVisibility(true);
}

function failureCallback(object) { alert("Failed to start Google Earth"); }
```

And our CSS code is

```
BODY {
  background: #FFF;
  margin-left: 1em; /* # 1%; */
  margin-right: 3em; /* #10%;  */
  font-family: Verdana, Arial, Helvetica, sans-serif;
}

div.ge {
  border: 1px solid red;
  clear: left;
}

h1 {
  color: #FF0000;
}
```

At this point, we are good to go. Load the basic.html file into your browser. After a few moments, you will hopefully see a Google Earth display within the document. If not, check the error console. (Get used to this!)

Now the problem is that we have not loaded our KML/KMZ file. To do this, we have to call google.earth.fetchKML. We do this after the plugin has been initialized, i.e. in the initCallback we passed to createInstance. In this callback, we add the command

```
  google.earth.fetchKml(ge, 'http://www.omegahat.org/RKMLDevice/boxplots.kmz', fet
```

We define our fetchCallback() as

```
function fetchCallback(obj) {
    ge.getFeatures().appendChild(obj);
    alert("Should be visible now " + obj);
}
```

The call to alert pops up a window in the browser and displays the message. The + in "Should be visible now " + obj concatenates the strings, coercing obj to a string. We can use multiple +'s in a row, e.g. "A number " + 1 + " and another value " + object.

Now one thing we will quickly notice is that there are no controls displayed on our Google Earth display. We can call a method of the GEPlugin object ge

```
ge.getNavigationControl().setVisibility(ge.VISIBILITY_AUTO);
```

There are various different settings for how the controls appear. This is specified by the sole argument. The options are SHOW, HIDE and AUTO prefixed by ge.VISIBILITY_

How do we find this stuff out? From the main reference page http://code.google.com/apis/earth/documentation/reference/index.html, click on the class of interest, e.g. GEPlugin. Within this, there is a list of methods. Each has a return type and you can click on that. For example, clicking on GEPlugin and then GENavigationControl, we have information about its methods.

There are many things we can do with the GEPlugin instance. We can load numerous KML/KMZ files. We can control where we are viewing. We can create new content within JavaScript, either with KML content as a string, or by calling methods to create new elements, e.g. Placemark, ...

JavaScript

Let's take this a little further than just showing the Google Earth display within our Web page. If that was all we were doing, we'd almost be as well off using the regular Google Earth application. (Of course, we have put HTML text around the Google Earth display to add context, etc.) But let's start by adding some user interface (UI) controls to the HTML page. We might add a series of radio buttons that allow the viewer to control which KML/KMZ file to view and to hide the others. We might also use a pull-down menu or use check-boxes which are not exclusive like radio buttons. So now we need to know about HTML forms.

The simplest HTML form element is a button. We can create this with

```
<input type="button" value="Show boxplots"/>
```

in our HTML document. The only thing we need to do is specify what happens when the viewer clicks the button. We can do this with an HTML form or a JavaScript command. In this version of our HTML file, we'll in-line the JavaScript code that creates the Google Earth instance and have it add the controls. But in addition to the three different functions (init, initCallback and failureCallback), we'll define a function that we can use to respond to the click of the button to fetch the KMZ file as we did before. We still have the fetchCallback function so all we need to do when responding to the viewer clicking the button is evaluate the command

```
  google.earth.fetchKml(ge, 'http://www.omegahat.org/RKMLDevice/boxplots.kmz', fet
```

So we can specify this as the command for the onclick attribute of our **<button>** element.

```
<input type="button" value="Show boxplots"
   onclick="google.earth.fetchKml(ge, 'http://www.omegahat.org/RKMLDevice/boxplots
```

However, we can do better. We can automate the rotation of the globe and zooming in to show the region of interest more clearly. The JavaScript to do this is something like

```
  var lookAt = ge.getView().copyAsLookAt(ge.ALTITUDE_RELATIVE_TO_GROUND);
  lookAt.setLatitude(37);
```

```
  lookAt.setLongitude(-122);
  ge.getView().setAbstractView(lookAt);
```

We could add these commands to the `onclick` attribute by separating the commands with ';'. However, it is better to define a function to set the view and then call this from the `onclick` attribute. So we can define a JavaScript function setView() as

```
function setView(long, lat) {
  var lookAt = ge.getView().copyAsLookAt(ge.ALTITUDE_RELATIVE_TO_GROUND);
  lookAt.setLongitude(long);
  lookAt.setLatitude(lat);
  lookAt.setAltitude(100); /* We also set the altitude. */
  ge.getView().setAbstractView(lookAt);
}
```

Then we can call it in our `onclick` attribute as

```
<input type="button" value="Show boxplots"
   onclick="google.earth.fetchKml(ge, 'http://www.omegahat.org/RKMLDevice/boxplots
```

Note

This is not working w.r.t. altitude

If we wanted to have check boxes, we would add something like the following to the HTML

```
<input type="checkbox" onchange="fetch('http://www.omegahat.org/RKML/Examples/City
<input type="checkbox" onchange="fetch('http://www.omegahat.org/RKML/Examples/seal
```

Now we need to define the JavaScript function fetch() . This basically looks at its argument, the name of the KML file to load, and sees whether it that file has already been loaded and is currently visible. If it is not, it fetches and displays it; if it is, it hides it. To do this, we need to keep a "hash" table or associative array containing the fetched object and indexed by the name of the file. This is used to store the object. We also need to know if the is currently displayed or not. This will allow us to toggle the display, while holding on to the object. So we use 3 "global" variables. (These are only visible within this HTML document.) Note the {} to initialize them. This makes an associative array, essentially the same as a named list in R. The fetch function is called with the name of the KML file to load. If it is not already in the table, we download it as before. The callback function specified in the fetchKml call needs to both show the KML object and also put the resulting object into the tables kmlObjects and kmlShown.

```
var kmlObjects = {};
  var kmlShown = {};
  var pendingURL;

  function fetch(url) {
     var tmp = kmlObjects[url];
     if(tmp == null) {
          // fetch it.
        pendingURL = url;
```

```
        google.earth.fetchKml(ge, url, fetchCallback);
    } else {
        if(kmlShown[url])
           ge.getFeatures().removeChild(tmp);
        else
           ge.getFeatures().appendChild(tmp);
          // change whether it was shown or not in the table.
        kmlShown[url] = !kmlShown[url];
    }
}

 function fetchCallback(obj) {
    var url = pendingURL;
    pendingURL = null;
    if(obj != null) {
      ge.getFeatures().appendChild(obj);
      kmlObjects[url] = obj;
      kmlShown[url] = true;
    } else
      alert("Failed to load " + url);
 }
```

Connecting to an R plot

Now, the next and final step in our example is to create a plot in R and connect that with the Google Earth display. We want to have an R plot beside the Google Earth display and to allow interactions on the R plot to change the view in the GE display. The R plot will be displayed in the browser. We naturally think of using a PNG or JPEG file. But how do we get interactivity on this? We can use an HTML image map. But a better way is to use an SVG plot. We can generate this in R. If

```
capabilities()["cairo"]
```

returns **TRUE**, the *svg*() function will be available. Alternatively, we can use the Cairo package.

SVG is an XML dialect. The SVG plot from R will have instructions to draw each of the graphical elements in our plot. We can work with this XML document and annotate the elements to specify JavaScript event handler code.

We'll continue with the temperature data. Let's start by loading the temperature data and arranging it into a data frame with a variable giving temperature for each of the 4 months:

```
data(temperature, package = "RKML")
z = with(temperature, unstack(temperature,  temp ~ month))
```

Now let's create a time series for each city. We can do this as a specific case of a parallel coordinates plot or by using *matplot*(): We might also consider drawing box plots for each month and then superimposing the parallel coordinate points

We'll use the simpler *matplot*() Now we will create the SVG plot

```
library(SVGAnnotation)
doc = svgPlot({
     matplot(t(z), type = "l", axes = FALSE, ylab = "Temperature",
```

```
           main = "Temperatures for 100 cities for different seasons")
    box()
    axis(2)
    axis(1, at = 1:4, c("Jan", "Apr", "Jul", "Oct"))
    matplot(t(z), type = "l")
  })
```

Next we get the SVG objects that represent the time series lines in the plot:

```
series = unlist(getPlotPoints(doc))
```

Now, what we want to do with these XML (SVG) nodes is to add an `onmouseover` attribute to each. The code in this attribute is responsible for moving the view in the Google Earth display to the corresponding city. We need to get the longitude and latitude for the corresponding city, and then we can call our setView JavaScript function.

```
cmds = sprintf("parent.setView(%f, %f)",
               - temperature$longitude[1:100],
               temperature$latitude[1:100])
invisible(
   mapply(function(node, city, cmd) {
          #addToolTips(node, city)
          xmlAttrs(node) = c(onmouseover = cmd)
   }, series, temperature$city[1:100], cmds))
```

We add a CSS file to the SVG file to control the appearance of rectangles used for the tooltips. Then we save the SVG document to a file.

```
addCSS(doc)
saveXML(doc, "cityTemps.svg")
```

We have choices as to where we do this. We can do it in R and insert the JavaScript code as the attribute. Alternatively, we can defer getting the longitude, latitude pair to JavaScript. There is no point in doing the latter, but we could. We can write R data to be available in JavaScript using the JSON format and the RJSONIO (or rjson) package.

Now we assemble the HTML document that displays the GE plugin and the SVG plot. We have the same basic structure. The only things we add are

1. the code to fetch and load the KMZ file for the temperatures

2. display the SVG figure in the body of the HTML document

We use the following to display the SVG

```
<object data="cityTemps.svg"
        type="image/svg+xml" width="960" height="800"/>
```

Here we specify the name of the file, its width and height and important the type of content that is being displayed.

How we arrange the GE display and the SVG plot is another issue. We may want to put them side by side or one on top of the other. We can do the latter with separate paragraphs. Side by side can be done with

tables, but we are much better off using the more general, flexible but somewhat more complicated layout mechanism available via styles and CSS.

The idea is that the viewer can mouse over any of the lines in