Extracted from:

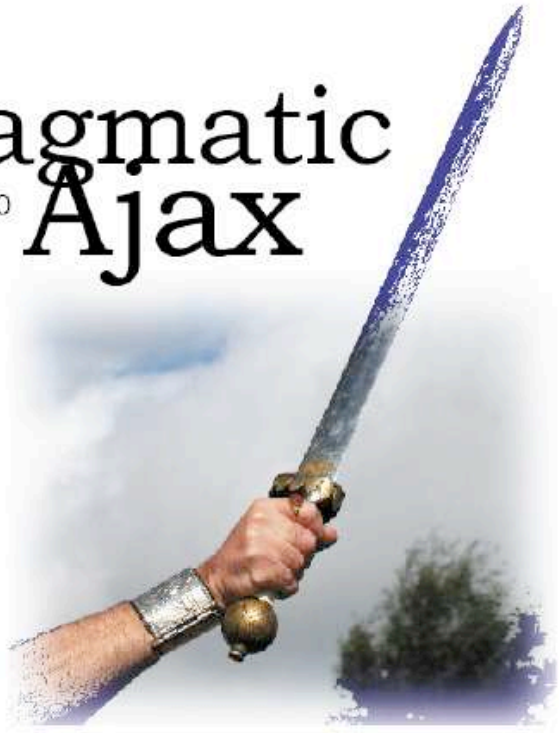
# Pragmatic Ajax

## A Web 2.0 Primer

The Pragmatic Programmers

# Pragmatic Ajax

A Web 2.0 Primer



*Justin Gehtland*
*Ben Galbraith*
*Dion Almaer*

**The Real Rocket Science**

OK, OK we admit—it isn't easy to create something like Google Maps. The geocoding features behind the scenes that map addresses to locations on a map, that normalize a maps features against satellite imagery to such an amazing degree that they can be overlaid on top of each other and look relatively accurate, and the plotting of routes from Point A to Point B are all incredibly nontrivial.

However, we maintain that it's not the geocoding features of Google Maps that is particularly innovative or impressive. MapQuest and other software packages have been doing this kind of work for years. No, what's impressive about Google Maps is the *web interface* on top of the geocoding engine. And it's that interface that we find easy, not the geocoding under the covers.

As our good friend Glenn Vanderburg says, though: "*Technically* it's easy, but the *conception* of this kind of interface is the really amazing part, just having the idea and then realizing that it could be done. So many things are simple once you've seen that they're possible." The take-home lesson is that Google Maps shows that once you have conceived of your next great UI idea, you can take comfort in knowing that the technical solution to implementing it might not be so daunting.

interface. (We should say, though, that we stand in awe of Lars Rasmussen and his team for being the brains and fingers behind Google Maps.) The reality is if we can create an interface like Google Maps in a couple of hours, imagine what a few capable web developers could do in a few weeks or a month.

## 2.2 Your Own Google Maps

In fact, we'll spare you from putting your imagination to the test. Let us show you firsthand how you can create your own version of Google Maps. In the next few pages, we'll walk you through the creation of Ajaxian Maps, our own derivative of the big GM. We'll start out by explaining how the Google Maps user interface works.

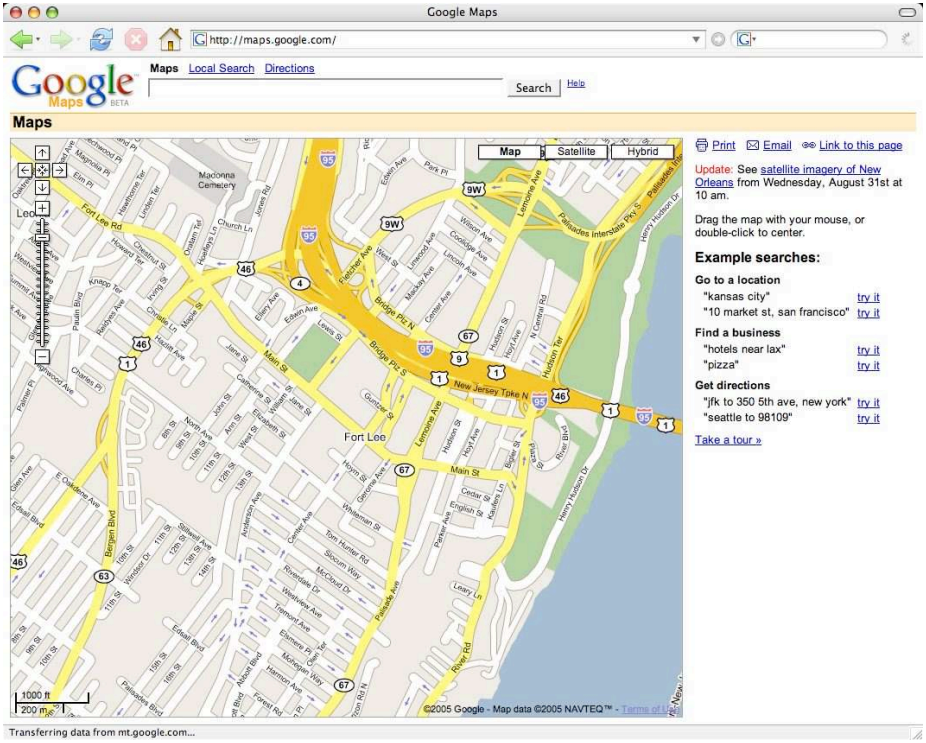



Figure 2.1: Google Maps

## Google Maps Deconstructed

We're going to break down the elements of Google Maps one by one. Let's start out with the most dramatic feature: the big scrolling map, the heart of the application.

### The Map

As you know, the map works by allowing you to interactively move the map by dragging the map using the mouse. We've seen mouse dragging in browsers for years, but the impressive bit is that the scrolling map is massive in size, can have the zoom level changed and so forth. How do they do that?

Of course, the browser could never fit such a large map in memory at once. For example, a street-level map of the entire world would prob-

**More Than A Million Pixels**

We say in "The Map" section that a street-level map of the world would be about a million square pixels. Actually, that number's a wild underestimate. At Google's highest level of magnification, a square mile consumes about 7,700,000 pixels. The Earth is estimated to contain 200,000,000 square miles, but only 30% of that is land, so let's reduce the number to 60,000,000 square miles.

Multiplying the number of pixels by the number of square miles in the Earth produces the mind boggling number of 462 million million pixels, which at 16.7 million colors (the color depth of any modern home computer) would consume at least three times that amount of memory in bytes. Of course, most image viewing programs have some sort of paged memory subsystem that views a portion of the image at any one time, but you get the idea....

ably be about a million pixels square. How much memory would it take to display that map? For the sake of conversation, let's assume that the map is displayed with just 256 colors, meaning each pixel would consume just 1 byte of memory. Such a map would require 1,000,000,000,000 bytes of memory, or roughly 1 terabyte (1000 gigabytes) of RAM. So, simply displaying an <*img*> element just isn't going to work.

What the Googlers do to work around the paltry amount of memory our desktop PCs have is split up the map into various tiles. These tiles are laid out contiguously to form one cohesive image. Figure 2.2, on the next page, shows an example of these tiles. While the size of these tiles has changed, the current size is 250 pixels square.

The tiles themselves are all laid out within a single HTML div element, and this div element is contained within another div; we'll call these two divs the *inner* and *outer* divs, respectively.

We mentioned just a moment ago that the browser couldn't fit the entire map image in memory. Of course, dividing a single map into an arbitrary number of tiles and then displaying all those tiles at once would consume an equal amount of memory as the entire image. To compensate for memory limitations, Google Maps virtualizes the grid of tiles

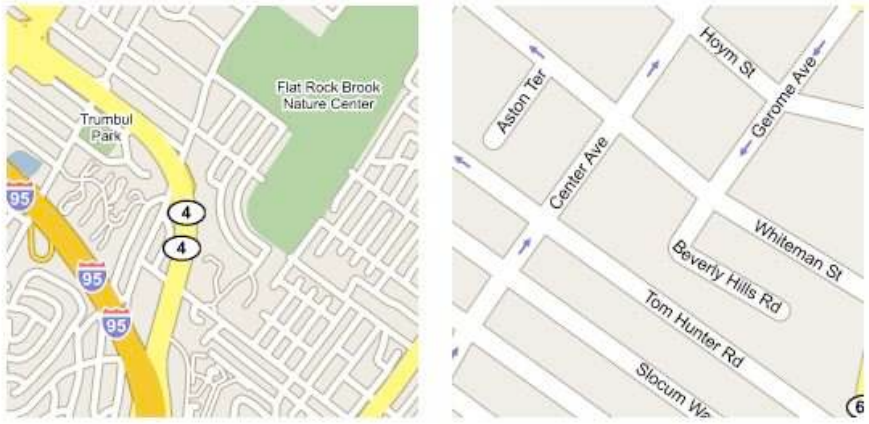



Figure 2.2: Google Maps Tiles

in memory and displays only the set of tiles that the user can see, in addition to a few additional tiles outside of the viewing area to keep the scrolling smooth.

If this whole grid virtualization mishmash sounds a little complex, don't worry; it's fairly straightforward, though it is the most complicated bit of the UI.

### Zoom Level

Another key feature of Google Maps is the ability to zoom in and out, enlarging or reducing the size of the map, which lets you get a view of the entire world at one moment and a view of your street the next. This is actually the simplest of the features to implement. Changing the zoom level just changes the size of the tile grid in memory as well as the URLs of the tile images that are requested.

For example, the URL to one of the tiles in Figure 2.2 is as follows:

```
http://mt.google.com/mt?v=w2.5&n=404&x=4825&y=6150&zoom=3
```

By changing the value of the zoom parameter to another value, such as 1, you can retrieve a tile at a different zoom level. In practice, it's not quite that simple because the grid coordinates change rather a great deal with each zoom level and they often become invalid.

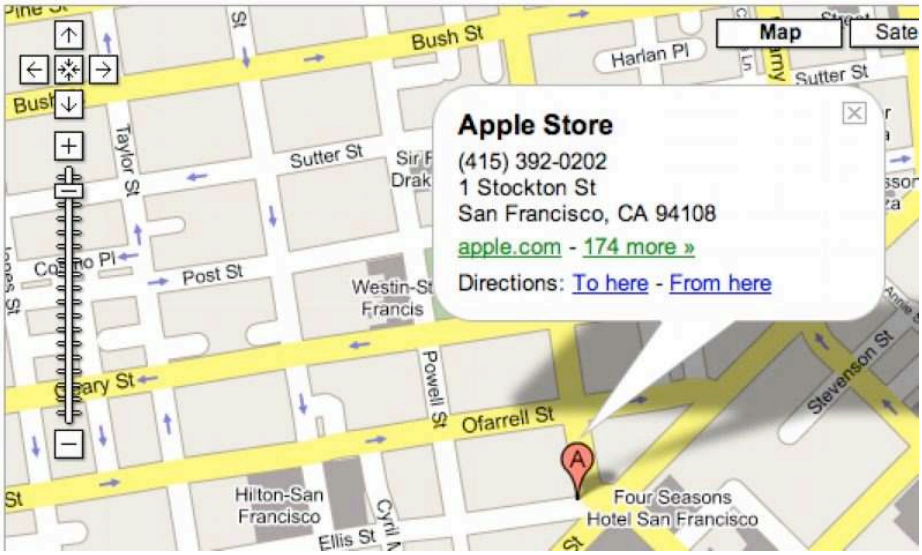



Figure 2.3: The Google Maps Push Pin and Dialog

How do they get the zoom level to constantly hover over the map in a constant position? The zoom level widget is an image embedded in the outer div, and makes use of transparency to blend in with the map image.

### Push Pins and Dialogs

Other neat-o features are the push pins and dialogs that appear after a search. Figure 2.3 shows these elements. These are especially cool because they both include rounded edges and shadows that make them blend in with the background map in a sophisticated fashion.

We said the zoom level was the easiest feature, and frankly, we were probably wrong. This is ridiculously easy. The push pins and dialogs are simply a PNG image. The PNG image format is supported by the major browsers and supports a nice feature called *alpha transparency*. Alpha transparency allows for more than just the simple transparency that GIF images support; it allows a pixel to be one of 254 different values between fully transparent and fully opaque, and it's this gradient transparency support that allows the push pins and dialog to use a shadow that blends in with the map.

*alpha transparency*

Showing these features is simply a matter of positioning images in the inner div at an absolute position.

#### Feature Review

There are other features, of course. But we'll stick to the set of features we've enumerated; we think these represent the vast majority of the "ooh, ahh" factors. In review, they were as follows:

- *The scrolling map*: This is implemented as an outer div containing an inner div. Mouse listeners allow the inner div to be moved within the confines of the outer div. Tiles are displayed as img elements inside the inner div, but only those tiles necessary to display the viewing area and a buffer area around it are present in the inner div.
- *The zoom level*: This is an image embedded in the outer div. When clicked, it changes the size of the grid representing the tiles and changes the URL used to request the tiles.
- *The push pins and dialogs*: These are PNG images with alpha transparency, placed in absolute positions within the inner div.

Now that we've deconstructed Google Maps a bit, let's set about implementing it.

## 2.3 Creating Ajaxian Maps

Because Ajaxian Maps won't bother with all of that geocoding mumbo jumbo, all of our heavy lifting will be in JavaScript. However, we will use Java to provide some server features and a few image manipulation tasks.

#### IE 6, Firefox 1.*x*, and Safari 2.*x* Only

We've tested this version of Ajaxian Maps in the three major browsers but haven't bothered with older versions and more obscure browsers (sorry, Opera users). It should work on older platforms, but without testing, we can't be sure we've caught everything.

#### Step 1: Create a Map

The first step in displaying a map is, err, creating it. While we could simply steal the wonderful map that Google Maps uses, Google might

not appreciate that. So, we'll go ahead and use a map that is explicitly open source. The Batik project (http://xml.apache.org/batik), an open-source Java-based SVG renderer, comes with an SVG map of Spain. We'll use that.

Because most browsers don't provide native support for SVG, we'll need to convert the map to a bitmap-based format. Fortunately, Batik can do that for us. One of the nice features of SVG is that it can scale to arbitrary sizes, so we could conceivably create a huge image for our map. However, creating truly huge images is a little tricky; because of memory limitations, we'd have to render portions of the SVG image, generate our tiles over the portions, and have some sort of scheme for unifying everything together. To keep this chapter simple, we'll just limit our map to 2,000 pixels in width and 1,400 pixels in height. In order to implement zooming, we'll also generate a smaller image that represents a view of the map in a zoomed-out mode.

The following code excerpt shows how to use Batik to convert the map of Spain into both a 2000x1400 pixel JPG file and a 1500x1050 pixel JPG file:

File 31

```
package com.ajaxian.amaps;

import org.apache.batik.apps.rasterizer.DestinationType;
import org.apache.batik.apps.rasterizer.SVGConverter;

import java.io.File;

public class SVGSlicer {
    private static final String BASE_DIR = "resources/";

    public static void main(String[] args) throws Exception {
        SVGConverter converter = new SVGConverter();

        // width in pixels; height auto-calculated
        converter.setWidth(2000);
        converter.setSources(new String[] { BASE_DIR + "svg/mapSpain.svg" });
        converter.setDst(new File(BASE_DIR + "tiles/mapSpain.jpg"));
        converter.setDestinationType(DestinationType.JPEG);
        converter.execute();

        converter.setWidth(1500);
        converter.setDst(new File(BASE_DIR + "tiles/mapSpain-smaller.jpg"));
        converter.execute();
    }
}
```

To compile the code, you'll need to put the Batik JARs in your classpath

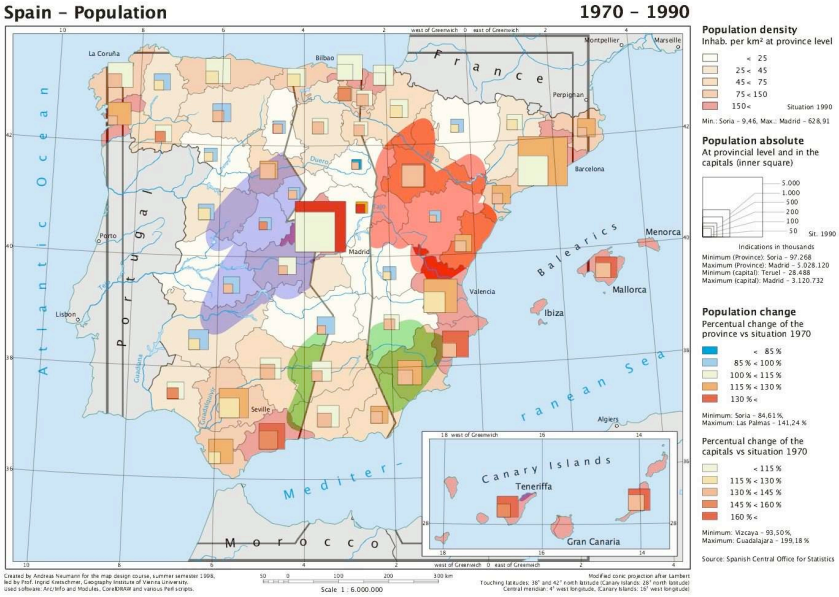



Figure 2.4: Batik's SVG Spain Map

(everything in BATIK_HOME and BATIK_HOME/lib) and place the source code in the following directory hierarchy: com/ajaxian/amaps. Figure 2.4 shows what either map JPG file should look like. You can also replace the value of the BASE_DIR variable with whatever is most convenient for you.

### Step 2: Create the Tiles

Now that we have a map at two different zoom levels, we need to slice it up into tiles. This is pretty easy with the nice image manipulation libraries available in many programming languages. We'll demonstrate how to do that with Java here:

File 30

```
package com.ajaxian.amaps;

import org.apache.batik.apps.rasterizer.DestinationType;
import org.apache.batik.apps.rasterizer.SVGConverter;

import javax.imageio.ImageIO;
import java.io.File;
import java.awt.*;
import java.awt.image.BufferedImage;
```

```
public class ImageTiler {
    private static final String BASE_DIR = "resources/";
    private static final int TILE_WIDTH = 100;
    private static final int TILE_HEIGHT = 100;

    public static void main(String[] args) throws Exception {
        // create the tiles
        String[][] sources = { { "tiles/mapSpain.jpg", "0" },
                {"tiles/mapSpain-smaller.jpg", "1"} };
        for (int i = 0; i < sources.length; i++) {
            String[] source = sources[i];
            BufferedImage bi = ImageIO.read(new File(BASE_DIR + source[0]));
            int columns = bi.getWidth() / TILE_WIDTH;
            int rows = bi.getHeight() / TILE_HEIGHT;
            for (int x = 0; x < columns; x++) {
                for (int y = 0; y < rows; y++) {
                    BufferedImage img = new BufferedImage(TILE_WIDTH, TILE_HEIGHT,
                            bi.getType());
                    Graphics2D newGraphics = (Graphics2D) img.getGraphics();
                    newGraphics.drawImage(bi, 0, 0, TILE_WIDTH, TILE_HEIGHT,
                            TILE_WIDTH * x, TILE_HEIGHT * y,
                            TILE_WIDTH * x + TILE_WIDTH,
                            TILE_HEIGHT * y + TILE_HEIGHT,
                            null);
                    ImageIO.write(img, "JPG", new File(BASE_DIR + "tiles/" +
                            "x" + x + "y" + y + "z" + source[1] + ".jpg"));
                }
            }
        }
    }
}
```

Note that to make things interesting, we made our tile size a bit smaller than Google Maps: 100 pixels square. We chose x0y0z0.jpg as the naming convention for the tiles, where the zeros are replaced with the $x$ and $y$ grid coordinates (0-based) and the zoom level (0 or 1; 0 is for the bigger of the two maps).

### Step 3: Creating the Inner and Outer Divs

Now that we have the image tiles, we can start building our map UI. We'll start with a simple web page, shown here:

File 32

```
Line 1   <html>
-            <head>
-                <title>Ajaxian Maps</title>
-                <style type="text/css">
5                   h1 {
-                       font: 20pt sans-serif;
```

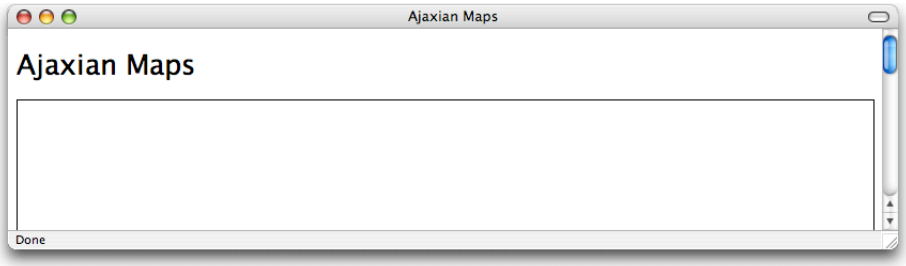



Figure 2.5: Humble Beginnings

```
                }
                #outerDiv {
                    height: 600px;
                    width: 800px;
                    border: 1px solid black;
                    position: relative;
                    overflow: hidden;
                }
            </style>
        </head>
        <body>
            <p>
                <h1>Ajaxian Maps</h1>
            </p>
            <div id="outerDiv">
            </div>
        </body>
    </html>
```

Figure 2.5 show this page. Pretty simple so far. Let's get to the good stuff. The div on line 21 will become what we've called the outer div. The outer div is the visible window into the tiles and will be entirely contained in the visible space within the browser. The inner div, on the other hand, will contain all the tiles and be much larger than the available visible space. Let's start out by giving it an inner div with some simple content:

File 33

```
<html>
    <head>
        <title>Ajaxian Maps</title>
        <style type="text/css">
            h1 {
                font: 20pt sans-serif;
            }
            #outerDiv {
```

```
                height: 600px;
                width: 800px;
                border: 1px solid black;
                position: relative;
                overflow: hidden;
            }

            #innerDiv {
                position: relative;
                left: 0px;
                top: 0px;
            }
        </style>
    </head>
    <body>
        <p>
            <h1>Ajaxian Maps</h1>
        </p>
        <div id="outerDiv">
            <div id="innerDiv">
                The rain in Spain falls mainly in the plains.
            </div>
        </div>
    </body>
</html>
```

Now we need to make the inner div large enough to contain all of the image tiles. We could just set a style on the inner div to make it some arbitrary size, as in * *, but we'll do this via JavaScript. Why? Well, because we'll implement the ability to change zoom levels a little later, we know we'll have to change the size of the inner div dynamically anyway, so we might as well start out that way. We'll use an onload JavaScript handler to initialize the size of the inner div once we load the page. Check out the code:

File 34

```
<html>
    <head>
        <title>Ajaxian Maps</title>
        <style type="text/css">
            h1 {
                font: 20pt sans-serif;
            }
            #outerDiv {
                height: 600px;
                width: 800px;
                border: 1px solid black;
                position: relative;
                overflow: hidden;
            }
```

```
            #innerDiv {
                position: relative;
                left: 0px;
                top: 0px;
            }
        </style>
        <script type="text/javascript">
            function init() {
                setInnerDivSize('2000px', '1400px')
            }

            function setInnerDivSize(width, height) {
                var innerDiv = document.getElementById("innerDiv")
                innerDiv.style.width = width
                innerDiv.style.height = height
            }
        </script>
    </head>
    <body onload="init()">
        <p>
            <h1>Ajaxian Maps</h1>
        </p>
        <div id="outerDiv">
            <div id="innerDiv">
                The rain in Spain falls mainly in the plains.
            </div>
        </div>
    </body>
</html>
```

OK, now we've got an inner div big enough to display the tiles for the largest of our two maps. Now we need to add the dragging functionality.

### Step 4: Dragging the Map

We'll implement dragging using three different mouse event listeners. When the user clicks the mouse in the map area, we'll use a listener to indicate that a drag operation has started. Now, if the user moves the mouse, we'll use a listener to move the inner div along with the user's mouse movements to create the dragging effect. Finally, we'll use a listener to turn off the dragging operation when the mouse is released. The following code demonstrates how we implemented the listeners:

File 35

```
// used to control moving the map div
var dragging = false;
var top;
var left;
var dragStartTop;
var dragStartLeft;
```

```
function init() {
    // make inner div big enough to display the map
    setInnerDivSize('2000px', '1400px');

    // wire up the mouse listeners to do dragging
    var outerDiv = document.getElementById("outerDiv");
    outerDiv.onmousedown = startMove;
    outerDiv.onmousemove = processMove;
    outerDiv.onmouseup = stopMove;

    // necessary to enable dragging on IE
    outerDiv.ondragstart = function() { return false; }
}

function startMove(event) {
    // necessary for IE
    if (!event) event = window.event;

    dragStartLeft = event.clientX;
    dragStartTop = event.clientY;
    var innerDiv = document.getElementById("innerDiv");
    innerDiv.style.cursor = "-moz-grab";

    top = stripPx(innerDiv.style.top);
    left = stripPx(innerDiv.style.left);

    dragging = true;
    return false;
}

function processMove(event) {
    if (!event) event = window.event;  // for IE
    var innerDiv = document.getElementById("innerDiv");
    if (dragging) {
        innerDiv.style.top = top + (event.clientY - dragStartTop);
        innerDiv.style.left = left + (event.clientX - dragStartLeft);
    }
}

function stopMove() {
    var innerDiv = document.getElementById("innerDiv");
    innerDiv.style.cursor = "";
    dragging = false;
}

function stripPx(value) {
    if (value == "") return 0;
    return parseFloat(value.substring(0, value.length - 2));
}
```

If you run the code at this point, you'll now be able to drag that inner

<*div*> around.

### Step 5: Displaying the Map Tiles

The next step requires us to populate our inner div with the map tiles. Our approach to this will be fairly simple. The scrolling map effect is achieved by moving an inner div inside of an outer div; therefore, the tiles we need to display are calculated by determining the current position of the inner div relative to the outer div and then working out which tiles are visible in the portion of the inner div that is visible. We'll then add those tiles to the inner div.

It turns out implementing this behavior is not terribly difficult. We'll create the function checkTiles() to do all this and call it from within the processMove() function. processMove() is called when the user drags the map, so by calling it from within, we'll be able to load our tiles as the map moves. The following code excerpt shows how we've added these elements to our JavaScript code; for now, checkTiles() is just stubbed out with comments:

File 39

```
function processMove(event) {
    if (!event) event = window.event;  // for IE
    var innerDiv = document.getElementById("innerDiv");
    if (dragging) {
        innerDiv.style.top = top + (event.clientY - dragStartTop);
        innerDiv.style.left = left + (event.clientX - dragStartLeft);
    }

    checkTiles();
}

function checkTiles() {
    // check which tiles should be visible in the inner div

    // add each tile to the inner div, checking first to see
    // if it has already been added
}
```

Now, let's implement our stubbed-out checkTiles() function.

#### Calculating the Visible Tiles

Calculating the set of tiles that the user can see in the inner <*div*> is fairly straightforward. To understand how this works, it will help to visualize the inner div as a grid where each grid cell is a placeholder of the tiles that we'll load. Figure 2.6, on the following page illustrates this concept.

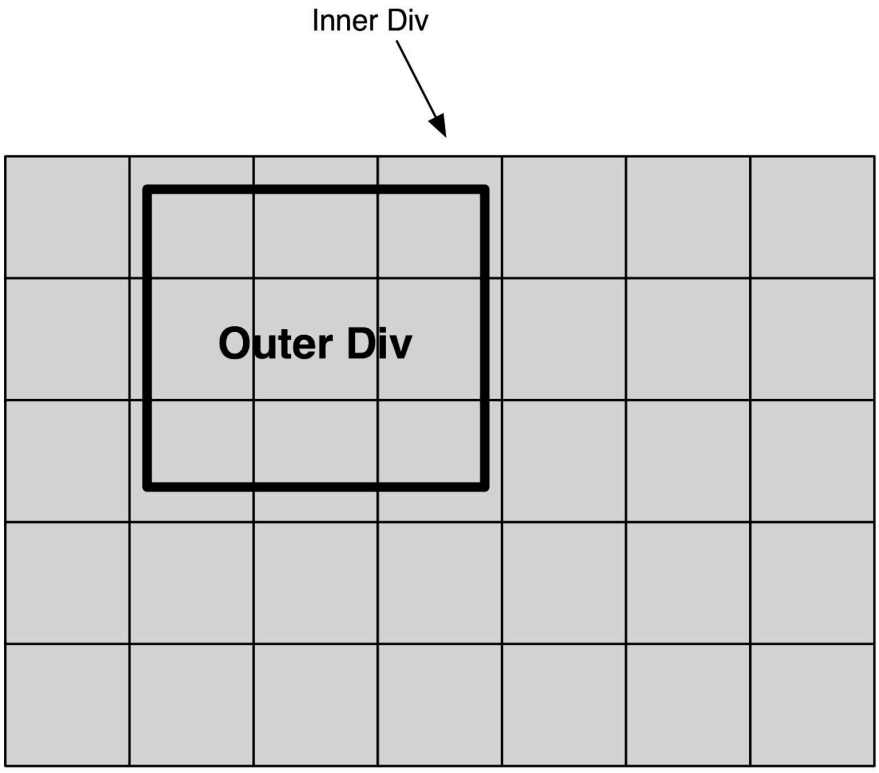



Figure 2.6: The Tile Grid

Because we can't load *all* the tiles in the grid up front, we'll need to calculate which of these grid cells are visible and load the tiles needed to fit into these cells. As Figure 2.6 shows, this is accomplished by calculating which grid cells are visible within the viewport created by the size of the outer div. In the figure, we see that nine cells are visible across three rows. Note that those cells that are only partially visible still count as being visible.

Let's see how to implement all this behavior we just described. To make things simple, we'll encapsulate all of the code to figure out which tiles are visible in a particular method, which we'll call getVisibleTiles(). The first thing we need to figure out in getVisibleTiles() is the position of the inner div relative to the outer div. This is fairly easy:

```
function getVisibleTiles() {
    var innerDiv = document.getElementById("innerDiv");
    var mapX = stripPx(innerDiv.style.left);
    var mapY = stripPx(innerDiv.style.top);
}
```

The stripPx() function, shown earlier, converts the string value returned by innerDiv.style.left (such as 100px) to a numeric value (say, 100). Now, we can divide these positions by the size of the tiles to work out the starting row and column of the tiles. This is just two lines of code:

```
var startX = Math.abs(Math.floor(mapX / tileSize)) - 1;
var startY = Math.abs(Math.floor(mapY / tileSize)) - 1;
```

Note that we haven't yet defined the tileSize variable; we'll do that globally (at the top of our JavaScript code), and you'll see it when we show the entire page in just a few paragraphs. (Or, you can see it now on the following page.) The call to Math.floor() will round the quotient to an integer, discarding the remainder (so 1.4 will be rounded down to 1). This will cause partial tiles to be displayed. Math.abs() converts negative values to a positive number, which in our case is necessary because the inner div position will nearly always be negative to the outer div, and because our tile columns/rows are always positive numbers. Finally, we subtract 1 from the result to make our map load the tiles a touch early for a smoother effect.

The final bit of calculation is to determine the number of rows and columns visible in the viewport:

```
var tilesX = Math.ceil(viewportWidth / tileSize) + 1;
var tilesY = Math.ceil(viewportHeight / tileSize) + 1;
```

As with tileSize(), we'll declare both viewportWidth and viewportHeight as global variables and show that in just a bit. We use Math.ceil(), the opposite of Math.floor() (so it rounds the quotient up regardless of the size of the remainder), to ensure that if any portion of a column or row is visible, we'll display it. And, just as we subtracted 1 from the index of the tiles in the previous lines, we'll add 1 to the number of columns and rows to make the scroll effect smooth.

We now have all the data we need to calculate all of the visible tiles in the viewport plus, as we've discussed, a few around the edges that aren't immediately visible but will be shortly. Now we'll build an array that contains all of the tiles that need to be loaded. To build this array, we'll write two for loops, one nested inside the other, that each perform an iteration for each column and row that is currently visible. Inside

each loop iteration, we'll add the column and row number of each tile to display:

```
var visibleTileArray = [];
var counter = 0;
for (x = startX; x < (tilesX + startX); x++) {
    for (y = startY; y < (tilesY + startY); y++) {
        visibleTileArray[counter++] = [x, y];
    }
}
return visibleTileArray;
```

Note that we're actually creating a two-dimensional array; the value of each item in our array is another array. We did this because we need to pass back two values: the column and row index. And now, we're done calculating the tiles that are visible in the inner div, and we can move on and work on the code to actually display them. But first, let's review all of the code we've written so far:

File 36

```
function checkTiles() {
    // check which tiles should be visible in the inner div
    var visibleTiles = getVisibleTiles();

    // add each tile to the inner div, checking first to see
    // if it has already been added
}

function getVisibleTiles() {
    var innerDiv = document.getElementById("innerDiv");

    var mapX = stripPx(innerDiv.style.left);
    var mapY = stripPx(innerDiv.style.top);

    var startX = Math.abs(Math.floor(mapX / tileSize)) - 1;
    var startY = Math.abs(Math.floor(mapY / tileSize)) - 1;

    var tilesX = Math.ceil(viewportWidth / tileSize) + 1;
    var tilesY = Math.ceil(viewportHeight / tileSize) + 1;

    var visibleTileArray = [];
    var counter = 0;
    for (x = startX; x < (tilesX + startX); x++) {
        for (y = startY; y < (tilesY + startY); y++) {
            visibleTileArray[counter++] = [x, y];
        }
    }
    return visibleTileArray;
}
```

### Displaying the Visible Tiles

We've now coded half of the checkTiles() function, which as you may recall is the function responsible for both calculating the visible tiles and displaying them. Now, let's implement the other half of that function: displaying the tiles.

All we need to do here is iterate through each element of the array of visible tiles we returned from the getVisibleTiles() function and for each array element add a tile image to the inner div. Here's the new code for our checkTiles() function:

File 37

```
Line 1  function checkTiles() {
        // check which tiles should be visible in the inner div
        var visibleTiles = getVisibleTiles();

5       // add each tile to the inner div, checking first to see
        // if it has already been added
        var innerDiv = document.getElementById("innerDiv");
        var visibleTilesMap = {};
        for (i = 0; i < visibleTiles.length; i++) {
10          var tileArray = visibleTiles[i];
            var tileName = "x" + tileArray[0] + "y" + tileArray[1] + "z0";
            visibleTilesMap[tileName] = true;
            var img = document.getElementById(tileName);
            if (!img) {
15              img = document.createElement("img");
                img.src = "resources/tiles/" + tileName + ".jpg";
                img.style.position = "absolute";
                img.style.left = (tileArray[0] * tileSize) + "px";
                img.style.top = (tileArray[1] * tileSize) + "px";
20              img.setAttribute("id", tileName);
                innerDiv.appendChild(img);
            }
        }
    }
```

We start out on line 8 by creating an empty map (*map* in the JavaScript sense; a hash that contains key-to-value mappings). We're going to add an entry to this map for each visible image; we'll discuss why we're doing this a little later.

On line 9, we start looping through each element in the array we sent back from getVisibleTiles(). For each element, we build the name of the image file that will be loaded in. (If you recall, the file-naming convention we chose in Step 2 was x0y0z0, where the numbers are replaced with the index of the tile in the tile grid.) We also use this name as the key in the visibleTilesMap variable, and on lines 13 and 20 you can see that

# The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style, and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help programmers stay on top of their game.

# Visit Us Online

**Pragmatic Ajax**
`pragmaticprogrammer.com/titles/ajax`
Source code from this book, errata, and other resources. Come give us feedback, too!

**Register for Updates**
`pragmaticprogrammer.com/updates`
Be notified when updates and new books become available.

**Join the Community**
`pragmaticprogrammer.com/community`
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

**New and Noteworthy**
`pragmaticprogrammer.com/news`
Check out the latest pragmatic developments in the news.

# Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragmaticprogrammer.com/titles/ajax.

# Contact Us

| | |
|---|---|
| Phone Orders: | 1-800-699-PROG (+1 919 847 3884) |
| Online Orders: | www.pragmaticprogrammer.com/catalog |
| Customer Service: | orders@pragmaticprogrammer.com |
| Non-English Versions: | translations@pragmaticprogrammer.com |
| Pragmatic Teaching: | academic@pragmaticprogrammer.com |
| Author Proposals: | proposals@pragmaticprogrammer.com |