BING MAPS V8

BY JAMES MCCAFFREY

SUCCINCTLY E-BOOK SERIES

Bing Maps V8 Succinctly

By

James McCaffrey

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

Important licensing information. Please read.

Technical Reviewer: Chris Lee
Copy Editor: John Elderkin
Acquisitions Coordinator: Tres Watkins, content development manager, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

James McCaffrey works for Microsoft Research in Redmond, WA. He holds a B.A. in psychology from the University of California at Irvine, a B.A. in applied mathematics from California State University at Fullerton, an M.S. in information systems from Hawaii Pacific University, and a doctorate in cognitive psychology and computational statistics from the University of Southern California. James enjoys exploring all forms of activity that involve human interaction and combinatorial mathematics, such as the analysis of betting behavior associated with professional sports, machine learning algorithms, and data mining.

Chapter 1 Getting Started

It's no secret that all sorts of enterprises are generating ever-increasing amounts of data. And much of that data contains geolocation information. Any statistics related to data are doomed to be instantly obsolete, but it's generally accepted that 80% of all information has some sort of geolocation reference.

The Bing Maps V8 library is a very large collection of JavaScript code that allows web developers to place a map on a webpage, query for data, and manipulate objects on a map. Put another way, the Bing Maps library enables you to create geo-applications.

The Bing Maps V8 library was released in June 2016. In terms of number of features and performance, the new version offers significant improvements over the version V7 library. Before V7, Bing Maps was called Virtual Earth, and some legacy code still refers to VE.

In this e-book, we'll assume that you have basic web development knowledge, but we won't assume you know anything about geo-applications or the Big Maps library. If you can get a web server up and running, you should be able to follow along.

We'll cover creating web-based geo-applications but will not cover using Bing Maps V8 with WPF or Silverlight applications, and each section of this e-book will present a complete, self-contained demo web application.

Every developer I know learns how to program using a new technology by getting an example application up and running, then experimenting with the program by making changes. If you want to learn how to use the Bing Maps library in the most efficient way possible, copy the source code from a demo web application in this e-book into your favorite editor, save it on a local web server, play with the application, then fiddle with the code.

You can get all the source code and data files used in this e-book at: https://github.com/jdmccaffrey/bing-maps-v8-succinctly.

I will not present hundreds of code snippets here. That's what documentation is for. Instead, I've tried to pick key techniques and combine a few of them in each demo web application. Because the Bing Maps library is constantly evolving, it's more important to have a deep understanding of a few key techniques than it is to have tentative knowledge of hundreds of rarely used techniques.

In my opinion, the most difficult part of learning any technology is getting a first application to run. After that, it's all just details. But getting started can be frustrating, so the purpose of this first chapter is to make sure you can get your initial geo-application working.

In section 1.1, you'll learn how to create the simplest possible web application that uses the Bing Maps V8 library. In section 1.2, you'll learn how to get a Bing Maps key so that you can use the Bing Maps services for free (subject to usage limitations). In section 1.3, you'll learn how to communicate between ordinary HTML controls on a webpage and objects on a map.

Enough chit-chat already. Let's get started.

1.1 Hello World

Often, the most difficult part of learning a new technology is simply getting over the initial hurdle and making a first example work. The goal of this section is to help you create a very simple Bing Maps V8 “Hello World” (literally) demo page.

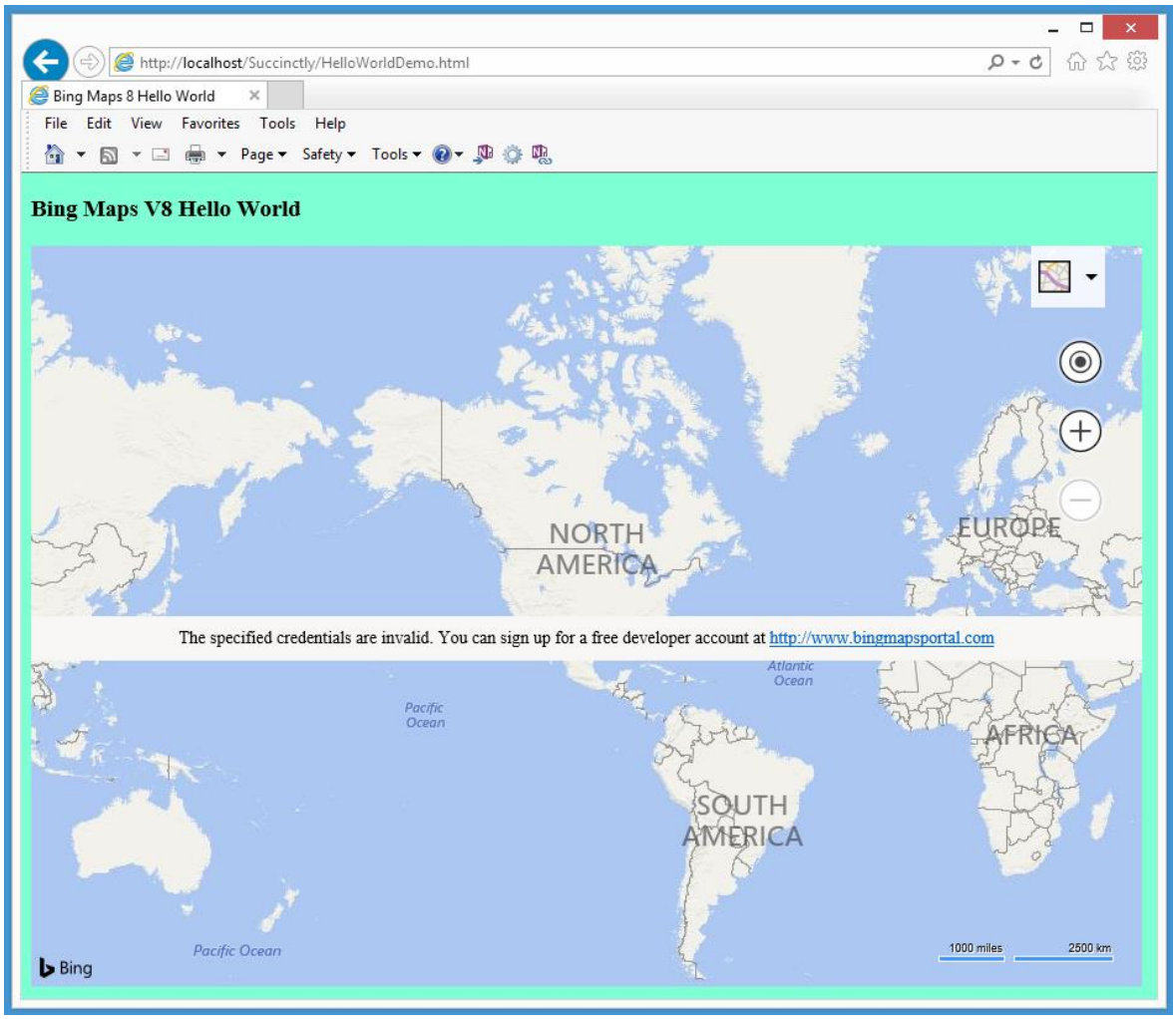

Figure 1: Hello World Demo

The first thing you might notice in Figure 1 is the message across the center of the map that indicates “the specified credentials are invalid.” I’ll walk you through the process of getting a Bing Maps key in the next section, but for now you can ignore the error message.

As you’ll see shortly, the HTML and JavaScript code needed to place a Bing Maps map on a webpage is quite simple. The URL in the browser address bar in Figure 1 indicates the demo webpage is hosted locally—on a Windows 10 machine running the Internet Information Services (IIS) web server. The examples in this e-book contain only HTML and JavaScript and have no dependencies, which means any web server will work.

The demo web application is named HelloWorldDemo.html and is defined in a single file.

Code Listing 1: HelloWorldDemo.html

```
<!DOCTYPE html>
<!-- HelloWorldDemo.html -->

<html>
  <head>
    <title>Bing Maps 8 Hello World</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <script type='text/javascript'>

    var map = null;

    function GetMap()
    {
      var options = {
        credentials: "TheBingMapsKeyGoesHere",
        center: new Microsoft.Maps.Location(35.00, -100.00),
        mapTypeId: Microsoft.Maps.MapTypeId.road,
        zoom: 2,
        enableClickableLogo: false,
        showCopyright: false
      };

      var mapDiv = document.getElementById("mapDiv");  // Where to place map.
      map = new Microsoft.Maps.Map(mapDiv, options);   // Create the map.
    }

    </script>
  </head>

  <body style="background-color:aquamarine">
    <h3>Bing Maps V8 Hello World</h3>
    <div id='mapDiv' style='width:900px; height:600px;'></div>

    <script type='text/javascript'
      src='http://www.bing.com/api/maps/mapcontrol?callback=GetMap'
      async defer>
    </script>

  </body>
</html>
```

If you have a web server up and running, you can copy this file, place it on your web server, and navigate to the page. Unfortunately, working with web servers can be somewhat nightmarish with regards to configuration settings. In this e-book, we'll assume you have access to a machine with a running web service. Figure 2 shows the IIS components on the machine I've used throughout.

Most versions of Windows contain IIS, but IIS is not enabled by default. You can enable IIS by going to the Windows Control Panel | Programs and Features, then clicking **Turn Windows features on or off** in the upper-left corner. The entry marked "Internet Information Services" will

be cleared by default. If you select that entry, all the subcomponents you need to run IIS and the examples in this e-book will be enabled.

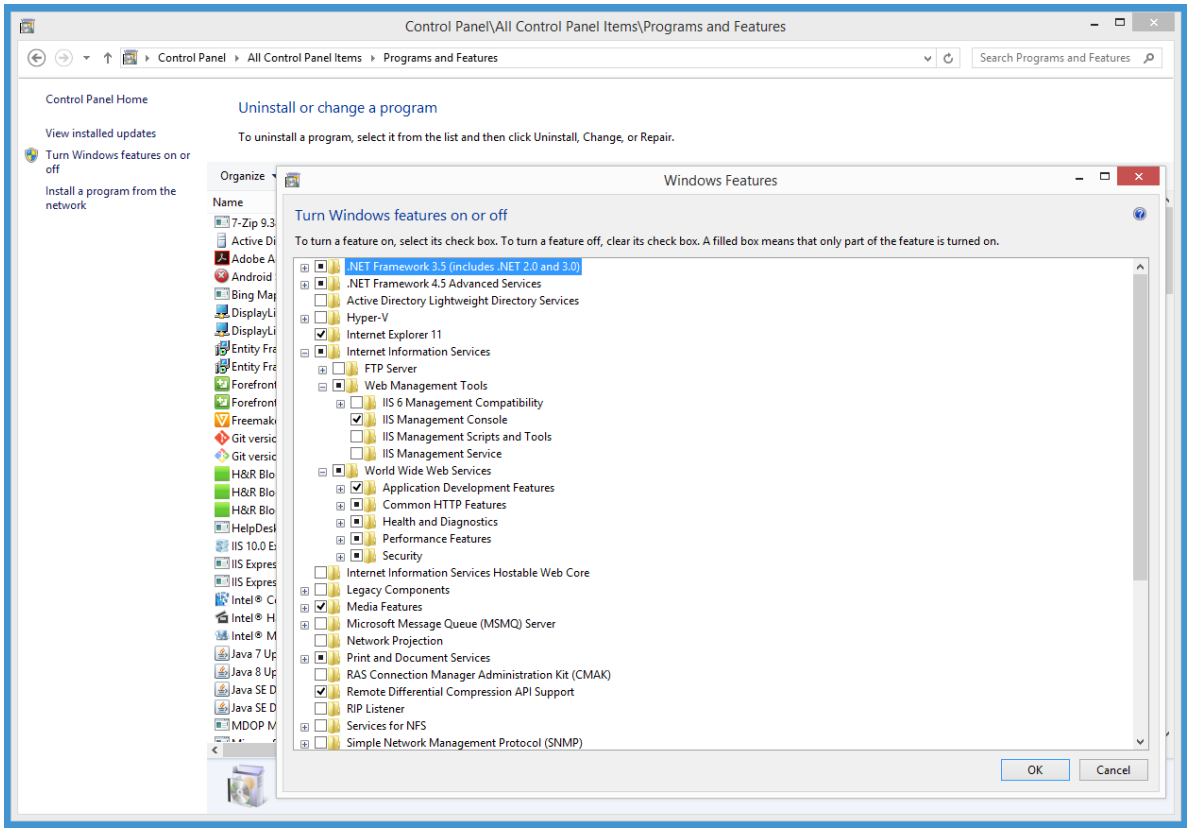

Figure 2: Internet Information Services Components

After you click **OK** in the lower-right corner, a dialog box will appear showing you the progress made as changes are applied to your machine.

In order to make the Hello World demo webpage, I first created a directory named “Succinctly” in the C:\inetpub\wwwroot root directory. Next, I launched the Notepad program using the “Run as administrator” option so that I’d be able to save in the restricted inetpub subdirectories.

Notice that the webpage has two script sections, one in the HTML head area and one in the HTML body area. The body-area script loads the Bing Maps V8 library asynchronously:

```
<script type='text/javascript'
  src='http://www.bing.com/api/maps/mapcontrol?callback=GetMap'
  async defer>
</script>
```

When the library has loaded, control is transferred to a program-defined **`GetMap()`** function. Asynchronous loading is a new feature in Bing Maps V8, and it allows the HTML components of a webpage to display immediately when the library is slow to load.

You can also load the library synchronously if you'd rather. To do so, remove the script tag from the body section and replace it with a script tag in the HTML head section:

```
<script type='text/javascript' src='http://www.bing.com/api/maps/mapcontrol'>
```

Next, you modify the HTML body tag this way:

```
<body onload="GetMap();">
```

Inside the main script area, a script-global object named **map** is declared and initialized to **null**. In this example, the **map** object is only accessed by function **GetMap()**, which means the object could have been declared local to the function. The name "map" is not required, but it is by far the most common choice.

Function **GetMap()** begins by setting the initial display options:

```
function GetMap()
{
  var options = {
    credentials: "TheBingMapsKeyGoesHere",
    center: new Microsoft.Maps.Location(35.00, -100.00),
    mapTypeId: Microsoft.Maps.MapTypeId.road,
    zoom: 2,
    enableClickableLogo: false,
    showTermsLink: false
  };
. . .
```

Bing Maps library **Location** objects accept a latitude (the up-down value in degrees relative to the equator) followed by a longitude (the left-right value relative to the prime meridian), which is followed by two optional values related to altitude.

The map is displayed with these two statements:

```
. . .
  var mapDiv = document.getElementById("mapDiv");
  map = new Microsoft.Maps.Map(mapDiv, options);
}
```

The **mapDiv** object specifies where to place the map:

```
<div id='mapDiv' style='width:900px; height:600px;'></div>
```

Notice that you typically specify the size of the map in the map's HTML div container rather than as one of the map options. You can specify the width and height of the map area using pixels, as the demo does, or by using the newer CSS viewport units, vw and vh.

Map options are divided into two categories. There are eight options that control characteristics directly related to the visual appearance of the map, and there are 21 options that control additional characteristics (some of which are visual).

Table 1 lists the eight properties of the **ViewOptions** object. The demo uses the center, mapTypeId, and zoom properties. The 21 additional properties are contained in the **MapOptions** object.

Table 1: ViewOptions Object Properties

Name	Description
bounds	defines an area to display as a **LocationRect** object
center	specifies the center of the map as a **Location** object
heading	directional heading as a number where 0 = North, 90 = East, etc.
labelOverlay	specifies labels are to be displayed as a **LabelOverlay** object
mapTypeId	aerial, ordnanceSurvey (UK only), road, streetside
padding	padding in pixels for each side of map
pitch	angle relative to horizon
zoom	zoom level (1-19)

The eight visual ViewOptions properties can be set when we call the **Map** constructor, as the demo does, or the properties can be set programmatically after a map has been instantiated using the **setView()** function. For example:

```
map.setView({ zoom: 15 });
```

Some of the 21 MapOptions properties can be set only when calling the constructor, but some properties can be set programmatically using the **setOptions()** function. For example:

```
map.setOptions({ disableZooming: true });
```

In summary, you can load a map asynchronously (preferred) or synchronously. There are a total of 29 properties that can be set when calling the **Map()** constructor, categorized as ViewOptions properties and MapOptions properties.

Resources

For detailed information about the Map constructor, see:
https://msdn.microsoft.com/en-us/library/mt712547.aspx.

For detailed information about the eight ViewOptions properties, see:
https://msdn.microsoft.com/en-us/library/mt712641.aspx.

For detailed information about the 21 MapOptions properties, see:
https://msdn.microsoft.com/en-us/library/mt712646.aspx.

1.2 Creating a Bing Maps key

You can easily obtain a Bing Maps key that allows you to use Bing Maps services for free (subject to usage limitations). I'll walk you through the process, screen by screen, so that you'll know what to expect. Creating a Bing Maps key does not require you to submit credit card or other payment information.

You can start by opening a browser and navigating to the main Bing Maps portal site at https://bingmapsportal.com. In order to sign in and create a Bing Maps key, you'll need a Microsoft Account (typically a hotmail.com or live.com account).

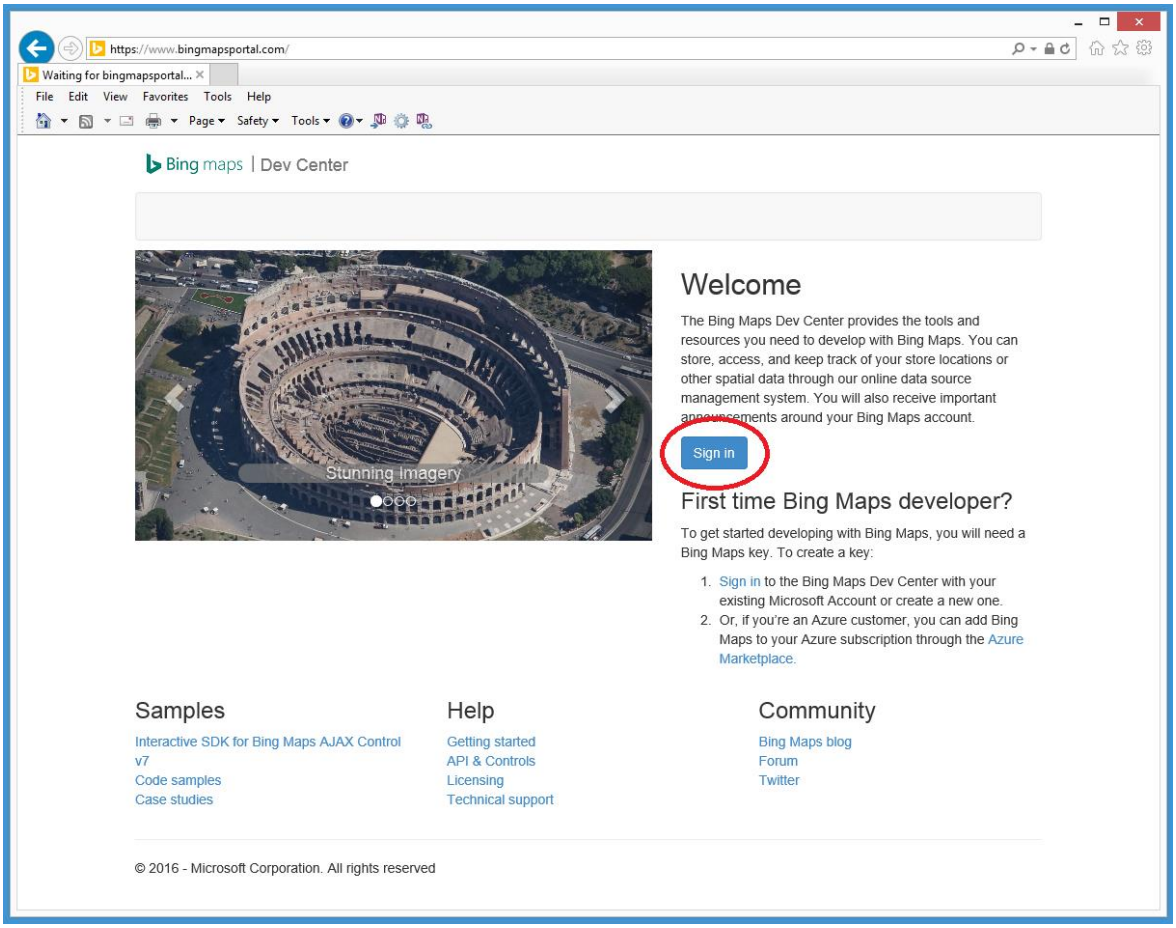

Figure 3: Bing Maps Portal Site Sign In

Click **Sign In** and you'll be directed to the Microsoft Sign In page. If you don't have a Microsoft account, you can create one on the Sign In page.

After signing in, you'll be redirected to the Bing Maps Developer Center site. Click the **My Account** drop-down in the upper left and select the **My Keys** option.

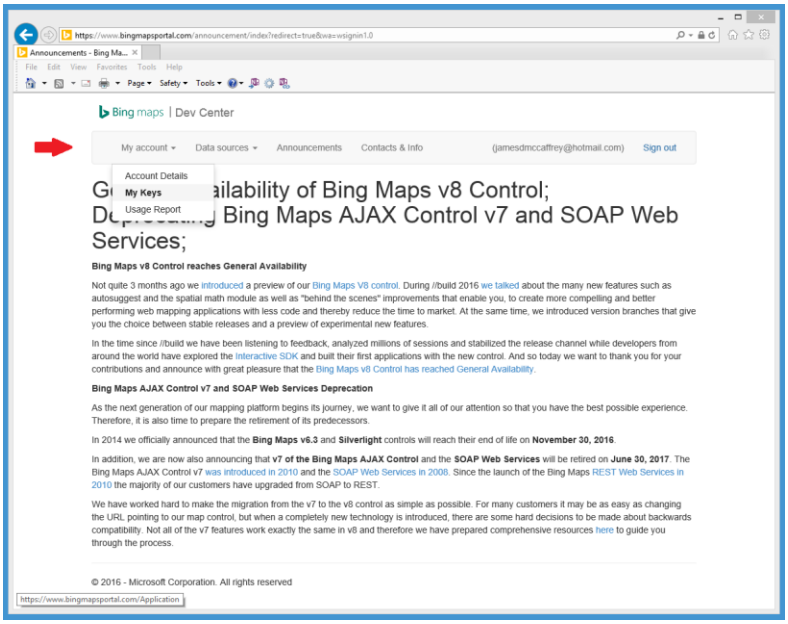

Figure 4: Select the My Keys Option

You'll be transferred to a page that lists your existing Bing Maps keys if you have any. Click the link labeled "Click **here** to create a new key."

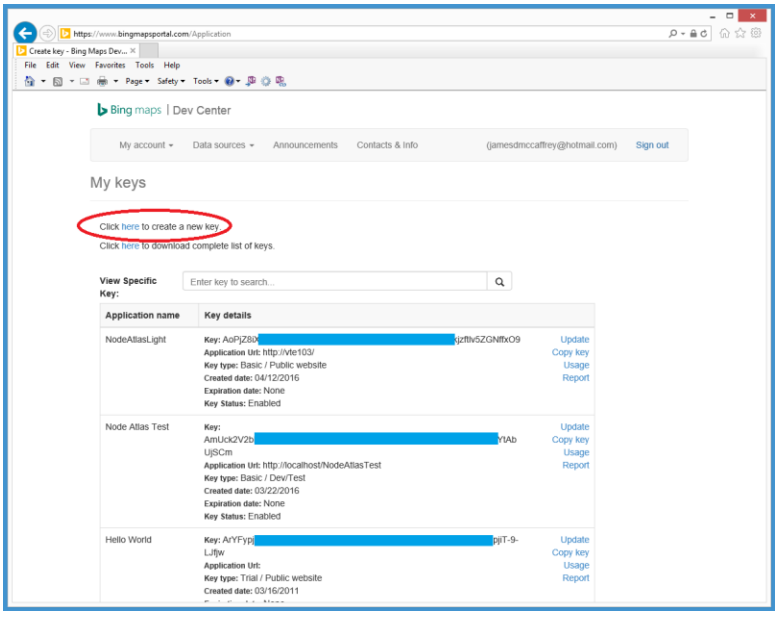

Figure 5: Click on the Create a New Key Link

Fill in any Application Name (in order to help you distinguish among multiple keys). For the Application URL, you can use http://localhost/Succinctly or something similar. For the Key Type, select **Basic**. For the Application Type, select **Dev/Test**. Click **Create**.

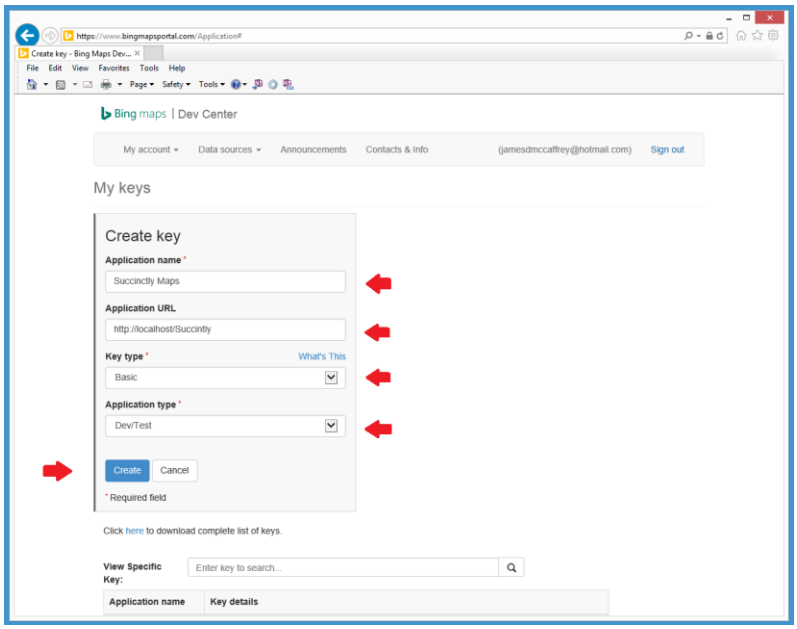

Figure 6: Supply Information for the New Key

Your new key will be created and its value displayed in the My Keys area. A key value is a long string such as “AgwlfQ-LtBJcKS77c5Bqvkpfx9aesf0X5KxXvahfcuSFdD-htUJCcFqcZ_koP2ke.”

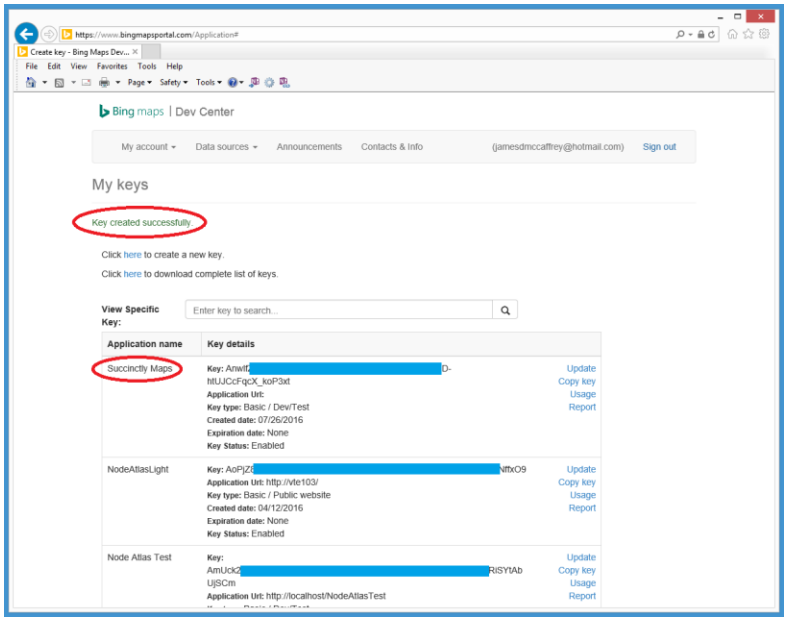

Figure 7: New Key Created

Now that you have a Bing Maps key, you should verify that it works. Recall that the HelloWorld.html demo page of the previous section sets up map options this way:

```
var options = {
  credentials: "TheBingMapsKeyGoesHere",
  center: new Microsoft.Maps.Location(35.00, -100.00),
  mapTypeId: Microsoft.Maps.MapTypeId.road,
  zoom: 2,
  enableClickableLogo: false,
  showCopyright: false
};
```

Open HelloWorld.html using Notepad (or the editor of your choice) with administrative privileges, copy the key value from the Bing Maps Dev Center page, and place it as the **`credentials`** value. Save the page, then use a browser to navigate to the page and verify that the invalid credentials error message across the center of the map is no longer displayed. And, by the way, the **`showCopyright`** property is undocumented—you should set it to false only during development or experimentation.

The Bing Maps portal site has a nice page that allows you to monitor usage:

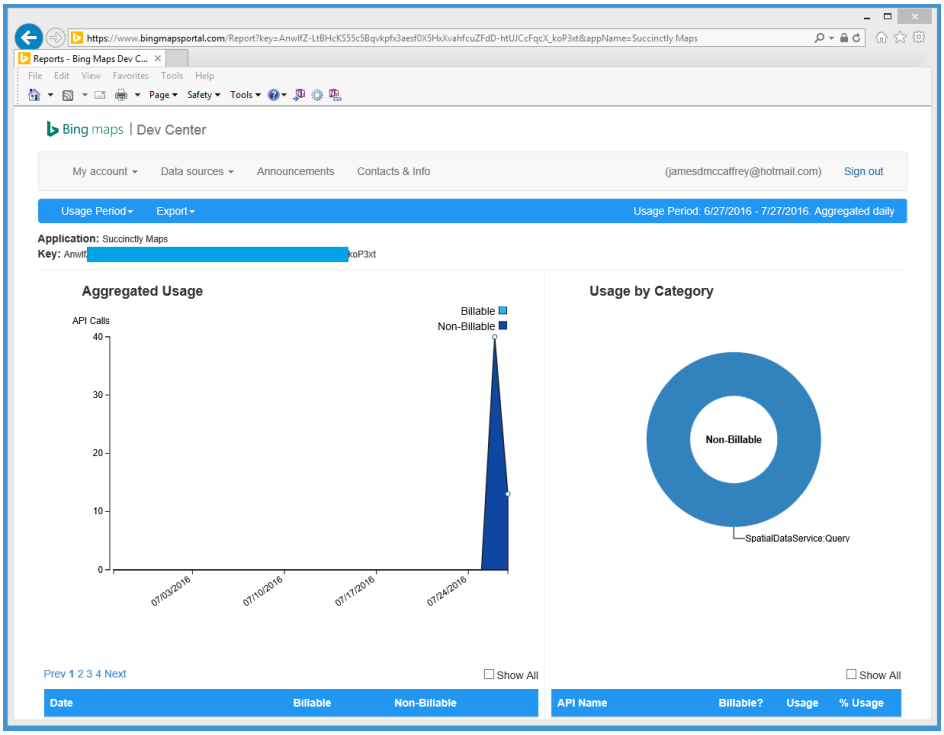

Figure 8: Bing Maps Usage Information

In summary, getting a free Bing Maps key for limited, noncommercial use is relatively simple. The process does not require you to submit credit card or other payment information. Go to https://www.bingmapsportal.com and sign in using a Microsoft account (which you can create on the fly), then follow the simple instructions.

1.3 User input and output

When working with a Bing Maps geo-application, user input is typically fired from a click on an HTML control on the container page or on the map (or an object on the map). Output is usually sent to a control on the container page, to a JavaScript alert box, or as an instruction to perform some action on the map.

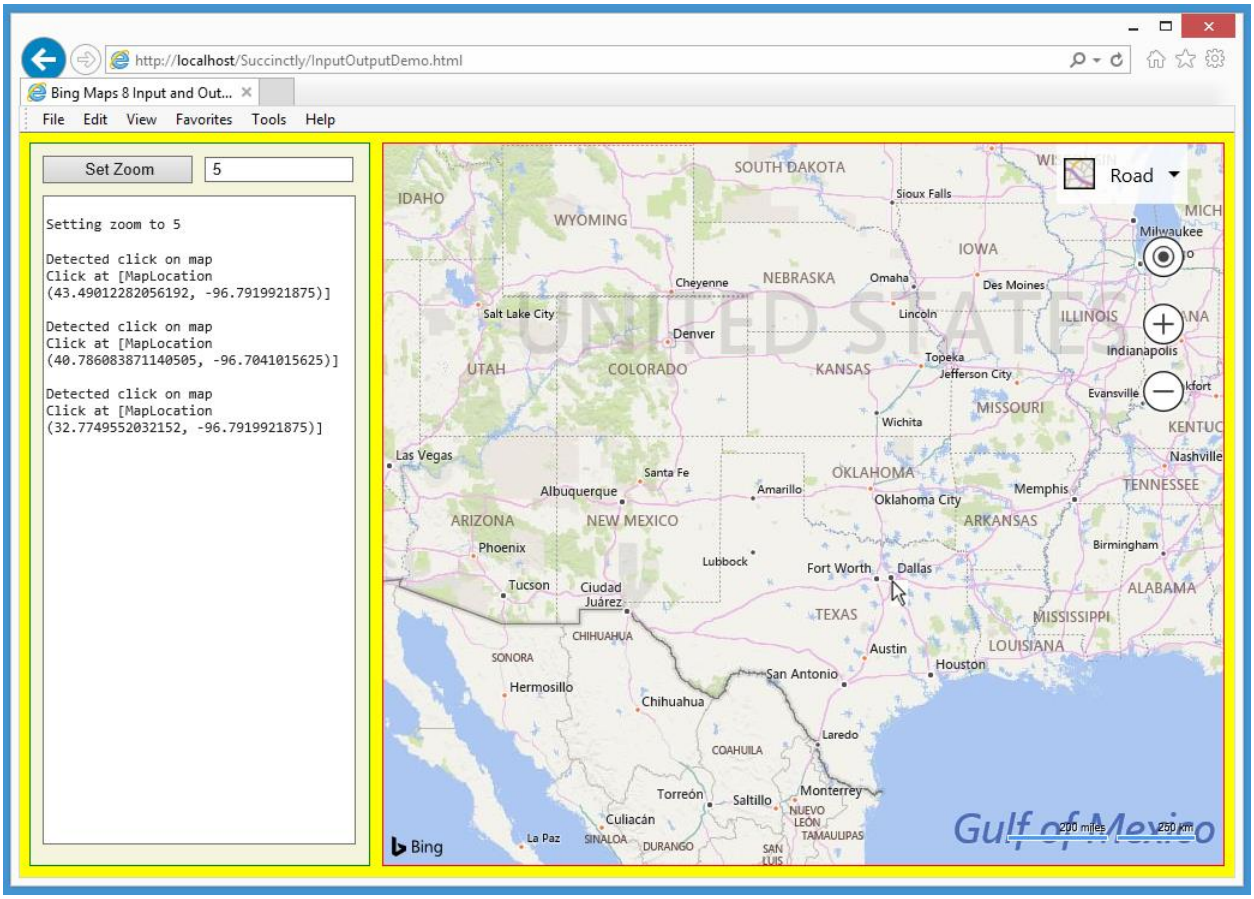

Figure 9: User Input and Output Demo

The demo web application in Figure 9 shows some basic examples of user input and output with a Bing Maps V8 map. The demo initially loads a map centered at (35.00, -100.00) with a zoom level of 2 (showing most of the world). Here, the user clicked the HTML button control labeled "Set Zoom" and control was transferred to a JavaScript function that fetched the value in the HTML input text box control (5) and programmatically set the map zoom level to that value.

Next, the user moved the mouse to Sioux Falls, South Dakota, and clicked the map. The application caught the click event, determined where the event occurred, and displayed the location in an HTML <textarea> tag. The user next clicked Lincoln, Nebraska, then Dallas, Texas. Notice the latitude values decreased from approximately 43 to 40 to 32.

The demo web application is named InputOutputDemo.html and is defined in a single file.

Code Listing 2: InputOutputDemo.html

```
<!DOCTYPE html>
<!-- InputOutputDemo.html -->

<html>
  <head>
    <title>Bing Maps 8 Input and Output</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <script type='text/javascript'>

    var map = null;

    function GetMap()
    {
      var options = {
        credentials: "Anw _ _ _ 3xt",
        center: new Microsoft.Maps.Location(35.00, -100.00),
        mapTypeId: Microsoft.Maps.MapTypeId.road,
        zoom: 2,
        enableClickableLogo: false,
        showCopyright: false
      };

      var mapDiv = document.getElementById("mapDiv");
      map = new Microsoft.Maps.Map(mapDiv, options);
      Microsoft.Maps.Events.addHandler(map, 'click', MapClicked);
    }

    function WriteLn(txt)
    {
      var existing = msgArea.value;
      msgArea.value = existing + txt + "\n";
    }

    function Button1_Click()
    {
      var z = parseInt(textbox1.value);
      WriteLn("\nSetting zoom to " + z);
      map.setView({ zoom: z });
    }

    function MapClicked(e)
    {
      var tt = e.targetType;
      WriteLn("\nDetected click on " + tt.toString());
      var loc = e.location;
      WriteLn("Click at " + loc);
    }

    </script>
  </head>

  <body style="background-color:yellow">
```

```
    <div id='controlPanel' style="float:left; width:262px; height:580px;
      border:1px solid green; padding:10px; background-color: beige">
      <input id="button1" type='button' style="width:125px;"
        value=' Set Zoom' onclick="Button1_Click();"></input>
      <div style="width:2px; display:inline-block"></div>
      <input id="textbox1" type='text' size='15' value=' 5'>
        </input><br/>
      <span style="display:block; height:10px"></span>

      <textarea id='msgArea' rows="38" cols="36" style="font-family:Consolas;
        font-size:12px"></textarea>
    </div>

    <div style="float:left; width:10px; height:600px"></div>

    <div id='mapDiv' style="float:left; width:700px; height:600px;
      border:1px solid red;"></div>
    <br style="clear: left;" />


   <script type='text/javascript'
     src='http://www.bing.com/api/maps/mapcontrol?callback=GetMap'
     async defer>
   </script>

  </body>
</html>
```

The demo application sets up an HTML page that consists of three main sections—an area for normal HTML controls, a dummy spacer, and the map. This is a crude structure, but it does point out that you can place a Bing Maps map anywhere on a webpage.

The three main sections are decorated with a **`style="float:left"`** annotation, which means that the map area will move if the user makes the width of the webpage small. The somewhat cryptic **`<br style="clear:left;" />`** tag keeps the three areas on the same row if the webpage is wide enough.

The left-most section contains a button control, a textbox control, and a <textarea> control. As you'll see shortly, you can use any HTML controls to provide user input to a map. The <textarea> element is used to display output messages. There are many alternatives you can use, including elements and <pre> elements and their innerHTML property.

The demo displays messages using a program-defined **`WriteLn()`** function defined as:

```
function WriteLn(txt)
{
  var existing = msgArea.value;
  msgArea.value = existing + txt + "\n";
}
```

This technique of grabbing the existing text and appending new text is simple and effective so long as the amount of text doesn't get too large.

The demo creates the map with function **GetMap()**. Note that I often capitalize the names of my program-defined functions in order to distinguish them from the library and built-in JavaScript functions.

```
var map = null;  // script-global map object
function GetMap()
{
  var options = {
    credentials: "Anw (etc) 3xt",
    center: new Microsoft.Maps.Location(35.00, -100.00),
    mapTypeId: Microsoft.Maps.MapTypeId.road,
    zoom: 2,
    enableClickableLogo: false,
    showCopyright: false
  };

  var mapDiv = document.getElementById("mapDiv");
  map = new Microsoft.Maps.Map(mapDiv, options);
  Microsoft.Maps.Events.addHandler(map, 'click', MapClicked);
}
```

Recall that the Map constructor accepts an options object that can contain both ViewOptions and MapOptions properties. Here the initial zoom level is arbitrarily set to 2.

After the Map constructor is called, the handler for the click event is modified using the **Events.addHandler()** function so that when a user clicks anywhere on the **map** object, control will be transferred to a function named **MapClicked()**. Alternatively, code can be supplied directly using an anonymous function this way:

```
Microsoft.Maps.Events.addHandler(map, 'click',
  function () { alert('Ouch!'); });
```

Function **MapClicked()** is defined as:

```
function MapClicked(e)
{
  var tt = e.targetType;
  WriteLn("\nDetected click on " + tt.toString());
  var loc = e.location;
  WriteLn("Click at " + loc);
}
```

The **e** object represents the click sender. In this example, the **targetType** property isn't needed because the only object that passes control to the **MapClicked()** function is the map. But when multiple objects transfer control to a single function, you can write code such as:

```
if (e.targetType == "map") {
  // do something
}
else if (e.targetType == "pushpin") {
  // something else
```

In Bing Maps V8, you can get a click lat-lon location using the **e.location** property. This is a nice new feature—in previous versions you were required to get the page pixel position of the click and convert that position to a **Location** object with code this way:

```
var pt = new Microsoft.Maps.Point(e.pageX, e.pageY,
  Microsoft.Maps.PixelReference.page);
var loc = e.target.tryPixelToLocation(pt,
  Microsoft.Maps.PixelReference.page);
WriteLn("\nMouse click at: " + loc);
```

By the way, the Internet is littered with incorrect code that resembles this:

```
var point = new Microsoft.Maps.Point(e.getX(), e.getY());  // Wrong
var loc = e.target.tryPixelToLocation(point);
```

This code gives an incorrect lat-lon position and also gives a different result each time the map is panned.

When a user clicks the button labeled Set Zoom, control is transferred to function **Button1_Click()**, which is defined this way:

```
function Button1_Click()
{
  var z = parseInt(textbox1.value);  // get new zoom value as integer
  WriteLn("\nSetting zoom to " + z);
  map.setView({ zoom: z });
}
```

In summary, user input and information can be easily exchanged between a Bing Maps map and HTML controls. In fact, previous versions of Bing Maps referred to the entire library as the Bing Maps Control because, conceptually, a map is a large, custom HTML control. You can modify event handlers on a map or objects on a map using the **Microsoft.Maps.Events.addHandler()** function.

Resources

Many of the examples in this book are extensions of those in the interactive SDK at: http://www.bing.com/api/maps/sdk/mapcontrol/isdk#overview.

For detailed information about the Events class, see: https://msdn.microsoft.com/en-us/library/mt750279.aspx.

Chapter 2 Fundamental Techniques

This chapter presents some of the most commonly used features of the Bing Maps V8 library. In section 2.1 you'll learn how to work with pushpins. In particular, you'll learn how to create pushpins on the fly programmatically by using two different techniques: the HTML5 canvas and scalable vector graphics (SVG).

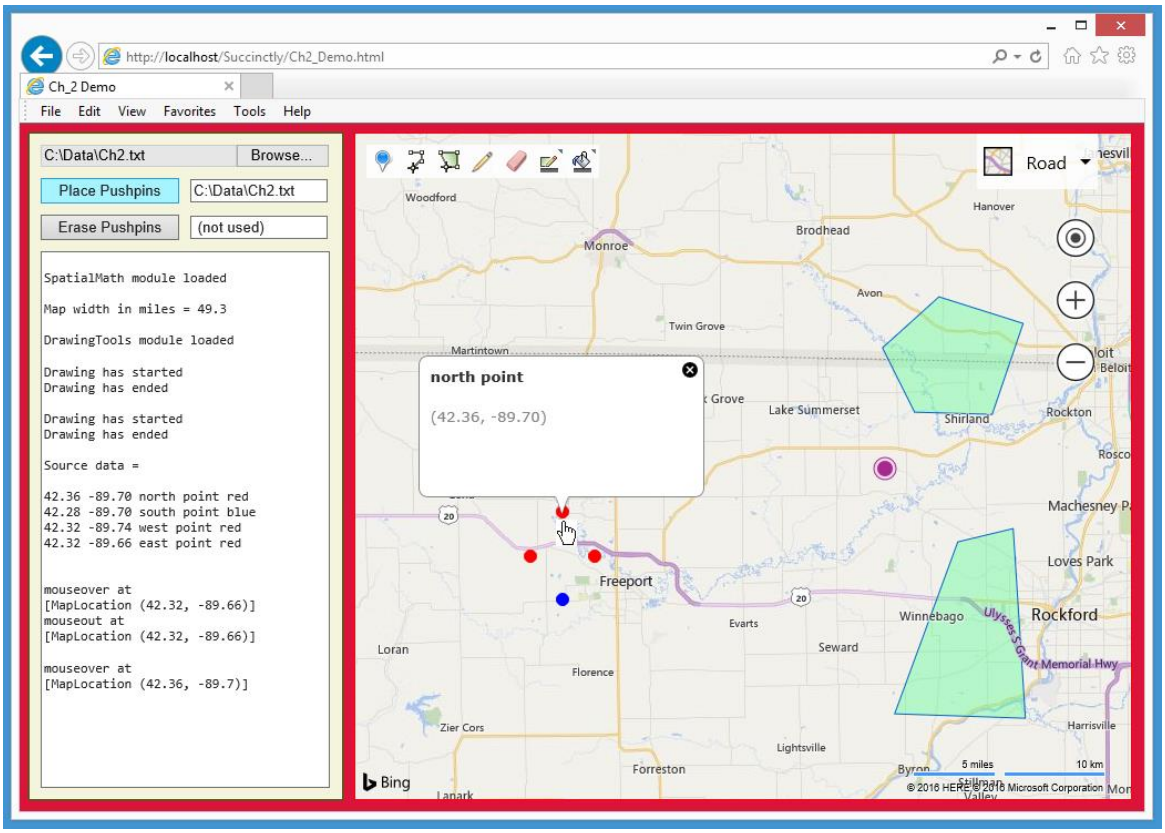

Figure 10: Fundamental Techniques Demo

Pushpins are often accompanied by **`Infobox`** objects, and section 2.1 explains how to create and manipulate them in conjunction with events such as mouseover and mouseout.

In section 2.2, you'll learn how to use the Maps.DrawingTools module to add a drawing control to a map, which allows users to interactively add pushpins, polylines, and polygons to a map. You'll also learn how to access these objects programmatically.

In section 2.3, you'll learn how to work with geomath functions in the Maps.SpatialMath module and how to use them to calculate geoproperties such as the location of the map center and the width and height of the current map view.

2.1 Pushpins and Infoboxes

Two of the most common ways to add information to a Bing Maps V8 map are to create **`pushpin`** objects and **`Infobox`** objects. They're often used together. Bing Maps gives you two ways to create static, custom pushpin icons and two ways to create dynamic, custom icons. The basic **`Infobox`** object is quite flexible and can display text and graphics information.

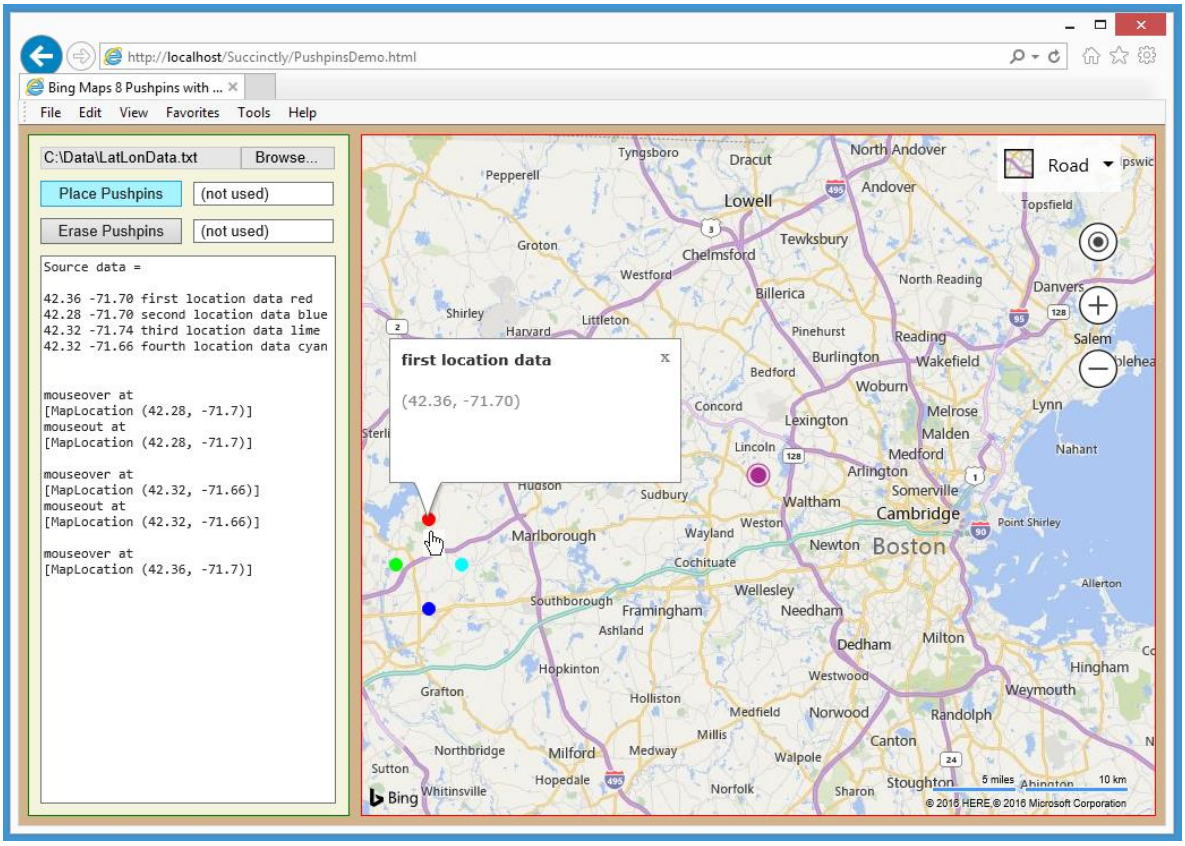

Figure 11: Pushpins and Infoboxes Demo

The demo web application depicted in Figure 11 shows some examples of **`pushpin`** and **`Infobox`** objects. The demo initially loads a map centered near Boston and also places a default-style large purple pushpin with a radius of about 10 pixels at the map center. The user clicked the "Browse" button HTML file control in the upper-left corner and pointed the Choose File dialog to a file named LatLonData.txt in a local C:\Data directory.

When the button labeled "Place Pushpins" was clicked, the webpage read and echoed the text file and used the data to dynamically create four different-colored pushpins. Then the user moved the mouse cursor over and away from some of the pushpins. Note that as each pushpin is passed, information about the pushpin appears in a default-style **`Infobox`** object, and when the cursor leaves the pushpin, the **`Infobox`** object automatically disappears.

The demo web application is named PushpinsDemo.html and is defined in a single file.

Code Listing 3: PushpinsDemo.html

```
<!DOCTYPE html>
<!-- PushpinsDemo.html -->

<html>
  <head>
    <title>Bing Maps 8 Pushpins with Infoboxes</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>

    <script type='text/javascript'>

    var map = null;
    var pushpins = [];
    var infobox = null;
    var ppLayer = null;

    function GetMap()
    {
      var options = {
        credentials: "Anw _ _ _ 3xt",
        center: new Microsoft.Maps.Location(42.40, -71.30),
        mapTypeId: Microsoft.Maps.MapTypeId.road,
        zoom: 10,
        enableClickableLogo: false,
        showTermsLink: false
      };

      var mapDiv = document.getElementById("mapDiv");
      map = new Microsoft.Maps.Map(mapDiv, options);

      var ibOptions = { visible: false,
        offset: new Microsoft.Maps.Point(0, 0) };
      var ibLoc = new Microsoft.Maps.Location(0, 0);
      infobox = new Microsoft.Maps.Infobox(ibLoc, ibOptions);
      infobox.setMap(map);

      ppLayer = new Microsoft.Maps.Layer();
      var cpp = new Microsoft.Maps.Pushpin(map.getCenter(), null);
      ppLayer.add(cpp);
      map.layers.insert(ppLayer);
    }

    function WriteLn(txt)
    {
      var existing = msgArea.value;
      msgArea.value = existing + txt + "\n";
    }

    function LatLonStr(loc)
    {
      var s = "(" + Number(loc.latitude).toFixed(2) + ", " +
        Number(loc.longitude).toFixed(2) + ")";
      return s;
```

```
}

function Button1_Click()
{
  var f = file1.files[0];
  var reader = new FileReader();
  reader.onload = function(e) {
    WriteLn("Source data = \n");
    var lines = reader.result.split('\n');
    for (var i = 0; i < lines.length; ++i) {
      var line = lines[i];
      var tokens = line.split(',');
      WriteLn(tokens[0] + " " + tokens[1] + " " +
        tokens[2] + " " + tokens[3]);
      var loc = new Microsoft.Maps.Location(tokens[0], tokens[1]);
      var ppOptions = { icon: CreateCvsDot(6, tokens[3]),
        anchor: new Microsoft.Maps.Point(6, 6),
        subTitle: tokens[2] };

      var pp = new Microsoft.Maps.Pushpin(loc, ppOptions);
      pushpins[i] = pp;
      Microsoft.Maps.Events.addHandler(pp, 'mouseover', ShowInfobox);
      Microsoft.Maps.Events.addHandler(pp, 'mouseout', HideInfobox);
    }
    ppLayer.add(pushpins);
    map.layers.insert(ppLayer);
    WriteLn("");
  }
  reader.readAsText(f);
}

function CreateCvsDot(radius, clr)
{
  var c = document.createElement('canvas');
  c.width = 2 * radius;
  c.height = 2 * radius;
  var ctx = c.getContext("2d");
  ctx.beginPath();
  ctx.arc(radius,radius, radius, 0, 2 * Math.PI);
  ctx.fillStyle = clr;
  ctx.fill();
  return c.toDataURL();
}

function Button2_Click()
{
  ppLayer.clear();
}

function ShowInfobox(e)
{
  var loc = e.target.getLocation();
  WriteLn('\nmouseover at \n' + loc);
  infobox.setLocation(loc);
```

```
      infobox.setOptions( { visible: true, title: e.target.getSubTitle(),
        description: LatLonStr(loc)
      });
    }

    function HideInfobox(e)
    {
      WriteLn('mouseout at \n' + e.target.getLocation());
      infobox.setOptions({ visible: false });
    }

    </script>
  </head>

  <body style="background-color:tan">
    <div id='controlPanel' style="float:left; width:262px; height:580px;
      border:1px solid green; padding:10px; background-color: beige">

      <input type="file" id="file1" size="24"></input>
      <span style="display:block; height:10px"></span>

      <input id="button1" type='button' style="width:125px;"
        value='Place Pushpins' onclick="Button1_Click();"></input>
      <div style="width:2px; display:inline-block"></div>
      <input id="textbox1" type='text' size='15' value=' (not used)'>
        </input><br/>
      <span style="display:block; height:10px"></span>

      <input id="button2" type='button' style="width:125px;"
        value='Erase Pushpins' onclick="Button2_Click();"></input>
      <div style="width:2px; display:inline-block"></div>
      <input id="textbox2" type='text' size='15' value=' (not used)'>
        </input><br/>
      <span style="display:block; height:10px"></span>

      <textarea id='msgArea' rows="34" cols="36" style="font-family:Consolas;
        font-size:12px"></textarea>
    </div>

    <div style="float:left; width:10px; height:600px"></div>
    <div id='mapDiv' style="float:left; width:700px; height:600px;
      border:1px solid red;"></div>
    <br style="clear: left;" />

    <script type='text/javascript'
      src='http://www.bing.com/api/maps/mapcontrol?callback=GetMap'
      async defer>
    </script>

  </body>
</html>
```

The demo application declares four script-global objects:

```
var map = null;
var pushpins = [];
var infobox = null;
var ppLayer = null;
```

Object **pushpins** is an array that will hold all pushpins. Object **Infobox** is a single **Infobox** object that will be shared by all the pushpins. Alternatively, we can define an array of **Infobox** objects, but, as you'll see, for this scenario all we need is a single **Infobox**.

A function **GetMap()** creates the **map** object. The map options object does not use the undocumented **showCopyright** property for the map properties but instead sets the **showTermsLink** to **false** so that a small hyperlink to legal information will not be displayed in the lower-right corner of the map:

```
function GetMap()
{
  var options = {
    credentials: "Anw (etc) 3xt",
    center: new Microsoft.Maps.Location(42.40, -71.30),  // Boston
    mapTypeId: Microsoft.Maps.MapTypeId.road,
    zoom: 10,
    enableClickableLogo: false,
    showTermsLink: false  // is true by default
  };
  var mapDiv = document.getElementById("mapDiv");
  map = new Microsoft.Maps.Map(mapDiv, options);
. . .
```

Function **GetMap()** initializes the single **Infobox** object this way:

```
var ibOptions = { visible: false,
  offset: new Microsoft.Maps.Point(0, 0) };
var ibLoc = new Microsoft.Maps.Location(0, 0);
infobox = new Microsoft.Maps.Infobox(ibLoc, ibOptions);
infobox.setMap(map);
```

Unlike pushpins, an **Infobox** object lives "over a map" rather than "on a map," so to speak. Here an **ibOptions** object is set so that the Infobox is not initially visible, and it will be located at latitude-longitude (0, 0), which is the intersection of the equator and the prime meridian (off the west coast of Africa). The **setMap()** function is used to place the (currently invisible) **Infobox** object over the map. Notice the calling syntax is **infobox.SetMap(map)** rather than **map.setInfobox(infobox)**, which might seem a bit more natural.

Function **GetMap()** finishes initialization by preparing a visualization layer for the pushpins that will be dynamically created when the data file is read, and it creates and places a single pushpin at map center:

```
. . .
  ppLayer = new Microsoft.Maps.Layer();
  var cpp = new Microsoft.Maps.Pushpin(map.getCenter(), null);
  ppLayer.add(cpp);
  map.layers.insert(ppLayer);
} // end GetMap()
```

Pushpins, polygons, and other visual objects can now be placed in separate visual layers—an important new feature of Bing Maps V8. In this case, a layer object named **ppLayer** is created for the center pushpin and the four pushpins that will represent the data in the text file.

There are four steps to placing a pushpin on a map: 1) create a layer 2) create a pushpin 3) add the pushpin to the layer and 4) insert the layer into the map. The demo places all five pushpins in a single layer. An alternative is to create two layers, one for the center pushpin and one for the data-file pushpins.

The **null** value passed to the Pushpin constructor can be loosely interpreted to mean “make a default, round purple pushpin with radius 10 pixels.”

In Bing Maps V7 and earlier versions, all visual objects were placed into a single global Entities array object. That code would look like this:

```
var cpp = new Microsoft.Maps.Pushpin(map.getCenter(), null);
map.entities.push(cpp);
```

Bing Maps V8 supports the entities approach for backward compatibility. However, except for the simplest map scenarios, I believe that using the new layer approach is preferable.

Function **Button1_Click()** reads a text file of location information, parses each line, and creates a custom-colored pushpin. The function code is:

```
function Button1_Click()
{
  var f = file1.files[0];
  var reader = new FileReader();
  reader.onload = function(e) {
    WriteLn("Source data = \n");
    var lines = reader.result.split('\n');
    for (var i = 0; i < lines.length; ++i) {
      // create a pushpin, add to global pushpins[] array
    }
    ppLayer.add(pushpins);
    map.layers.insert(ppLayer);
    WriteLn("");
  }
  reader.readAsText(f);
}
```

Object **file1** is the HTML File control. Because it can point to multiple files, the first file is selected as **files[0]**. The **FileReader** object reads asynchronously. Notice that you first must define an anonymous function associated with the **onload** event in order to provide instructions for later, after the file has been read into memory, then you issue a single instruction to read the file using the **readAsText()** function.

When the text file has been read into object **reader.result**, it is broken into an array of lines using the built-in JavaScript **split()** function. Each line is processed this way:

```
for (var i = 0; i < lines.length; ++i) {
  var line = lines[i];
  var tokens = line.split(',');

  WriteLn(tokens[0] + " " + tokens[1] + " " +
    tokens[2] + " " + tokens[3]);

  var loc = new Microsoft.Maps.Location(tokens[0], tokens[1]);

  var ppOptions = { icon: CreateCvsDot(6, "orangered"),
    anchor: new Microsoft.Maps.Point(6, 6),
    subTitle: tokens[2] };

  var pp = new Microsoft.Maps.Pushpin(loc, ppOptions);
  pushpins[i] = pp;
  Microsoft.Maps.Events.addHandler(pp, 'mouseover', ShowInfobox);
  Microsoft.Maps.Events.addHandler(pp, 'mouseout', HideInfobox);
}
```

The contents of the source LatLonData.txt file are:

```
42.36,-71.70,first location data,red
42.28,-71.70,second location data,blue
42.32,-71.74,third location data,lime
42.32,-71.66,fourth location data,cyan
```

Each comma-delimited line is broken into its four parts. The first two parts are used to create a **Location** object for the pushpin. The key to creating a custom pushpin is to set its option properties. The icon property defines what the pushpin looks like, and the demo calls a program-defined function **CreateCvsDot()** to create a circle with radius 6 pixels and the specified color. I will explain this function shortly. The anchor property controls where the pushpin will be placed, in pixels, relative to its location property where an anchor of (0, 0) is the top-left corner of the icon.

A pushpin can have a **title**, a **subTitle**, and a **text** property. Here the **subTitle** is set to the third field in the text file, but it will not be displayed because no **title** has been set.

After the pushpin's options and location have been set, it is created as an object **pp** using the **Pushpin()** constructor and added to the global **pushpins** array. I could have written **pushpins.push(pp)** instead, but that's a lot of p's in one line of code. Each pushpin has its mouseover and mouseout event handlers modified so as to display or hide an **Infobox** object.

Function **ShowInfoBox()** is defined:

```
function ShowInfobox(e)
{
  var loc = e.target.getLocation();
  WriteLn('\nmouseover at \n' + loc);
  infobox.setLocation(loc);
  infobox.setOptions( { visible: true,
    title: e.target.getSubTitle(),
    description: LatLonStr(loc)
  });
}
```

Recall that the **GetMap()** function created the single **Infobox** object, but it is not visible. When a user moves the mouse cursor over a pushpin, the location of the pushpin is fetched on the fly using the **getLocation()** function. Because the **Infobox** object already exists, setting its **visible** property to **true** is all we must do to make the object appear.

The description property of the **Infobox** object is set to the location of the associated pushpin. The demo uses a short helper function **LatLonStr()** to format the location to two decimals:

```
function LatLonStr(loc)
{
  var s = "(" + Number(loc.latitude).toFixed(2) + ", " +
    Number(loc.longitude).toFixed(2) + ")";
  return s;
}
```

A slightly more flexible approach would be to pass the number of decimals as a parameter to the function. Function **HideInfoBox()** is very simple and merely sets the **visible** property back to false:

```
function HideInfobox(e)
{
  WriteLn('mouseout at \n' + e.target.getLocation());
  infobox.setOptions({ visible: false });
}
```

To recap, the demo creates a single **Infobox** object that is not initially visible. When a mouseover event on a pushpin fires, the pushpin's location is fetched, information stored in the pushpin is retrieved, and the **Infobox** object is made visible. When the mouseout event on a pushpin fires, the **Infobox** is made not visible.

One of my favorite features of the Bing Maps library is its ability to programmatically create pushpin icons. You can create icons using an HTML canvas or by using an SVG. The demo uses the HTML canvas technique. The creation function is defined this way:

```
function CreateCvsDot(radius, clr)
{
  var c = document.createElement('canvas');
  c.width = 2 * radius;

  c.height = 2 * radius;
  var ctx = c.getContext("2d");
  ctx.beginPath();
  ctx.arc(radius,radius, radius, 0, 2 * Math.PI);
  ctx.fillStyle = clr;
  ctx.fill();
  return c.toDataURL();
}
```

You aren't limited to creating circles for a custom pushpin icon, and an HTML canvas can be used to create just about any kind of image you can imagine, although relatively simple geometric shapes such as circles, ovals, triangles, and rectangles are most common. After a canvas object is created, the rather weirdly named **`toDataURL()`** function is used to create a PNG format image from the canvas.

I love the ability to dynamically create different-styled pushpins (and other objects, as you'll see). Without this ability, an application would have to create dozens or maybe even hundreds of static PNG images in order to account for different situations, then use code logic to determine which image to use.

Function **`Button2_Click()`** erases all pushpins with the single statement:

```
function Button2_Click()
{
  ppLayer.clear();
}
```

Here, both the default-style center pushpin and the data pushpins are erased. However, if two layers had been used, the demo could have targeted an individual collection of pushpins along the lines of:

```
centerPpLayer.clear();
dataPpLayer.clear();
```

In summary, a default pushpin icon is purple with a radius of about 10 pixels. There are four ways to create a custom image icon for a pushpin: by dynamically using an HTML canvas (as shown in the demo) or an SVG string, and by statically using an ordinary image file or a Base64-encoded object. **`Infobox`** objects are most often used to display text information, but they are very flexible.

Resources

For detailed information about pushpin options, see:
https://msdn.microsoft.com/en-us/library/mt712673.aspx.

For detailed information about **`Infobox`** object options, see:
https://msdn.microsoft.com/en-us/library/mt712658.aspx.

A good resource for information about HTML canvas images is at:
http://www.w3schools.com/tags/ref_canvas.asp.

2.2 Interactive drawing

The Bing Maps V8 library has a DrawingTools module that allows you to place a drawing control on a map. The drawing control lets users interactively add pushpins, lines, and polygons. These shapes aren't just art; they are objects that can be programmatically accessed and manipulated.

The demo web application in Figure 12 initially loads a map centered near Kansas City with a zoom level set to 10.

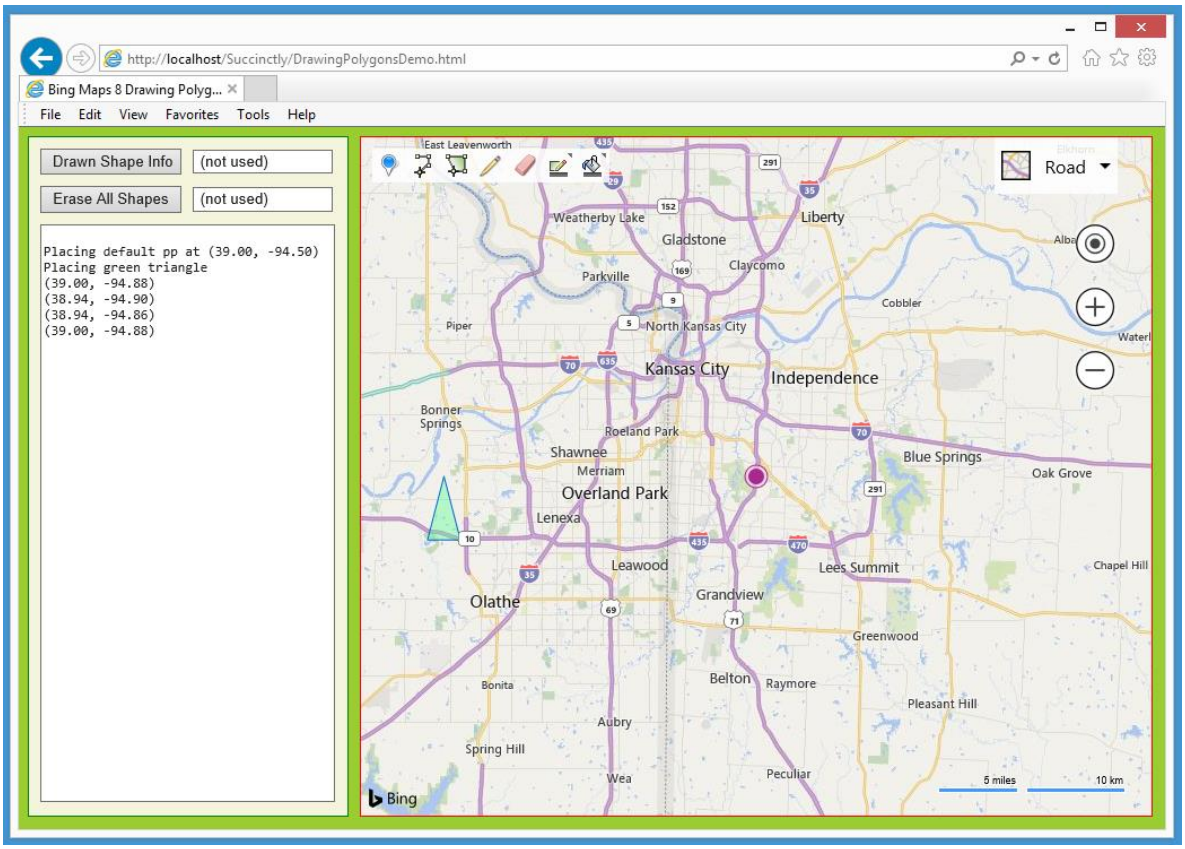

Figure 12: Initial Map View for Drawing Polygons Demo

During map initialization, the application places a default-style purple pushpin at the center of the map to act as a point of reference. The application also places a green, three-sided polygon to the left of center using hard-coded locations.

The drawing control is visible in the upper-left corner. There are seven icons. Users can add a pushpin, add a polyline, add a polygon, edit a drawn shape, erase a drawn shape, color a drawn shape, or set the color to use with a new shape.

In this example, after the map loaded, the user added a default-style (green) pushpin to the right of the city of Independence and drew a default-style, five-sided polygon to the left of center.

Next, the user clicked the right-most icon on the drawing control in order to select the color (purple) and transparency for a new shape.

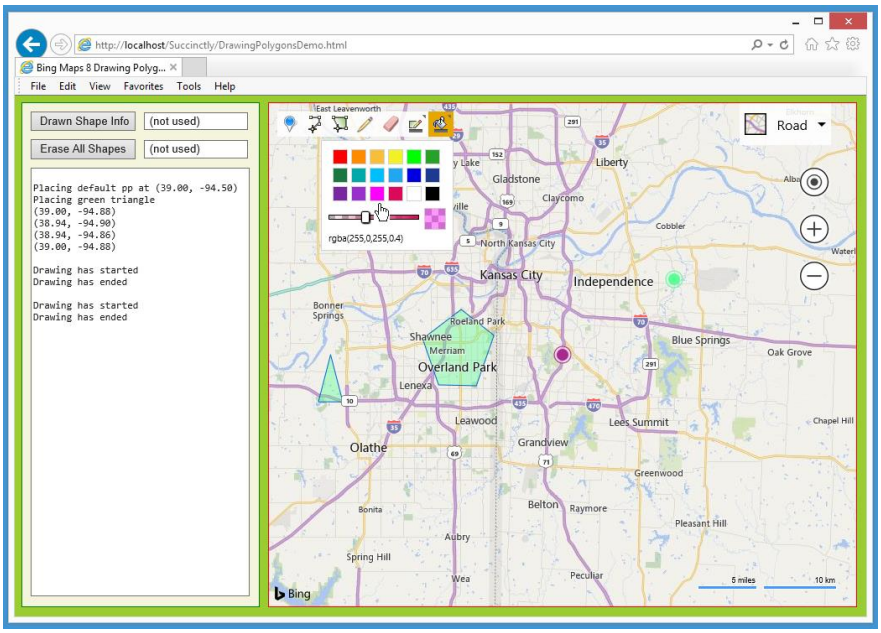

Figure 13: Using the Drawing Control Color Selector

Next, the user clicked the third icon to draw a polygon, then added a three-sided polygon with top-most vertex at the map center. A click on the top button control programmatically fetched information about the most recently drawn shape and displayed that information.

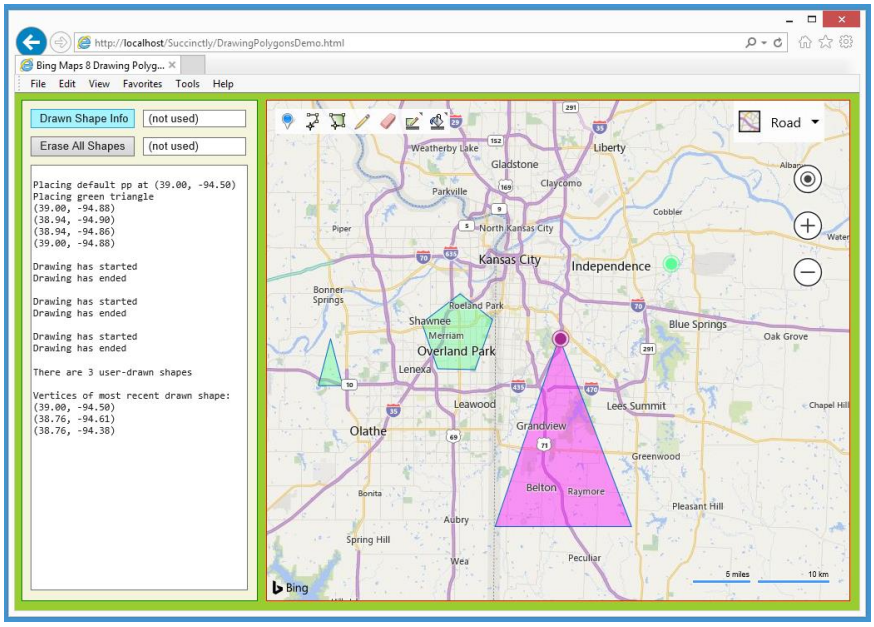

Figure 14: Retrieving Information about Interactively Created Shapes

The demo application is named DrawingPolygonsDemo.html and is defined in a single file.

Code Listing 4: DrawingPolygonsDemo.html

```
<!DOCTYPE html>
<!-- DrawingPolygonsDemo.html -->

<html>
  <head>
    <title>Bing Maps 8 Drawing Polygons</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>

    <script type='text/javascript'>

    var map = null;
    var ppLayer = null;

    var drawMgr = null;
    var drawnShapes = [];

    function GetMap()
    {
      var options = {
        credentials: "Anw _ _ _ 3xt",
        center: new Microsoft.Maps.Location(39.00, -94.50),
        mapTypeId: Microsoft.Maps.MapTypeId.road,
        zoom: 10,
        enableClickableLogo: false,
        showCopyright: false
      };

      var mapDiv = document.getElementById("mapDiv");
      map = new Microsoft.Maps.Map(mapDiv, options);

      ppLayer = new Microsoft.Maps.Layer();
      WriteLn("\nPlacing default pp at " + LatLonStr(map.getCenter()));
      var cpp = new Microsoft.Maps.Pushpin(map.getCenter(), null);
      ppLayer.add(cpp);

      WriteLn("Placing green triangle");
      var v0 = new Microsoft.Maps.Location(39.00, -94.88);
      var v1 = new Microsoft.Maps.Location(38.94, -94.90);
      var v2 = new Microsoft.Maps.Location(38.94, -94.86);
      var arr = [];
      arr.push(v0); arr.push(v1); arr.push(v2);
      var triangle = new Microsoft.Maps.Polygon(arr, null);
      ppLayer.add(triangle);
      map.layers.insert(ppLayer);

      var triangleLocs = triangle.getLocations();
      for (var i = 0; i < triangleLocs.length; ++i) {
        WriteLn(LatLonStr(triangleLocs[i]));
      }

      Microsoft.Maps.loadModule('Microsoft.Maps.DrawingTools',
        function() {
```

```
      var tools = new Microsoft.Maps.DrawingTools(map);
      tools.showDrawingManager(AddDrawControlEvents);
    });
}

function AddDrawControlEvents(manager)
{
  Microsoft.Maps.Events.addHandler(manager, 'drawingStarted',
    function() { WriteLn('\nDrawing has started');  });
  Microsoft.Maps.Events.addHandler(manager, 'drawingEnded',
    function(e) { WriteLn('Drawing has ended'); });
  drawMgr = manager;  // Save for get-shapes-info function.
}

function WriteLn(txt)
{
  var existing = msgArea.value;
  msgArea.value = existing + txt + "\n";
}

function LatLonStr(loc)
{
  var s = "(" + Number(loc.latitude).toFixed(2) + ", " +
    Number(loc.longitude).toFixed(2) + ")";
  return s;
}

function Button1_Click()
{
  drawnShapes = drawMgr.getPrimitives();
  var numShapes = drawnShapes.length;
  var mostRecent = drawnShapes[numShapes-1];

  WriteLn("\nThere are " + numShapes + " user-drawn shapes");
  var isPoly = (mostRecent instanceof Microsoft.Maps.Polygon);
  var isPushpin = (mostRecent instanceof Microsoft.Maps.Pushpin);

  if (isPoly == true) {
    WriteLn("\nVertices of most recent drawn shape: ");
    var vertices = mostRecent.getLocations();
    for (var i = 0; i < vertices.length; ++i) {
      WriteLn(LatLonStr(vertices[i]));
    }
    WriteLn("");
  }
  else if (isPushpin == true) {
    WriteLn("\nLocation of most recent pushpin: ");
    var loc = mostRecent.getLocation();
    WriteLn("loc = " + loc);
  }
}

function Button2_Click()
{
```

```
      drawMgr.clear();  // drawn shapes
      ppLayer.clear();  // other shapes
    }

    </script>
  </head>

  <body style="background-color:#9ACD32">
    <div id='controlPanel' style="float:left; width:262px; height:580px;
      border:1px solid green; padding:10px; background-color: beige">

      <input id="button1" type='button' style="width:125px;"
        value='Drawn Shape Info' onclick="Button1_Click();"></input>
      <div style="width:2px; display:inline-block"></div>
      <input id="textbox1" type='text' size='15'
        value=' (not used)'> </input><br/>
      <span style="display:block; height:10px"></span>

      <input id="button2" type='button' style="width:125px;"
        value='Erase All Shapes' onclick="Button2_Click();"></input>
      <div style="width:2px; display:inline-block"></div>
      <input id="textbox2" type='text' size='15'
        value=' (not used)'></input><br/>
      <span style="display:block; height:10px"></span>

      <textarea id='msgArea' rows="36" cols="36" style="font-family:Consolas;
        font-size:12px"></textarea>
    </div>

    <div style="float:left; width:10px; height:600px"></div>
    <div id='mapDiv' style="float:left; width:700px; height:600px;
      border:1px solid red;"></div>
    <br style="clear: left;" />

    <script type='text/javascript'
      src='http://www.bing.com/api/maps/mapcontrol?callback=GetMap'
      async defer>
    </script>

  </body>
</html>
```

The demo begins by setting up four script-global objects:

```
var map = null;
var ppLayer = null;

var drawMgr = null;
var drawnShapes = [];
```

The **ppLayer** object will hold pushpins (in this case just the one pushpin at map center) that are programmatically added. Note that you do not create a **Layer** object for shapes that are interactively added using the drawing control. The **drawMgr** object will be created when the drawing control is added to the map, and it will be used to access drawn shapes. The **drawnShapes** array is declared global but is used by only one function, which means **drawnShapes** could have been declared local to that function.

The **map** object is created in the usual way:

```
function GetMap()
{
  var options = {
    credentials: "Anw _ _ _ 3xt",
    center: new Microsoft.Maps.Location(39.00, -94.50),
    mapTypeId: Microsoft.Maps.MapTypeId.road,
    zoom: 10,
    enableClickableLogo: false,
    showCopyright: false
  };
  var mapDiv = document.getElementById("mapDiv");
  map = new Microsoft.Maps.Map(mapDiv, options);
. . .
```

Note that no special initialization is needed if you're going to use the drawing control. Next, the center pushpin is created but not yet placed on the map:

```
ppLayer = new Microsoft.Maps.Layer();
WriteLn("\nPlacing default pp at " + LatLonStr(map.getCenter()));
var cpp = new Microsoft.Maps.Pushpin(map.getCenter(), null);
ppLayer.add(cpp);
```

The demo application creates a hard-coded, three-sided default-style polygon and adds it to the **ppLayer** layer:

```
WriteLn("Placing green triangle");
var v0 = new Microsoft.Maps.Location(39.00, -94.88);
var v1 = new Microsoft.Maps.Location(38.94, -94.90);
var v2 = new Microsoft.Maps.Location(38.94, -94.86);
var arr = [];
arr.push(v0); arr.push(v1); arr.push(v2);
var triangle = new Microsoft.Maps.Polygon(arr, null);
ppLayer.add(triangle);
map.layers.insert(ppLayer);
```

In a nondemo scenario, you'd likely want to create a separate **Layer** object for the polygon rather than using the pushpin layer.

The demo programmatically fetches information about the just-added polygon to point out an important idea:

```
var triangleLocs = triangle.getLocations();
for (var i = 0; i < triangleLocs.length; ++i) {
  WriteLn(LatLonStr(triangleLocs[i]));
}
```

If you look at the figures at the beginning of this section, you'll notice that the triangle polygon is created with three **Location** objects, but the **getLocations()** function returns four locations. The first location is repeated. Some shapes are closed by duplicating the first location, and some shapes are not closed. This can cause you a lot of trouble if you're not careful.

Map initialization concludes by adding the drawing control:

```
. . .
  Microsoft.Maps.loadModule('Microsoft.Maps.DrawingTools',
    function() {
      var tools = new Microsoft.Maps.DrawingTools(map);
      tools.showDrawingManager(AddDrawControlEvents);
    });
}
```

The anonymous callback function adds the control to the map and calls a program-defined function named **AddDrawControlEvents()**. This function is defined as:

```
function AddDrawControlEvents(manager)
{
  Microsoft.Maps.Events.addHandler(manager, 'drawingStarted',
    function() { WriteLn('\nDrawing has started');  });
  Microsoft.Maps.Events.addHandler(manager, 'drawingEnded',
    function(e) { WriteLn('Drawing has ended'); });
  drawMgr = manager;
}
```

The function adds code for writing a message when a drawing has started or ended. Notice that the anonymous function associated with the drawingEnded event has an **e** argument that is not used. The **e** argument represents the just-drawn shape, so if you want to perform some action on a user-drawn shape immediately after it has been created, you can access the shape through the **e** object. The global **drawMgr** object is saved—this is how user-drawn shapes can be accessed later through another function.

The button control labeled "Drawn Shape Info" is associated with function **Button1_Click()**:

```
function Button1_Click()
{
  drawnShapes = drawMgr.getPrimitives();
  var numShapes = drawnShapes.length;
  var mostRecent = drawnShapes[numShapes-1];
. . .
```

An array of shapes that have been interactively created using the drawing control can be retrieved using the **getPrimitives()** function. In general, the retrieved shapes are read-only. For example, the statement **mostRecent.setOptions({fillColor:'red'})** will have no effect. In order to indirectly manipulate a drawn shape, you can fetch information about the shape, programmatically create a duplicate, then delete the original drawn shape.

The function gets and displays information about the nyh recent shape this way:

```
. . .
  WriteLn("\nThere are " + numShapes + " user-drawn shapes");
  var isPoly = (mostRecent instanceof Microsoft.Maps.Polygon);
  var isPushpin = (mostRecent instanceof Microsoft.Maps.Pushpin);
  if (isPoly == true) {
    WriteLn("\nVertices of most recent drawn shape: ");
    var vertices = mostRecent.getLocations();
    for (var i = 0; i < vertices.length; ++i) {
      WriteLn(LatLonStr(vertices[i]));
    }
    WriteLn("");
  }
  else if (isPushpin == true) {
    WriteLn("\nLocation of most recent pushpin: ");
    var loc = mostRecent.getLocation();
    WriteLn("loc = " + loc);
  }
}
```

In order to determine the type of a drawn shape, you must use the built-in **instanceof** operator rather than the built-in **typeof()** function because **typeof()** will only return “object.”

In summary, you can use the DrawingManager class of the Maps.DrawingTools module to provide users with a way to interactively add pushpins, polylines, and polygons. (There are plans to enhance the functionality of the drawing control to include other shapes).

Resources

For detailed information about the Maps.DrawingTools module, see: https://msdn.microsoft.com/en-us/library/mt750543.aspx.

For detailed information about the DrawingManagerClass, see: https://msdn.microsoft.com/en-us/library/mt750462.aspx.

For detailed information about using the Polygon class, see: https://msdn.microsoft.com/en-us/library/mt712647.aspx.

2.3 Map bounds

Because users can change a map's view (including zoom level and position), geo-applications sometimes need to know map-location properties such as the current zoom level, current map center location, width of the map in degrees, width in miles, and so on.

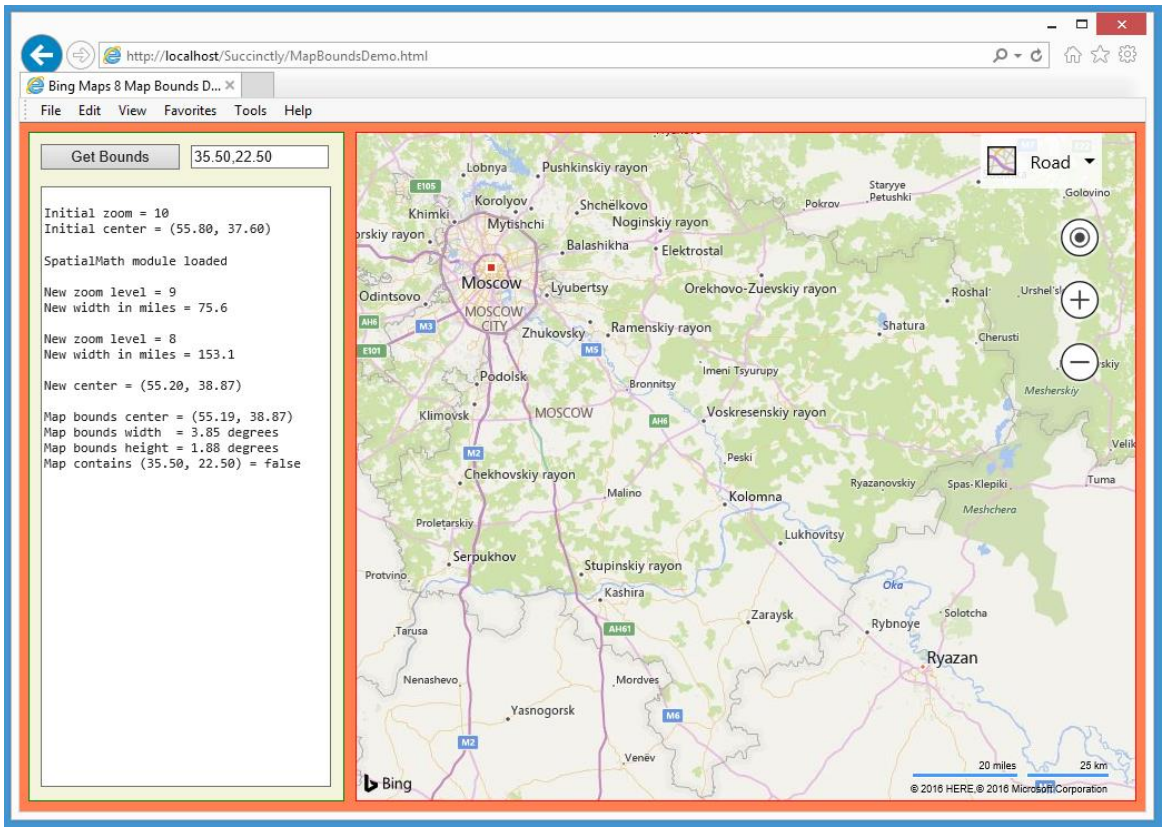

Figure 15: Getting Map Properties Information

The demo web application shown in Figure 15 loads a map with initial zoom level of 10 centered at (55.80, 37.60) near Moscow (Russia, not Idaho!). During map initialization, the application loaded the Maps.SpatialMath module that contains many useful geomath functions.

The user clicked the built-in zoom control labeled with a minus sign to zoom out to level 9. The application is coded to detect changes in the map's view, and it programmatically calculated the new width of the map view (75.6 miles). Then the user zoomed out again, to level 8. The new width of the map is 153.1 miles.

The user then dragged the map so that Moscow was repositioned to the upper-left corner of the map. The application detected a new map center at (55.15, 38.96) and displayed it. The user clicked "Get Bounds" and the application code displayed the current bounds information, then calculated that the current view doesn't contain location (35.50, 22.50).

The demo web application is named MapBoundsDemo.html and is defined in a single file.

Code Listing 5: MapBoundsDemo.html

```
<!DOCTYPE html>
<!-- MapBoundsDemo.html -->

<html>
  <head>
    <title>Bing Maps 8 Map Bounds Demo</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>

    <script type='text/javascript'
      src='http://www.bing.com/api/maps/mapcontrol'>
    </script>

    <script type="text/javascript">

    var map = null;
    var zoomLev;
    var mapCtr = null;

    function GetMap()
    {
      var options = {
        credentials: "Anw _ _ _ 3xt",
        center: new Microsoft.Maps.Location(55.80, 37.60),
        mapTypeId: Microsoft.Maps.MapTypeId.road,
        zoom: 10,
        enableClickableLogo: false,
        showTermsLink: false
      };

      var mapDiv = document.getElementById("mapDiv");
      map = new Microsoft.Maps.Map(mapDiv, options);
      zoomLev = map.getZoom();
      mapCtr = map.getCenter();
      WriteLn("\nInitial zoom = " + zoomLev);
      WriteLn("Initial center = " + LatLonStr(mapCtr));

      Microsoft.Maps.Events.addHandler(map, 'viewchangeend', ViewChanged);

      Microsoft.Maps.loadModule("Microsoft.Maps.SpatialMath",
        function () {
          WriteLn("\nSpatialMath module loaded");
        });
    }

    function ViewChanged()
    {
      var z = map.getZoom();
      var mc = map.getCenter();

      if (z != zoomLev) {
        WriteLn("\nNew zoom level = " + z);
        zoomLev = z;
```

```
    var bounds = map.getBounds();
    var se = bounds.getSoutheast();
    var s = bounds.getSouth();
    var w = bounds.getWest();
    var sw = new Microsoft.Maps.Location(s, w);
    var inMiles = Microsoft.Maps.SpatialMath.DistanceUnits.Miles;
    var widthOfMap =
      Microsoft.Maps.SpatialMath.getDistanceTo(se, sw, inMiles);
    WriteLn("New width in miles = " + Number(widthOfMap).toFixed(1));
  }

  if (mc.latitude != mapCtr.latitude || mc.longitude != mapCtr.longitude) {
    WriteLn("\nNew center = " + LatLonStr(mc));
    mapCtr = mc;
  }

}

function WriteLn(txt)
{
  var existing = msgArea.value;
  msgArea.value = existing + txt + "\n";
}

function LatLonStr(loc)
{
  var s = "(" + Number(loc.latitude).toFixed(2) + ", " +
    Number(loc.longitude).toFixed(2) + ")";
  return s;
}

function Button1_Click()
{
  var bounds = map.getBounds();

  var ctr = bounds.center;
  var wid = bounds.width;
  var ht = bounds.height;

  WriteLn("\nMap bounds center = " + LatLonStr(ctr));
  WriteLn("Map bounds width  = " + Number(wid).toFixed(2) + " degrees");
  WriteLn("Map bounds height = " + Number(ht).toFixed(2) + " degrees");

  var strLoc = textbox1.value;
  var latLon = strLoc.split(',');
  var lat = latLon[0];
  var lon = latLon[1];
  var loc = new Microsoft.Maps.Location(lat, lon);
  var b = bounds.contains(loc);

  WriteLn("Map contains " + LatLonStr(loc) + " = " + b);
}
```

```
    </script>
  </head>

  <body onload="GetMap();" style="background-color:coral">
    <div id='controlPanel' style="float:left; width:262px; height:580px;
      border:1px solid green; padding:10px; background-color: beige">

      <input id="button1" type='button' style="width:125px;"
        value='Get Bounds' onclick="Button1_Click();"></input>
      <div style="width:2px; display:inline-block"></div>
      <input id="textbox1" type='text' size='15'
        value='35.50,22.50'></input><br/>
      <span style="display:block; height:15px"></span>

      <textarea id='msgArea' rows="38" cols="36" style="font-family:Consolas;
        font-size:12px"></textarea>
    </div>

    <div style="float:left; width:10px; height:600px"></div>
    <div id='mapDiv' style="float:left; width:700px; height:600px;
      border:1px solid red;"></div>
    <br style="clear: left;" />

  </body>
</html>
```

The demo application loads the Bing Maps library using the old synchronous technique by placing an ordinary referencing script tag in the HTML head section of the page. I don't recommend this approach, but it's worth seeing what it looks like.

The application sets up three global script-scope objects:

```
var map = null;
var zoomLev;
var mapCtr = null;
```

Typically, global objects in a map application are used by more than one function (such as the **map** object) or those that record and maintain state—for example, the **zoomLev** and **mapCtr** objects here that store the current zoom level and map center location.

Map initialization is performed by program-defined function **GetMap()**, which is referenced in the HTML body tag:

```
<body onload="GetMap();" style="background-color:coral">
```

Function **GetMap()** sets up more or less standard options:

```
function GetMap()
{
  var options = {
    credentials: "Anw _ _ _ 3xt",
    center: new Microsoft.Maps.Location(55.80, 37.60),
    mapTypeId: Microsoft.Maps.MapTypeId.road,
    zoom: 10,
    enableClickableLogo: false,
    showTermsLink: false
  };
. . .
```

After the map is created and placed on the map, the zoom and center states are saved and displayed:

```
var mapDiv = document.getElementById("mapDiv");
map = new Microsoft.Maps.Map(mapDiv, options);
zoomLev = map.getZoom();
mapCtr = map.getCenter();
WriteLn("\nInitial zoom = " + zoomLev);
WriteLn("Initial center = " + LatLonStr(mapCtr));
```

Map initialization concludes by adding an event handler and loading a module:

```
. . .
  Microsoft.Maps.Events.addHandler(map, 'viewchangeend', ViewChanged);
  Microsoft.Maps.loadModule("Microsoft.Maps.SpatialMath",
    function () {
      WriteLn("\nSpatialMath module loaded");
    });
}
```

The **`map`** object supports 13 events, including “viewchange,” “viewchangestart,” and “viewchangeend.” A view change includes a change in zoom level, map type (aerial, road, etc.), and map position. Here the instructions to execute on a change are placed in a program-defined function **`ViewChanged()`**.

The Maps.SpatialMath module is loaded so that the static functions in the module can be ready to be called rather than being loaded the module when needed. The anonymous callback function will display a message after the module is loaded. The callback function is required so that you can’t pass **`null`** as an argument, but you can use a **`return true`** statement if you wish.

Most of the application code logic is in the **`ViewChanged()`** function that begins:

```
function ViewChanged()
{
  var z = map.getZoom();
  var mc = map.getCenter();
. . .
```

When the **ViewChanged()** function is entered, you don't know what type of event acted as the trigger, so the function fetches the map zoom level and center location to see if they have changed.

The function first checks to see if there was a change in zoom level:

```
if (z != zoomLev) {
  WriteLn("\nNew zoom level = " + z);
  zoomLev = z;
  var bounds = map.getBounds();
  var se = bounds.getSoutheast();
  var s = bounds.getSouth();
  var w = bounds.getWest();
  var sw = new Microsoft.Maps.Location(s, w);
  var inMiles = Microsoft.Maps.SpatialMath.DistanceUnits.Miles;
  var widthOfMap =
    Microsoft.Maps.SpatialMath.getDistanceTo(se, sw, inMiles);
  WriteLn("New width in miles = " + Number(widthOfMap).toFixed(1));
}
```

You might expect the return value from the **getBounds()** function to store the four corners of the map, but instead the return is a **LocationRec** object that has a center **Location** object with width and height properties that are given in degrees. Luckily, a **LocationRect** object has 13 useful methods. Some examples are shown in Table 2.

Table 2: Representative LocationRect Methods

Name	Description
contains()	determines if a Location is in the LocationRect area
getEast()	returns the right-side longitude (left-right) value
getNorth()	returns the top latitude (up-down) value
getNorthwest()	returns the Location of the upper-left corner
intersects()	determines if one LocationRect overlaps another

Interestingly, there are **getNorthwest()** and **getSoutheast()** functions, but no **getNortheast()** and **getSouthwest()** functions. The **ViewChanged()** function calculates and displays the width of the current view by using the built-in **getSoutheast()** function and by calculating a Southwest location using the **getSouth()** and **getWest()** functions. The **getDistanceTo()** function accepts two **Location** objects. The return value can be computed in different units using a DistanceUnits enumeration of **Feet**, **Kilometers**, **Meters**, **Miles**, **NauticalMiles**, or **Yards**.

Function **ViewChanged()** concludes by checking to see if the map view has been moved:

```
. . .
  if (mc.latitude != mapCtr.latitude || mc.longitude != mapCtr.longitude) {
    WriteLn("\nNew center = " + LatLonStr(mc));
    mapCtr = mc;
  }
}
```

Here, the latitude and longitude properties of the **Location** object are used; if either has changed, the new map center is not the same as the old map center. Alternatively, we can use a slightly less-flexible technique—the static **Location.areEqual()** function. Function **Button1_Click()** uses a different way to get the width of the current map view:

```
function Button1_Click()
{
  var bounds = map.getBounds();
  var ctr = bounds.center;
  var wid = bounds.width;
  var ht = bounds.height;
  WriteLn("\nMap bounds center = " + LatLonStr(ctr));
  WriteLn("Map bounds width  = " + Number(wid).toFixed(2) + " degrees");
  WriteLn("Map bounds height = " + Number(ht).toFixed(2) + " degrees");
. . .
```

Here the width and height properties are used directly to give dimensions measured in degrees. The function concludes by determining if a location is in the current view area:

```
. . .
  var strLoc = textbox1.value;
  var latLon = strLoc.split(',');
  var lat = latLon[0];
  var lon = latLon[1];
  var loc = new Microsoft.Maps.Location(lat, lon);
  var b = bounds.contains(loc);
  WriteLn("Map contains " + LatLonStr(loc) + " = " + b);
}
```

Note that the test location is constructed by breaking apart the text box input because the Location class doesn't have a method to parse from a raw string that combines latitude and longitude.

In summary, the Maps.SpatialMath module has dozens of useful geomath functions that eliminate the need for you to write custom code.

Resources

For detailed information about the Maps.SpatialMath module, see: https://msdn.microsoft.com/en-us/library/mt712834.aspx.

Chapter 3 Working with Data

This chapter presents some of the most common techniques that use data to create map features. Common data sources used with geo-applications include ordinary files stored locally, SQL data stored on a corporate network, and data stored remotely but accessible through a web service. Common data formats include plain text, Well Known Text (WKT), JSON, and GeoJSON.

In section 3.1, you'll learn how to read data in WKT format from the Bing Spatial Data Service and how to filter and parse that data. You'll also learn about color palettes and how to create choropleth maps. You'll also learn about event-handling techniques and how to use object metadata.

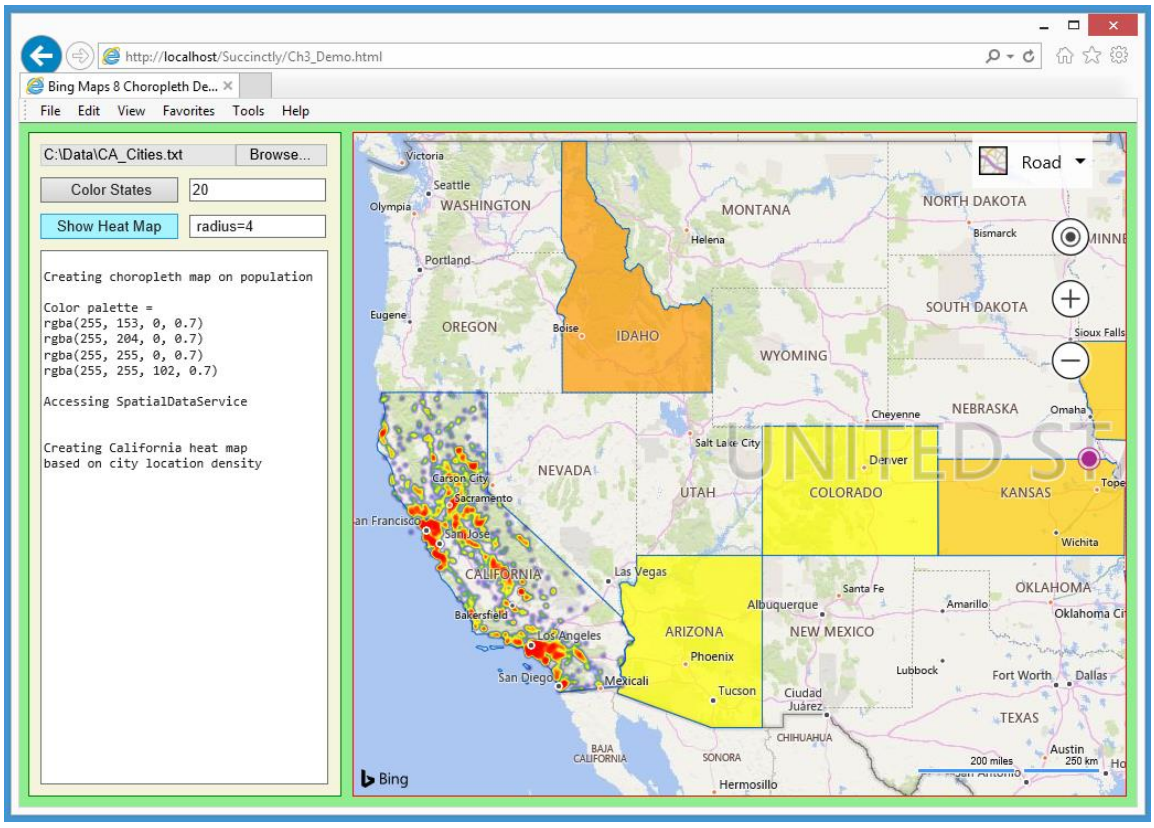

Figure 16: Working with Data Demo

In section 3.2, you'll learn how to read data in JSON from a web service and how to parse that data to create objects that can be added onto a map. You'll also learn how to integrate JavaScript libraries, such as jQuery and AngularJS, into a Bing Maps geo-application.

In section 3.3, you'll learn about color gradients and how to handle large data sets by creating heat maps. You'll also learn how to access government geodata sources such as those provided by the U.S. Census Bureau.

3.1 Choropleth maps

A choropleth (pronounced koro-pleth) map uses different shades of a particular color to indicate the relative proportion of some statistic—for example, population or per-capita income. Creating choropleth maps using the Bing Maps V8 library is relatively simple. You define a color palette, get relevant shape and statistics data, apply a palette color, and add the result to a layer.

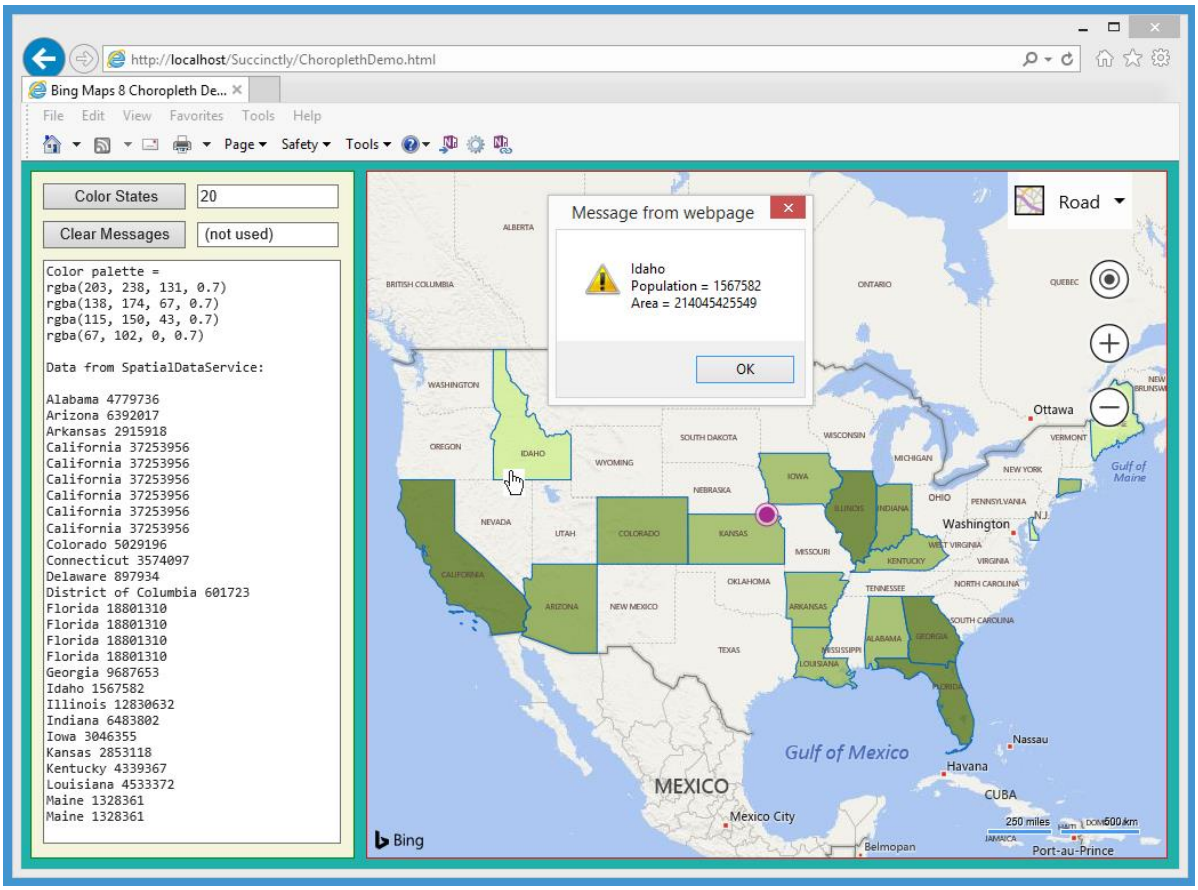

Figure 17: Choropleth Map Demo

The demo web application shown in Figure 17 displays a choropleth map of the United States by population. Darker shades indicate higher populations. The demo initially loads a map centered at (40.00, -96.00) and places a default-style purple pushpin at center. During initialization, a **Layer** object is created to hold the shapes of U.S. states.

When a user clicks the first button control, live shape and population data for the specified number of states is fetched using a REST service. A custom-color palette that consists of four shades of green is hard-coded. For each state, a color is determined based on the state's population, and the state's polygon data is added to the display **Layer** object.

When the polygon for each state is drawn, a click-event handler is added that will display an alert dialog that has the target state's name, population, and land area.

The demo web application is named ChoroplethDemo.html and is defined in a single file.

Code Listing 6: ChoroplethDemo.html

```
<!DOCTYPE html>
<!-- ChoroplethDemo.html -->

<html>
  <head>
    <title>Bing Maps 8 Choropleth Demo</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>

    <script type='text/javascript'>

    var map = null;
    var ppLayer = null;
    var statesLayer = null;

    function GetMap()
    {
      var options = {
        credentials: "Anw _ _ _ 3xt",
        center: new Microsoft.Maps.Location(40.00, -96.00),
        mapTypeId: Microsoft.Maps.MapTypeId.road,
        zoom: 4,
        enableClickableLogo: false,
        showCopyright: false
      };

      var mapDiv = document.getElementById("mapDiv");
      map = new Microsoft.Maps.Map(mapDiv, options);

      ppLayer = new Microsoft.Maps.Layer();
      var cpp= new Microsoft.Maps.Pushpin(map.getCenter(), null);
      ppLayer.add(cpp);
      map.layers.insert(ppLayer);
      statesLayer = new Microsoft.Maps.Layer();
    }

    function WriteLn(txt)
    {
      var existing = msgArea.value;
      msgArea.value = existing + txt + "\n";
    }

    function Button1_Click()
    {
      var paletteGreen = [ 'rgba(203, 238, 131, 0.7)',
        'rgba(138, 174, 67, 0.7)', 'rgba(115, 150, 43, 0.7)',
        'rgba(67, 102, 0, 0.7)' ];

      WriteLn("Color palette = ");
      for (var i = 0; i < paletteGreen.length; ++i) {
        WriteLn(paletteGreen[i]);
```

```
    }
    var maxCount = textbox1.value;

    Microsoft.Maps.loadModule('Microsoft.Maps.SpatialDataService',
      function() {
      var world = Microsoft.Maps.LocationRect.fromEdges(90, -180, -90, 180);

      var queryOptions = {
        queryUrl: 'https://spatial.virtualearth.net/REST/v1/data/' +
          '755aa60032b24cb1bfb54e8a6d59c229/USCensus2010_States/States',
        spatialFilter: {
          spatialFilterType: 'intersects',
          intersects: world
        },
        top: maxCount
      };

      Microsoft.Maps.SpatialDataService.QueryAPIManager.search(queryOptions,
        map, function (data) {

        WriteLn("\nData from SpatialDataService: \n");
        for (var i = 0; i < data.length; ++i) {
          // WriteLn(Object.keys(data[i].metadata));
          var name = data[i].metadata.Name;
          var pop = data[i].metadata.Population;

          if (name == "Alaska" || name == "Hawaii") {
            continue;
          }
          WriteLn(name + " " + pop);

          data[i].setOptions({
            fillColor: GetStateColor(pop, paletteGreen)
          });

          Microsoft.Maps.Events.addHandler(data[i], 'click', function (e) {
            var n = e.target.metadata.Name;
            var p = e.target.metadata.Population;
            var a = e.target.metadata.Area_Land;
            alert(n + '\nPopulation = ' + p + '\nArea = ' + a);
          });
        }

        statesLayer.add(data);
        map.layers.insert(statesLayer);
      });
    });
  } // Button1_Click()

  function GetStateColor(pop, palette)
  {
    if (pop < 2000000) {
      return palette[0];
    }
```

```
      else if (pop < 5000000) {
        return palette[1];
      }
      else if (pop < 8000000) {
        return palette[2];
      }
      else {
        return palette[3];
      }
    }

    function Button2_Click()
    {
      msgArea.value = "";
    }

    </script>
  </head>

  <body style="background-color:lightseagreen">
    <div id='controlPanel' style="float:left; width:262px; height:580px;
      border:1px solid green; padding:10px; background-color: beige">
      <input id="button1" type='button' style="width:125px;"
        value=' Color States' onclick="Button1_Click();"></input>
      <div style="width:2px; display:inline-block"></div>
      <input id="textbox1" type='text' size='15' value='20'>
        </input><br/>
      <span style="display:block; height:10px"></span>

      <input id="button2" type='button' style="width:125px;"
        value=' Clear Messages ' onclick="Button2_Click();"></input>
      <div style="width:2px; display:inline-block"></div>
      <input id="textbox2" type='text' size='15' value=' (not used) '>
        </input><br/>
      <span style="display:block; height:10px"></span>

      <textarea id='msgArea' rows="36" cols="36" style="font-family:Consolas;
        font-size:12px"></textarea>
    </div>

    <div style="float:left; width:10px; height:600px"></div>
    <div id='mapDiv' style="float:left; width:700px; height:600px;
      border:1px solid red;"></div>
    <br style="clear: left;" />

    <script type='text/javascript'
      src='http://www.bing.com/api/maps/mapcontrol?callback=GetMap'
      async defer>
    </script>

  </body>
</html>
```

The demo application sets up three global script-scope objects:

```
var map = null;
var ppLayer = null;
var statesLayer = null;
```

Object **ppLayer** is a map layer for holding ordinary pushpins, in this case a single default-style pushpin to mark the map center. The **statesLayer** object is the main display layer for the choropleth coloring. The palette that defines the colors used by the choropleth is defined locally to the first button control's click handler, but it could have been defined with the global objects.

The demo application loads the **map** object asynchronously by calling the program-defined **GetMap()** function in the usual way. The **stateLayer** object is instantiated in the **GetMap()** function.

All of the work is done inside the first button control's click handler function. The function definition begins with:

```
function Button1_Click()
{
  var paletteGreen = ['rgba(203, 238, 131, 0.7)', 'rgba(138, 174, 67, 0.7)',
    'rgba(115, 150, 43, 0.7)', 'rgba(67, 102, 0, 0.7)'];

  WriteLn("Color palette = ");
  for (var i = 0; i < paletteGreen.length; ++i) {
    WriteLn(paletteGreen[i]);
  }
  var maxCount = textbox1.value;
. . .
```

There's actually quite a bit of research on how many and which colors to use for a choropleth map, but I arbitrarily chose four shades of green and created a palette using the built-in **rgba()** function. You can also use color names. For example:

```
var paletteGreen = [ 'greenyellow', 'lightgreen',
  'mediumseagreen', 'darkolivegreen' ];
```

The maximum number of states to process (20 in the demo) is pulled from the textbox1 control. It's not necessary to cast the value to an integer in this situation.

There are several ways to get choropleth data. The demo application uses the Bing Maps V8 SpatialDataService module to get live state data from a Bing Maps RESTful web service. All of the work is done inside the module's callback function:

```
Microsoft.Maps.loadModule('Microsoft.Maps.SpatialDataService', function () {
  var world = Microsoft.Maps.LocationRect.fromEdges(90, -180, -90, 180);
  var queryOptions = {
    queryUrl: 'https://spatial.virtualearth.net/REST/v1/data/' +
      '755aa60032b24cb1bfb54e8a6d59c229/USCensus2010_States/States',
    spatialFilter: {
      spatialFilterType: 'intersects',
      intersects: world
    },
    top: maxCount
  };
```

There are several different types of queries supported by the web service, but most require some sort of spatial filter to narrow the result list down. For example, if you query the service for points of interest, you'd probably want to limit your results to a city or a region. The demo sets its filter to the entire world because there will only be a maximum of 52 results returned (the 50 states plus Puerto Rico and the District of Columbia).

Next, the **`search()`** function is invoked:

```
Microsoft.Maps.SpatialDataService.QueryAPIManager.search(queryOptions,
  map, function (data) {
  WriteLn("\nData from SpatialDataService: \n");
  for (var i = 0; i < data.length; ++i) {
    var name = data[i].metadata.Name;
    var pop = data[i].metadata.Population;
    if (name == "Alaska" || name == "Hawaii") {
      continue;
    }
    WriteLn(name + " " + pop);
    data[i].setOptions({
      fillColor: GetStateColor(pop, paletteGreen)
    });
    Microsoft.Maps.Events.addHandler(data[i], 'click', function (e) {
      var n = e.target.metadata.Name;
      var p = e.target.metadata.Population;
      var a = e.target.metadata.Area_Land;
      alert(n + '\nPopulation = ' + p + '\nArea = ' + a);
    });
  }
```

The callback function returns results into an object named **`data`**. Each item in **`data`** has a metadata property that in turn has a Name, Population, Land_Area, and other information. I determined the metadata names during development by placing this statement in the for-loop:

```
WriteLn(Object.keys(data[i].metadata));
```

The demo performs a secondary programmatic filter to remove Alaska and Hawaii. The color for each state is set by calling a program-defined **`GetStateColor()`** function. Then each data item has an event handler added that will display metadata information in a humble JavaScript alert box. Using a custom **`Infobox`** object is an alternative.

The function that determines color value from population is:

```
function GetStateColor(pop, palette)
{
  if (pop < 2000000) {
    return palette[0];
  }
  else if (pop < 5000000) {
    return palette[1];
  }
  else if (pop < 8000000) {
    return palette[2];
  }
  else {
    return palette[3];
  }
}
```

The function is simple but effective because only four colors are being dealt with. When you have many different choropleth colors, one alternative approach is to define an array of threshold values. For example:

```
var popLimits = [ 0, 2000000, 5000000, 8000000 ];
```

You could write code that accepts a population value, scans through the threshold values, and returns an index into the palette array.

In summary, it's relatively easy to create a choropleth map once you have shape data and associated statistics data. The most difficult part is getting shape data. You can use the virtualearth.net REST service, one of many external data sources available, and you can parse the results of that service by using the **`search()`** function in the Bing Maps V8 library SpatialDataService module. Alternative approaches for creating a choropleth map include getting data as plain text, WKT, GeoJSON, or standard JSON format from a government or commercial web service and parsing the data programmatically.

Resources

For detailed information about the Bing Maps V8 Spatial Data Service module, see: https://msdn.microsoft.com/en-us/library/mt712849.aspx.

For detailed information about the Query API component, see: https://msdn.microsoft.com/en-us/library/gg585126.aspx.

You can see collections of shades for various colors at: http://www.w3schools.com/colors/colors_groups.asp.

3.2 Data from a web service

One of the most common ways to obtain data for a map application is to query a web service. In most cases a web service returns data in JSON or XML format. An effective technique for getting data for a Bing Maps application from a web service is to use the jQuery library. Because the Bing Maps library is simply ordinary JavaScript, using the jQuery library is a natural approach.

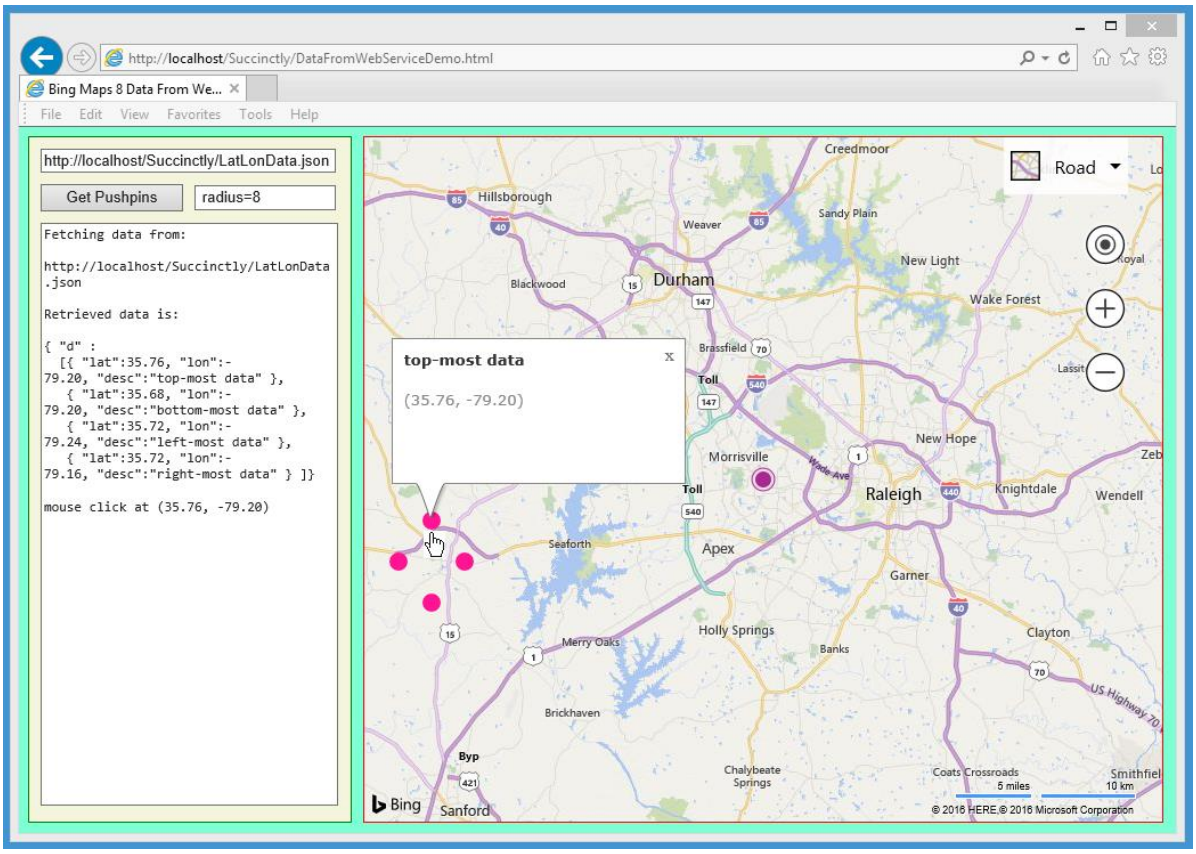

Figure 18: Data from a Web Service Demo

The demo web application shown in Figure 18 gives an example of fetching JSON data from a simulated web service. The demo initially loads a map centered near Morrisville, N.C., and places a default-style large purple pushpin at the map center.

The text box control in the upper left points to a data file named LatLonData.json on the local web server, but the technique presented in this section can use any URL that points to a web service that returns JSON data. When the user clicks “Get Pushpins,” a request is sent to the simulated web service. The returned data is echoed and used to place four pushpins on the map. Each pushpin has a mouse click event handler that displays an **`Infobox`** object containing data about the associated pushpin.

The demo application is named DataFromWebServiceDemo.html and is defined in a single file.

Code Listing 7: DataFromWebServiceDemo.html

```
<!DOCTYPE html>
<!-- DataFromWebServiceDemo.html -->

<html>
  <head>
    <title>Bing Maps 8 Data From Web Service</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>

    <script type="text/javascript"
      src="http://code.jquery.com/jquery-latest.min.js"> </script>
    <script type='text/javascript'>

    var map = null;
    var pushpins = [];
    var infobox = null;
    var ppLayer = null;

    function GetMap()
    {
      var options = {
        credentials: "Anw _ _ _ 3xt",
        center: new Microsoft.Maps.Location(35.80, -78.80),
        mapTypeId: Microsoft.Maps.MapTypeId.road,
        zoom: 10,
        enableClickableLogo: false,
        showTermsLink: false
      };

      var mapDiv = document.getElementById("mapDiv");
      map = new Microsoft.Maps.Map(mapDiv, options);

      infobox = new Microsoft.Maps.Infobox(new Microsoft.Maps.Location(0, 0),
        { visible: false, offset: new Microsoft.Maps.Point(0,0) });
      infobox.setMap(map);

      ppLayer = new Microsoft.Maps.Layer();
      var cpp= new Microsoft.Maps.Pushpin(map.getCenter(), null);
      ppLayer.add(cpp);
      map.layers.insert(ppLayer);
    }

    function WriteLn(txt)
    {
      var existing = msgArea.value;
      msgArea.value = existing + txt + "\n";
    }

    function LatLonStr(loc)
    {
      var s = "(" + Number(loc.latitude).toFixed(2) + ", " +
        Number(loc.longitude).toFixed(2) + ")";
      return s;
```

```
}

$(document).ready(function() {
  $("#button1").click(function() {
    var urlVal = $("#textbox1").val();
    WriteLn("Fetching data from: \n");
    WriteLn(urlVal + "\n");

    var radius = parseInt((textbox2.value).split('=')[1]);
    var pinkDot = CreateSvgDot(radius, "DeepPink");

    $.ajax({
      url: urlVal,
      dataType: "text",
      success: function (data) {
        WriteLn("Retrieved data is: \n");
        WriteLn(data);
        var parsed = JSON.parse(data);
        var len = parsed.d.length;
        for (var i = 0; i < len; ++i) {
          var loc = new Microsoft.Maps.Location(parsed.d[i].lat,
            parsed.d[i].lon);

          var ppOptions = { icon: pinkDot,
            anchor: new Microsoft.Maps.Point(radius,radius) };

          var pp = new Microsoft.Maps.Pushpin(loc, ppOptions);
          pp.metadata = parsed.d[i].desc;
          pushpins[i] = pp;
          Microsoft.Maps.Events.addHandler(pp, 'click', ShowInfobox);
        }
        ppLayer.add(pushpins);
        map.layers.insert(ppLayer);
      }
    });
  });
});

function ShowInfobox(e)
{
  var loc = e.target.getLocation();
  WriteLn('\nmouse click at ' + LatLonStr(loc));
  infobox.setLocation(loc);
  infobox.setOptions({ visible: true,
    title: e.target.metadata,
    description: LatLonStr(loc)
  });
}

function CreateSvgDot(radius, clr)
{
  var s = '<svg xmlns="http://www.w3.org/2000/svg"';
```

```
      s += ' width="' + radius * 2 +'"';
      s += ' height="' + radius * 2 + '"';
      s += '>';

      s += '<circle cx="' + radius + '"';
      s += ' cy="' + radius + '"';
      s += ' r="' + radius + '"';
      s += ' fill="' + clr + '"';
      s += ' /></svg>';
      return s;
    }

    </script>
  </head>

  <body style="background-color:aquamarine">
    <div id='controlPanel' style="float:left; width:262px; height:580px;
      border:1px solid green; padding:10px; background-color: beige">

      <input id="textbox1" type="text" size="38"
        value="http://localhost/Succinctly/LatLonData.json"></input>
      <span style="display:block; height:10px"></span>

      <input id="button1" type='button' style="width:125px;"
        value='Get Pushpins' onclick="Button1_Click();"></input>
      <div style="width:2px; display:inline-block"></div>
      <input id="textbox2" type='text' size='15'
        value='radius=8'></input><br/>
      <span style="display:block; height:10px"></span>

      <textarea id='msgArea' rows="36" cols="36" style="font-family:Consolas;
        font-size:12px"></textarea>
    </div>

    <div style="float:left; width:10px; height:600px"></div>
    <div id='mapDiv' style="float:left; width:700px; height:600px;
      border:1px solid red;">
    </div>
    <br style="clear: left;" />

    <script type='text/javascript'
      src='http://www.bing.com/api/maps/mapcontrol?callback=GetMap'
      async defer>
    </script>

  </body>
</html>
```

A reference to the jQuery library is placed at the top of the HTML head section:

```
<script type="text/javascript"
  src="http://code.jquery.com/jquery-latest.min.js"></script>
<script type='text/javascript'>
```

The idea is to use the jQuery library to fetch JSON data from a web service. An alternative is to use the built-in JavaScript **`XMLHttpRequest()`** function. In general, I try to avoid external dependencies, but the jQuery library's **`ajax()`** function is very convenient.

Most of the work is performed by the event handler for the HTML button1 control. The structure of the handler is:

```
$(document).ready(function() {
  $("#button1").click(function() {
    var urlVal = $("#textbox1").val();
    WriteLn("Fetching data from: \n");
    WriteLn(urlVal + "\n");

    var radius = parseInt((textbox2.value).split('=')[1]);
    var pinkDot = CreateSvgDot(radius, "DeepPink");

    $.ajax({
      url: urlVal,
      dataType: "text",
      success: function (data) {
        WriteLn("Retrieved data is: \n");
        WriteLn(data);
        var parsed = JSON.parse(data);
        var len = parsed.d.length;
        for (var i = 0; i < len; ++i) {
          // make pushpin, add to global pushpins[] array
        }

        ppLayer.add(pushpins);
        map.layers.insert(ppLayer);
      }
    });
  });
});
```

The code begins by grabbing the URL from the textbox1 control using jQuery syntax with the **`val()`** function, then echoing the URL to the HTML message area. In a production scenario, at this point you'd probably want to do some error checking of the URL value.

Next, a pink dot, to be used as the pushpin icon, is created dynamically by a call to a program-defined **`CreateSvgDot()`** function that I will explain shortly. First, the desired radius is fetched from the textbox2 control. I used standard non-jQuery syntax in order to show the difference.

The **ajax()** function can accept 34 settings, but the default values for most of the settings are used. The key setting in the demo is the **success** setting, which is a callback function that will execute after the data has been returned asynchronously.

The JSON data returned from the simulated web service is:

```
{ "d" :
  [
   { "lat":35.76, "lon":-79.20, "desc":"top-most data" },
   { "lat":35.68, "lon":-79.20, "desc":"bottom-most data" },
   { "lat":35.72, "lon":-79.24, "desc":"left-most data" },
   { "lat":35.72, "lon":-79.16, "desc":"right-most data" }
  ]
}
```

The top-level field is tersely named “d,” and the contents consist of an array of four items in which each item has a latitude, a longitude, and a description. The **JSON.parse()** function is used to break the data into key-value pairs, and the result is stored into a local object named **parsed**.

The code that creates the pushpins is:

```
for (var i = 0; i < len; ++i) {
  var loc = new Microsoft.Maps.Location(parsed.d[i].lat,
    parsed.d[i].lon);

  var ppOptions = { icon: pinkDot,
    anchor: new Microsoft.Maps.Point(radius,radius) };

  var pp = new Microsoft.Maps.Pushpin(loc, ppOptions);
  pp.metadata = parsed.d[i].desc;
  pushpins[i] = pp;
  Microsoft.Maps.Events.addHandler(pp, 'click', ShowInfobox);
}
```

The description information is added to each **pushpin** object as a property named metadata. The code here is using the rather strange JavaScript ability to add arbitrary properties to objects, which means the name metadata is not required and the code could have been written as:

```
pp.foo = parsed.d[i].desc;
```

Each pushpin has its click event modified so that control will be transferred to a program-defined function **ShowInfobox()**. Alternatively, you can directly define the actions you’ll take by using an anonymous function.

The demo creates custom pushpin icons using function **CreateSvgDot()**, which is defined this way:

```
function CreateSvgDot(radius, clr)
{
  var s = '<svg xmlns="http://www.w3.org/2000/svg"';
  s += ' width="' + radius * 2 +'"';
  s += ' height="' + radius * 2 + '"';


  s += '>';
  s += '<circle cx="' + radius + '"';
  s += ' cy="' + radius + '"';
  s += ' r="' + radius + '"';
  s += ' fill="' + clr + '"';
  s += ' /></svg>';
  return s;
}
```

An example of an SVG image is:

```
<svg xmlns="http://www.w3.org/2000/svg" width="16" height="16">
  <circle cx="8" cy="8" r="8" fill="red" />
</svg>
```

So, one technique for creating an SVG image on the fly is to write a function that returns a big string. An alternative is to use the built-in JavaScript **createElementNS()** and **setAttributeNS()** functions.

The **ShowInfobox()** callback function is executed when a user clicks a pushpin:

```
function ShowInfobox(e)
{
  var loc = e.target.getLocation();
  WriteLn('\nmouse click at ' + LatLonStr(loc));
  infobox.setLocation(loc);
  infobox.setOptions({
    visible: true,  title: e.target.metadata, description: LatLonStr(loc)
  });
}
```

Notice that the location of the Infobox is determined by programmatically fetching the location of a pushpin. Alternatively, when pushpin locations do not change, we can store pushpin locations in an array when they are created and return an index into the array when the mouseover or click event is fired.

In summary, when you need external data for a geolocation map application and that data is exposed through a web service that returns JSON or XML data, using the **ajax()** function in the jQuery library is a good approach. You can create a dynamic, custom icon image for a **pushpin** object by writing a function that returns an SVG string.

Resources

For detailed information about SVG images, see:
https://developer.mozilla.org/en-US/docs/Web/SVG/Element.

For detailed information about the jQuery **`ajax()`** function, see:
http://api.jquery.com/jquery.ajax/.

3.3 Heat maps

When dealing with large numbers of latitude-longitude data points, an alternative to displaying the data points as separate pushpins (or as clustered pushpins) is to display the data as a heat map. There are several kinds of heat maps, and one common type displays combined data points using a color gradient where different colors represent different data densities.

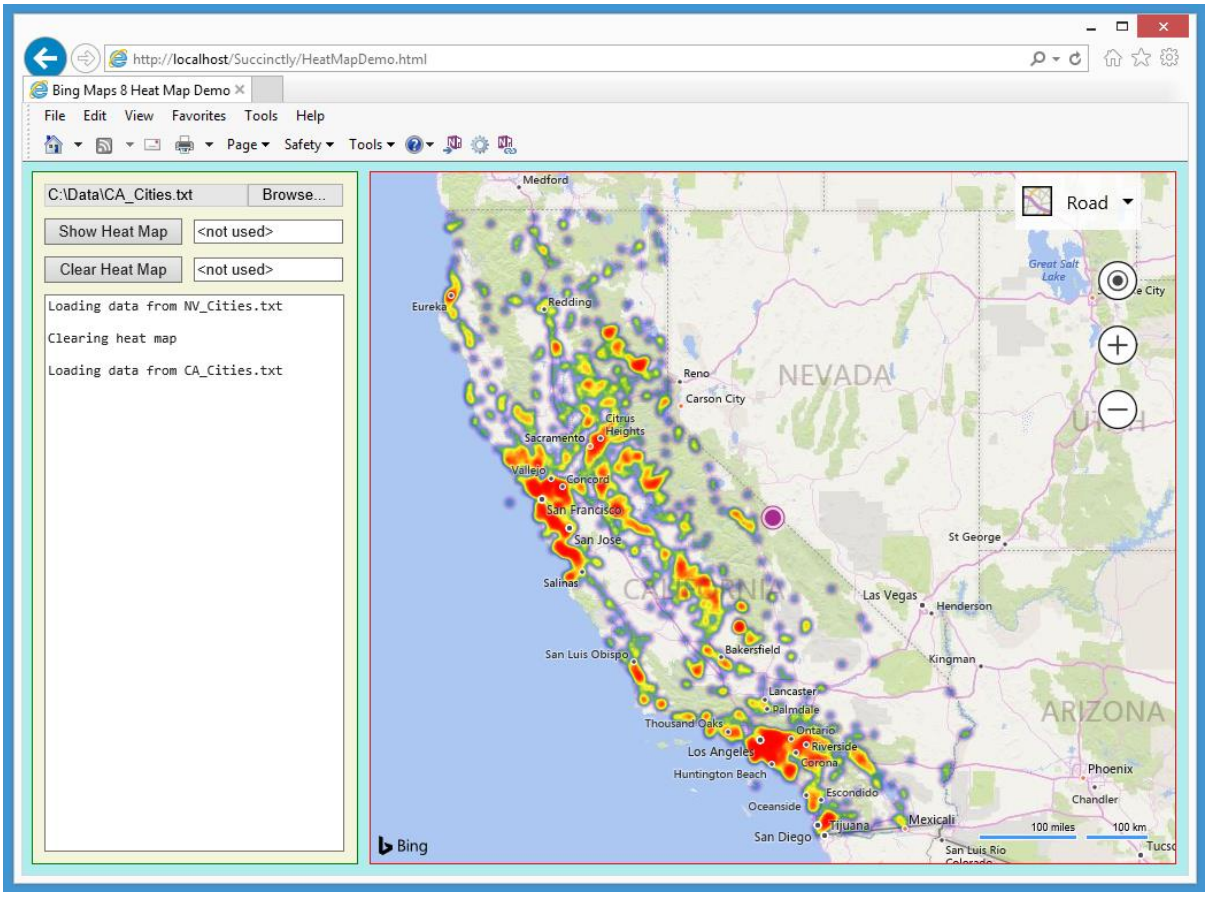

Figure 19: Heat Map Demo

The demo web application shown in Figure 19 initially loads a map centered at (37.50, -118.00) and places a default, large purple pushpin at center. First, the user clicked “HTML5 File Browse” and pointed to a local file named NV_Cities.txt that contained city data. Next, the user clicked the first button control, which loaded and displayed a heat map for city density in the state of Nevada. The user then cleared that heat map by using the second button control.

Note that the user next clicked “Browse” and pointed to a tab-separated text file named CA_Cities.txt. That data file contains a list of 1,522 cities in California and the corresponding latitude-longitude information for each. Next, the user clicked “Show Heat Map,” which read the text file, parsed out the lat-lon data, and stored that data into an array. Then the lat-lon data was displayed as a heat map, generating a city-density visualization.

The demo web application is named HeatMapDemo.html and is defined in a single file.

Code Listing 8: HeatMapDemo.html

```
<!DOCTYPE html>
<!-- HeatMapDemo.html -->

<html>
<head>
  <title>Bing Maps 8 Heat Map Demo</title>
  <meta http-equiv='Content-Type' content='text/html; charset=utf-8'/>

  <script type='text/javascript'>

    var map = null;
    var ppLayer = null;  // pushpin layer
    var hmLayer = null;  // heat map layer
    var reader = null;   // FileReader object
    var locs = [];       // lat-lon locations
    var cGrad = { '0.0': 'black', '0.2': 'purple', '0.4': 'blue',
     '0.6': 'green', '0.8': 'yellow', '0.9': 'orange',
     '1.0': 'rgb(255,0,0)' };
    var hmOptions = { intensity: 0.65, radius: 7, colorGradient: cGrad };

    function GetMap()
    {
      var options = {
        credentials: "Anw _ _ _ 3xt",
        center: new Microsoft.Maps.Location(37.50, -118.00),
          mapTypeId: Microsoft.Maps.MapTypeId.road,
        zoom: 6, enableClickableLogo: false, showCopyright: false
      };

      var mapDiv = document.getElementById("mapDiv");  // where to place map
      map = new Microsoft.Maps.Map(mapDiv, options);   // display the map

      ppLayer = new Microsoft.Maps.Layer();
      var cpp = new Microsoft.Maps.Pushpin(map.getCenter(), null);  // center pp
      ppLayer.add(cpp);
      map.layers.insert(ppLayer);
    }

    function Button1_Click()
    {
      if (reader == null || locs.length == 0) {
        LoadLocs();
      }
    }

    function LoadLocs()
    {
      var f = file1.files[0];  // get filename
      WriteLn('Loading data from ' + f.name + "\n");
      reader = new FileReader();
      reader.onload = function(e) {  // after file is read . .
        var lines = reader.result.split('\n');  // array of lines
```

```
      for (var i = 0; i < lines.length; ++i) {  // each line
        var line = lines[i];
        var tokens = line.split('\t');  // split on tabs
        try {
          var loc = new Microsoft.Maps.Location(tokens[12], tokens[13]);
          locs[i] = loc;
        }
        catch (err)  {
          WriteLn(err + " " + line);
        }
      }

      Microsoft.Maps.loadModule('Microsoft.Maps.HeatMap', function () {
        hmLayer = new Microsoft.Maps.HeatMapLayer(locs, hmOptions);
        map.layers.insert(hmLayer);
      });

    }
    reader.readAsText(f);  // read the file asynchronously
  }

  function Button2_Click()
  {
    WriteLn('Clearing heat map' + "\n");
    hmLayer.clear();
    reader = null;
    locs = [];
  }

  function WriteLn(txt)
  {
    var existing = msgArea.value;
    msgArea.value = existing + txt + "\n";
  }

  </script>

  </head>
  <body style="background-color:paleturquoise">

    <div id='controlPanel' style="float:left; width:262px; height:580px;
      border:1px solid green; padding:10px; background-color: beige">
      <input type="file" id="file1" size="24">
      <span style="display:block; height:10px"></span>

      <input id="button1" type='button' value='Show Heat Map'
        style="width:120px;" onclick="Button1_Click();"></input>
      <div style="width:2px; display:inline-block"></div>
      <input id="textbox1" type='text' size='16' value=' (not used) '>
        </input><br/>
      <span style="display:block; height:10px"></span>

      <input id="button2" type='button' value='Clear Heat Map'
        style="width:120px;" onclick="Button2_Click();"></input>
```

```
        <div style="width:2px; display:inline-block"></div>
        <input id="textbox2" type='text' size='16' value=' (not used) '>
          </input><br/>
        <span style="display:block; height:10px"></span>
       <textarea id='msgArea' rows="34" cols="36" style="font-family:Consolas;
         font-size:12px"></textarea>
    </div>
    <div style="float:left; width:10px; height:600px"></div>
    <div id='mapDiv' style="float:left; width:700px; height:600px;
      border:1px solid red;"></div>
    <br style="clear: left;" />

   <script type='text/javascript'
     src='http://www.bing.com/api/maps/mapcontrol?callback=GetMap'
     async defer>
    </script>

  </body>
</html>
```

The demo application sets up seven global script-scope objects:

```
var map = null;
var ppLayer = null;
var hmLayer = null;
var reader = null;
var locs = [];
var cGrad = { '0.0': 'black', '0.2': 'purple', '0.4': 'blue',
  '0.6': 'green', '0.8': 'yellow', '0.9': 'orange',
  '1.0': 'rgb(255,0,0)' };
var hmOptions = { intensity: 0.65, radius: 7, colorGradient: cGrad };
```

Object **ppLayer** is a map layer for holding ordinary pushpins, in this case a single pushpin that marks the map center. Object **hmLayer** is a map layer for holding the single heat map. Object **reader** is an HTML **FileReader** object to **read()** the source data. Array object **locs** holds all the lat-lon data from the source data file.

Object **cGrad** is a generic JavaScript key-value object that defines the color gradient for a heat map. The keys are values between 0.0 and 1.0 that control heat map colors. The values are colors that can be defined using strings or using the **rgb()** or **rgba()** functions.

Object **hmOptions** defines the heat map options. The **intensity** key can take a value between 0.0 and 1.0 in which larger values produce a brighter heat map. If omitted, the default value is 0.5. The **radius** key defines the size of each data point that defines the heat map, and the default value is 10 pixels. Larger values produce a heat map with fewer gradations.

As with the demo, the **colorGradient** key accepts a program-defined color gradient. The colorGradient is optional and, if omitted, the default values used are:

```
{
  '0.00': 'rgb(255,0,255)', // Magenta
  '0.25': 'rgb(0,0,255)',   // Blue
  '0.50': 'rgb(0,255,0)',   // Green
  '0.75': 'rgb(255,255,0)', // Yellow
  '1.00': 'rgb(255,0,0)'    // Red
}
```

There is also an optional **opacity** key, which is not used by the demo. The default value is 1.0, which will create no-see-through colors—values such as 0.5 allow you to see labeling under the heat map. There is an optional **unit** key that defines the units used for values of the radius key. The default is the string value "pixels." When the map area is very large (entire earth), the single alternative is "meters."

Table 3: The HeatMapLayerOptions Object

Key	Value
colorGradient	colors, optional, default = visible spectrum
intensity	(0.0 to 1.0) optional, default = 0.5
opacity	(0.0 to 1.0) optional, default = 1.0
radius	optional, default = 10 (pixels)
unit	optional, ["pixels" (default), "meters"]

The demo application loads the **map** object asynchronously by calling the program-defined **GetMap()** function in the usual way. If you want to display the **colorGradient** object at load time, you can do so with code similar to this:

```
for (key in cGrad) {
  WriteLn(key + " " + cGrad[key]);
}
```

The button control labeled "Show Heat Map" has ID property "button1" and the onclick attribute points to function **Button1_Click()**, which is defined as:

```
function Button1_Click()
{
  if (reader == null || locs.length == 0) {
    LoadLocs();
  }
}
```

So, function **Button1_Click()** is merely a wrapper to a call to a program-defined **LoadLocs()** function. There's no technical reason why the code in **LoadLocs()** can't be placed directly inside function **Button1_Click()**, but the two-function approach gives you a bit more flexibility.

The checks to the **reader** object and the **locs.length** property will become clear when we address function **Button2_Click()**. Note that in many situations, you'd want to check the length property as:

```
if (reader == null || locs == null || locs.length == 0) {
```

The application logic, however, ensures that the **locs** array will never be null. Function **LoadLocs()** does most of the work. The function code is:

```
function LoadLocs()
{
  var f = file1.files[0];
  WriteLn('Loading data from ' + f.name + "\n");
  reader = new FileReader();
  reader.onload = function(e) {
    // parse each line, create Location, store into locs

    Microsoft.Maps.loadModule('Microsoft.Maps.HeatMap', function () {
      hmLayer = new Microsoft.Maps.HeatMapLayer(locs, hmOptions);
      map.layers.insert(hmLayer);
    });
  }
  reader.readAsText(f);
}
```

Here's the key idea—when the FileReader onload event fires, control is transferred to the anonymous callback function that parses each line of the data file and populates the **locs** array. The **locs** array is fed to the HeatMap callback, creating the **hmLayer** object that is then inserted onto the map.

The line-parsing code is:

```
var lines = reader.result.split('\n');
for (var i = 0; i < lines.length; ++i) {
  var line = lines[i];
  var tokens = line.split('\t');
  try {
    var loc = new Microsoft.Maps.Location(tokens[12], tokens[13]);
    locs[i] = loc;
  }
  catch (err)  {
    WriteLn(err + " " + line);
  }
}
```

A lot can go wrong when we parse a text file, so the demo uses a try-catch when constructing each **Location** object. The source data files look like this:

```
CA  602000  2409704  Anaheim city    ( . . )  33.855497  -117.760071
CA  602028  2628706  Anchor Bay CDP  ( . . )  38.812653  -123.570267
. . .
```

Each line has 14 tab-delimited values. The first value is the state abbreviation. The next two fields are IDs. The fourth field is the place name, which can be a city, a town, or a Census Designated Place (CDP). Next are eight other fields, including U.S. census population count and land area. The last two fields are the latitude and longitude. You can get files CA_Cities.txt and NV_Cities.txt from https://github.com/jdmccaffrey/bing-maps-v8-succinctly.

The button control labeled "Clear Heat Map" is associated with function **`Button2_Click()`**, which is defined:

```
function Button2_Click()
{
  WriteLn('Clearing heat map' + "\n");
  hmLayer.clear();
  reader = null;
  locs = [];
}
```

The **`clear()`** method of the **`HeatMapLayer`** class deletes a heat map layer. Useful alternatives are the **`hide()`** and **`show()`** methods.

The **`FileReader`** object is set to null in order to force a file read the next time the **`LoadLocs()`** function is called. And the **`locs`** array is reset to an empty array so that the next file read doesn't append to the existing locations.

In summary, in order to create a heat map, you use the **`HeatMapLayer`** object in the **`Microsoft.Maps.HeatMap`** module. The **`HeatMapLayer`** object requires an array of **`Location`** objects and, optionally, a **`HeatMapayerOptions`** object to customize the color gradient and other visualization characteristics.

Resources

For detailed information about the HeatMapLayer class, see: https://msdn.microsoft.com/en-us/library/mt712811.aspx.

For detailed information about the HeatMapLayerOptions class, see: https://msdn.microsoft.com/en-us/library/mt712810.aspx.

You can see some example custom-color gradients at: https://msdn.microsoft.com/en-us/library/mt712854.aspx.

The source data files in .zip format can be found at the U.S. Census website at: https://www.census.gov/geo/maps-data/data/gazetteer2010.html.

You can find interesting map data at the U.S. Geological Survey website at: http://nationalmap.gov/.

Chapter 4 Advanced Techniques

This chapter presents some relatively advanced Bing Maps V8 techniques for geo-applications. In section 4.1, you'll learn how to create automatic pushpin clustering for use when you have a large number of data points. You'll also learn how to use functions such as **`getLocations()`** and **`getPolygons()`** in the Maps.TestDataGenerator module in order to generate random data during development.

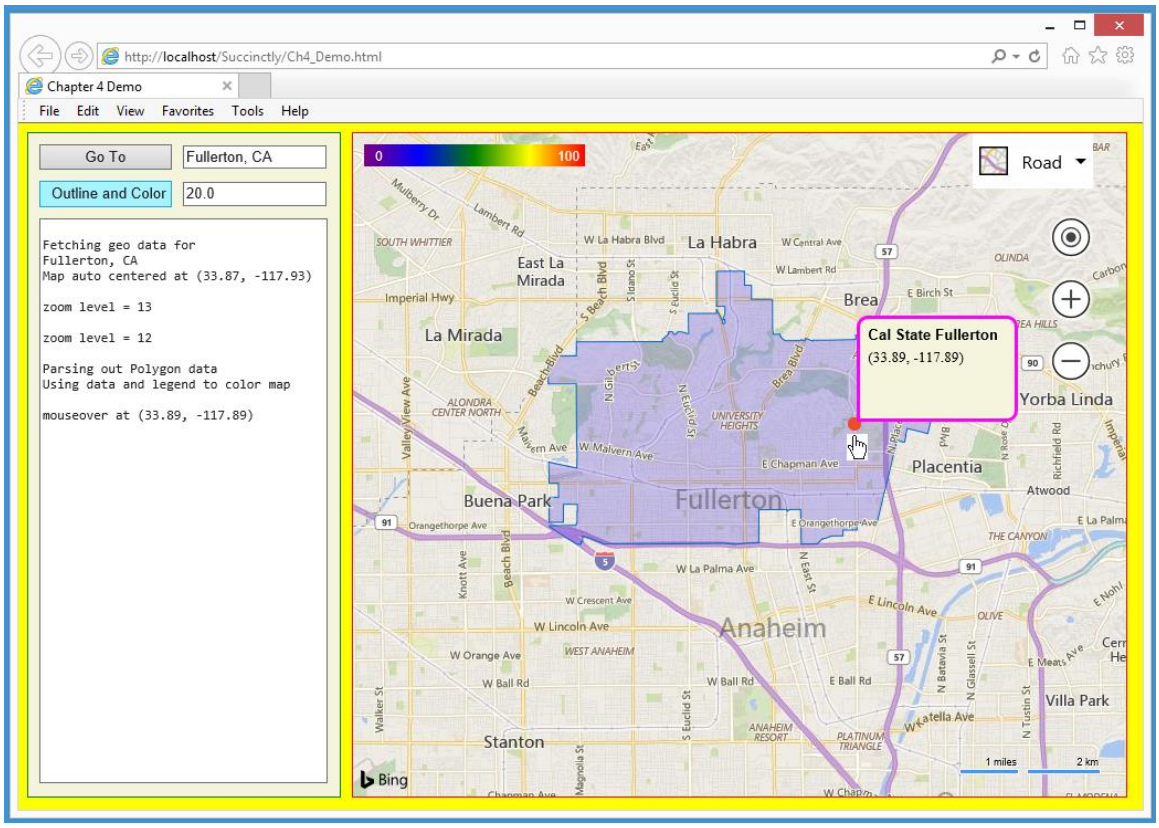

Figure 20: Advanced Techniques Demo

In section 4.2, you'll learn how to use functions such as **`SearchManager.geocode()`** from the Maps.Search module in order to read data. You'll also learn how to use functions such as **`GeoDataAPIManager.getBoundary()`** from the Maps.SpatialDataService module in order to parse and manipulate data.

In section 4.3, you'll learn how to create a color-gradient legend and how to use a legend to convert a numeric value from 0 to 100 into a corresponding color. You'll also learn how to use gradient-legend colors to shade areas on a map.

In section 4.4, you'll learn how to create custom-styled **`Infobox`** objects. You'll also learn how to generate reproducible pseudorandom numbers using the Lehmer algorithm (sometimes called the Park-Miller algorithm).

4.1 Clustered pushpins

Dealing with large numbers of data points on a map application can be challenging. One of the most powerful features of the Bing Maps V8 library is its ability to cluster data points as **`pushpin`** objects. The feature is best explained with a visual example.

The demo web application in Figure 21 loads a map centered near Indianapolis, Ind., with an initial zoom level set to 10.

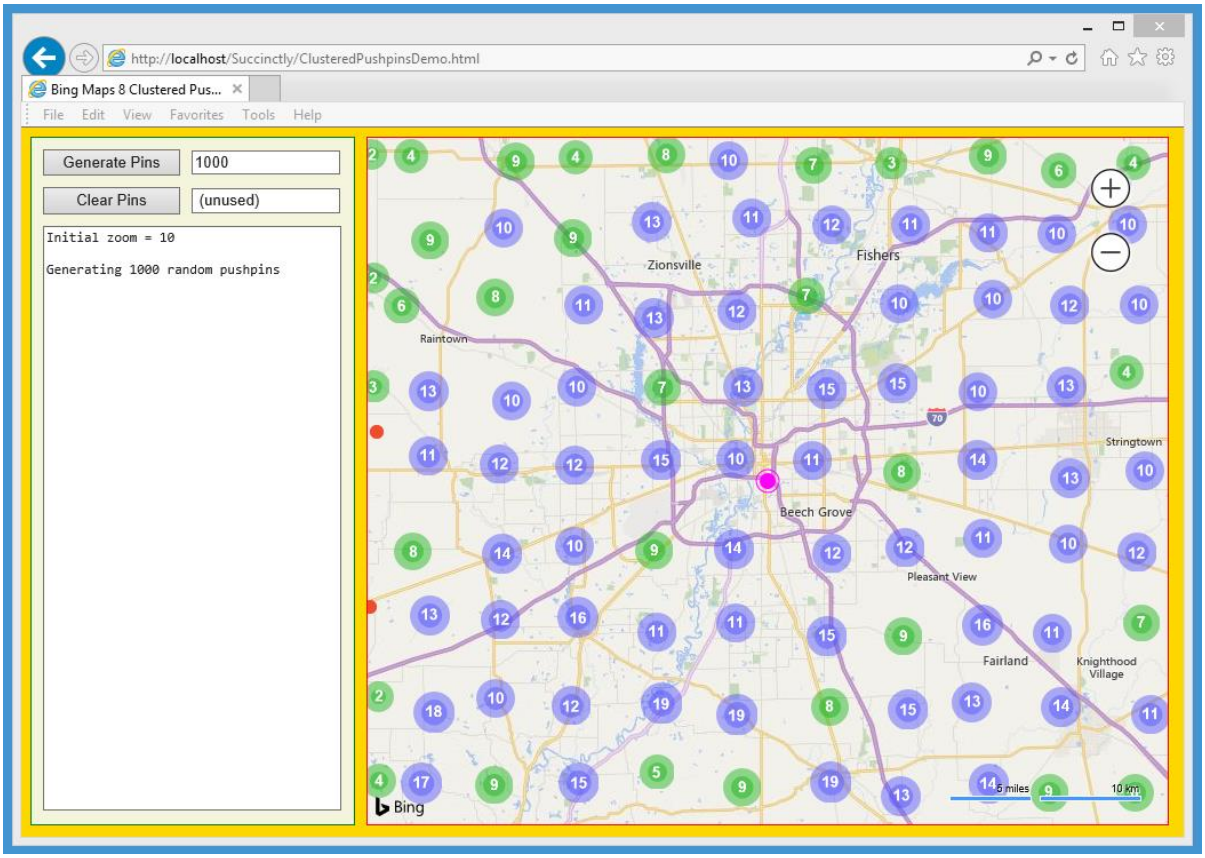

Figure 21: Clustered Pushpins Demo Initial View

When the user clicked “Generate Pins,” the application created 1000 pushpins with random locations within the initial map boundaries. The code logic is set up so that the map is divided into approximately 10 by 10 = 100 grid squares. A green circle icon indicates a grid has two to nine pushpins at that location. Individual pushpins are orange—you can see two of them at the left edge of the map. A blue circle means there are 10 to 99 pushpins.

Notice that most of the labels on the map, such as “Indianapolis,” are not visible. Behind the scenes, the Bing Maps library uses label-collision detection to automatically move or remove labels.

Next, the user clicked the “+” zoom control to get zoom level 12, so that most of the individual pushpins are visible and can be clicked. Notice that the “Indianapolis” label is now visible.

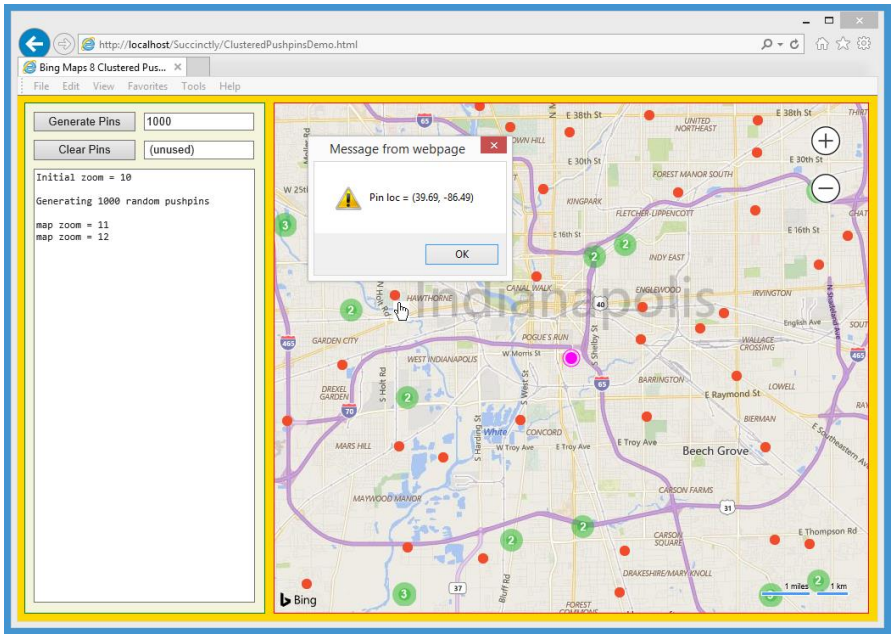

Figure 22: Clustered Pushpins Demo Zoom In

Next, the user clicked the "-" zoom control five times to zoom out to level 7. A red circle indicates there are 100 or more pushpins. In Figure 23, four clusters have counts that sum to 1000.

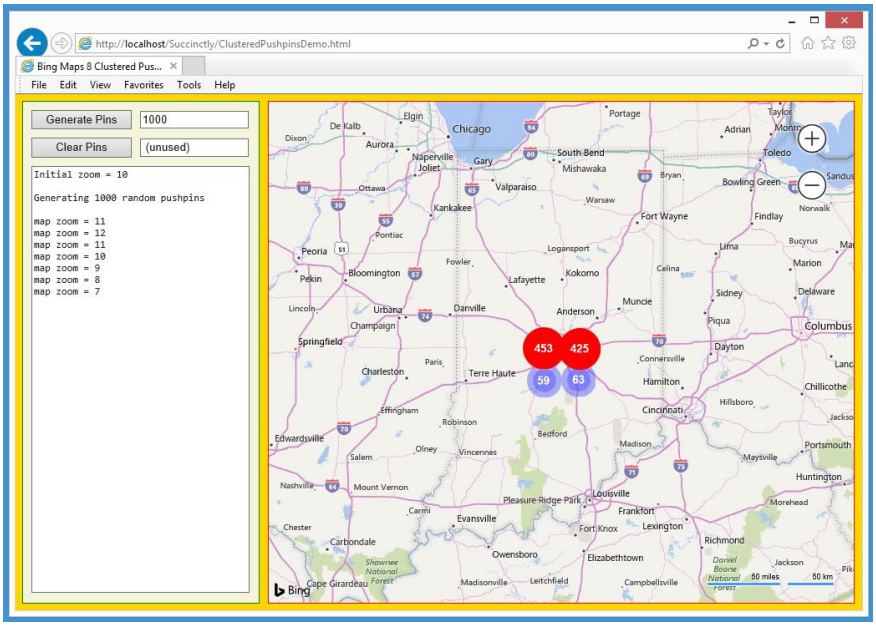

Figure 23: Cluster Pushpins Demo Zoom Out

The demo application is named ClusteredPushpinsDemo.html and is defined in a single file.

Code Listing 9: ClusteredPushpinsDemo.html

```
<!DOCTYPE html>
<!-- ClusteredPushpinsDemo.html -->

<html>
  <head>
    <title>Bing Maps 8 Clustered Pushpins</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>

    <script type="text/javascript">

    var map = null;
    var pushpins = [];
    var ppLayer = null;  // ordinary pushpin layer
    var clusterLayer = null;  // for pushpin clustering

    var orangeDot = '' +
      'AAAAMCAYAAABWdVznAAAAGXRFWHRTb2Z0d2FyZQBBZG9iZSBJbWFnZVJlYWR5cc' +
      'llPAAAAIxJREFUeNpiZICCD766CkCqHogDgFgAJgzEG4C4UWDz5QcgAUaoYgMgt' +
      'R9JIToAaXQEarrACDX5PB7FyJoMmaDOIKSYAaqmngnqZmJBABORpsNtYWIgETBB' +
      'PUMs+MAEDWdiwQaQhkYibQGpaWSCxqAjAU2wiHsA9jQoBkGRAsQL0DR+gIoZQtU' +
      'wAAQYAGgpKDzqLFoIAAAAAElFTkSuQmCC';

    function GetMap()
    {
      var options = {
        credentials: "Anw _ _ _ 3xt",
        center: new Microsoft.Maps.Location(39.75, -86.15), // Indy
        mapTypeId: Microsoft.Maps.MapTypeId.road,
        zoom: 10,
        enableClickableLogo: false,
        showCopyright: false,
        showMapTypeSelector: false,
        showLocateMeButton: false
      };

      var mapDiv = document.getElementById("mapDiv");
      map = new Microsoft.Maps.Map(mapDiv, options);

      WriteLn('Initial zoom = ' + map.getZoom());
      Microsoft.Maps.Events.addHandler(map, 'viewchangeend', ViewChanged);

      ppLayer = new Microsoft.Maps.Layer();
      var cpp = new Microsoft.Maps.Pushpin(map.getCenter(),
        { color: "fuchsia" });
      ppLayer.add(cpp);
      map.layers.insert(ppLayer);
    }

    function ViewChanged(e)
    {
      var z = map.getZoom();
      WriteLn('map zoom = ' + z);
```

```
}

function WriteLn(txt)
{
  var existing = msgArea.value;
  msgArea.value = existing + txt + "\n";
}

function Button1_Click()
{
  var numPins = parseInt(textbox1.value);
  var mb = map.getBounds();

  var tdg = Microsoft.Maps.TestDataGenerator;
  var locs = tdg.getLocations(numPins, mb);
  WriteLn("\nGenerating " + numPins + " random pushpins \n");

  var n = locs.length;
  var ppOptions = { icon: orangeDot,
    anchor: new Microsoft.Maps.Point(4,4) };
  for (var i = 0; i < n; ++i) {
    var pp = new Microsoft.Maps.Pushpin(locs[i], ppOptions);
    pp.meta = locs[i];
    Microsoft.Maps.Events.addHandler(pp, 'click',
      function() {
        alert( "Pin loc = " + LatLonStr(pp.meta) );
      }
    );
    pushpins[i] = pp;
  }

  // Add the pushpins to a cluster layer.
  Microsoft.Maps.loadModule('Microsoft.Maps.Clustering',
    function() {
      clusterLayer = new Microsoft.Maps.ClusterLayer(pushpins, {
        clusteredPinCallback: MakeClusterPins,
        gridSize: 70  // 70 pixels = ~1/10 of map width
    });
    map.layers.insert(clusterLayer);
  });
}

function MakeClusterPins(clusterDot)
{
  // Customize the pushpin/dot that represents a cluster.
  var minRadius = 12;
  var outlineWidth = 7;
  var count =  clusterDot.containedPushpins.length;
  var radius = 2.2 * Math.log(count) + minRadius;

  var fillColor = null;

  if (count >= 100) {
    fillColor = 'red';  // solid red
```

```
    }
    else if (count >= 10 && count <= 99) {
      fillColor = 'rgba(80, 80, 255, 0.45)';  // blue-ish
    }
    else if (count >= 2 && count <= 9) {
      fillColor = 'rgba(20, 180, 20, 0.45)';  // green-ish
    }

    var img = '<svg xmlns="http://www.w3.org/2000/svg" width="' +
      (radius * 2) + '" height="' + (radius * 2) + '">' +
      '<circle cx="' + radius + '" cy="' + radius + '" r="' +
      radius + '" fill="' + fillColor + '"/>' +
      '<circle cx="' + radius + '" cy="' + radius + '" r="' +
      (radius - outlineWidth) + '" fill="' + fillColor + '"/>' +
      '</svg>';

    clusterDot.setOptions({
      icon: img,
      anchor: new Microsoft.Maps.Point(radius, radius),
      textOffset: new Microsoft.Maps.Point(0, radius - 8)
    });

  }

  function LatLonStr(loc)
  {
    var s = "(" + Number(loc.latitude).toFixed(2) + ", " +
      Number(loc.longitude).toFixed(2) + ")";
    return s;
  }

  function Button2_Click(e)
  {
    WriteLn("Clearing all clustered pushpins");
    clusterLayer.clear();
  }

  </script>
</head>

<body style="background-color:gold">
  <div id='controlPanel' style="float:left; width:262px; height:580px;
    border:1px solid green; padding:10px; background-color: beige">

    <input id="button1" type='button' value='Generate Pins'
      style="width:120px;" onclick="Button1_Click();"></input>
    <div style="width:2px; display:inline-block"></div>
    <input id="textbox1" type='text' size='16'
      value='1000'></input><br/>
    <span style="display:block; height:10px"></span>

    <input id="button2" type='button' value='Clear Pins'
      style="width:120px;" onclick="Button2_Click();"></input>
    <div style="width:2px; display:inline-block"></div>
```

```
      <input id="textbox2" type='text' size='16'
        value=' (unused) '></input><br/>
      <span style="display:block; height:10px"></span>

      <textarea id='msgArea' rows="36" cols="36" style="font-family:Consolas;
        font-size:12px"></textarea>
    </div>
    <div style="float:left; width:10px; height:600px"></div>
    <div id='mapDiv' style="float:left; width:700px; height:600px;
      border:1px solid red;"></div>
    <br style="clear: left;" />

    <script type='text/javascript'
      src='http://www.bing.com/api/maps/mapcontrol?callback=GetMap'
      async defer>
    </script>

  </body>
</html>
```

The demo sets up five script-global objects:

```
var map = null;
var pushpins = [];
var ppLayer = null;  // ordinary pushpin layer
var clusterLayer = null;  // for pushpin clustering

var orangeDot = '' +
  'AAAAMCAYAAABWdVznAAAAGXRFWHRTb2Z0d2FyZQBBZG9iZSBJbWFnZVJlYWR5cc' +
  'llPAAAAIxJREFUeNpiZICCD766CkCqHogDgFgAJgzEG4C4UWDz5QcgAUaoYgMgt' +
  'R9JIToAaXQEarrACDX5PB7FyJoMmaDOIKSYAaqmngnqZmJBABORpsNtYWIgETBB' +
  'PUMs+MAEDWdiwQaQhkYibQGpaWSCxqAjAU2wiHsA9jQoBkGRAsQL0DR+gIoZQtU' +
  'wAAQYAGgpKDzqLFoIAAAAAElFTkSuQmCC';
```

When using pushpin clustering, typically you create one layer for the individual pushpins and a second layer for the clusters that represent multiple pushpins. The **orangeDot** object is a Base64-encoded PNG image of an orange circle with radius six pixels. A more common alternative is to use an ordinary image. For example:

```
var orangeDot = "./images/12x12Orange.png";
```

The map is loaded with a **zoom: 10** option and the **showCopyright**, **showMapTypeSelector**, and **showLocateMeButton** options are set to **false** in order to keep the map visually clean. A default-style ordinary pushpin with **color: fuchsia** is placed at the center to act as a point of reference.

The map **viewchangeend** event handler is modified so that control will be transferred to a program-defined function named **ViewChanged()** in order for the zoom level to be logged.

Function **ViewChanged()** is defined:

```
function ViewChanged(e)
{
  var z = map.getZoom();
  WriteLn('map zoom = ' + z);
}
```

The **viewchangeend** event is triggered by several actions in addition to a zoom change. Alternatively, we can use a global object named something like **zoomLev** and write a message only when the new value of **getZoom()** differs from the current **zoomLev** value.

The button control labeled “Get Pushpins” is associated with function **Button1_Click()**:

```
function Button1_Click()
{
  var numPins = parseInt(textbox1.value);
  var mb = map.getBounds();
  var tdg = Microsoft.Maps.TestDataGenerator;
  var locs = tdg.getLocations(numPins, mb);
  WriteLn("\nGenerating " + numPins + " random pushpins \n");
. . .
```

The key idea here is the use of the static **getLocations()** function in the Maps.TestDataGenerator module. That function expects a number of **Location** objects to generate and a **Bounds** object that limits the range of the latitudes and longitudes of the returned array of **Location** objects.

The TestDataGenerator module includes a minor weakness—it uses the built-in JavaScript **Math.random()** function that cannot be seeded, which means you’ll get different results every time **getLocations()** is called, even when you want reproducible results. Alternatively, you can write your own seedable, random-number generator that can be seeded and use it to generate **Location** objects.

The pushpins with random locations are created with this code:

```
var n = locs.length;
var ppOptions = { icon: orangeDot, anchor: new Microsoft.Maps.Point(4,4) };
for (var i = 0; i < n; ++i) {
  var pp = new Microsoft.Maps.Pushpin(locs[i], ppOptions);
  pp.meta = locs[i];
  Microsoft.Maps.Events.addHandler(pp, 'click', function() {
    alert( "Pin loc = " + LatLonStr(pp.meta) );
  });
  pushpins[i] = pp;  // store into global array
}
```

Each pushpin has its click event handler modified so that it will display its location, formatted using **LatLonStr()**, in a simple alert box. An alternative is to use an **Infobox** object.

After the pushpins have been created and stored, they are added to the cluster layer:

```
. . .
  Microsoft.Maps.loadModule('Microsoft.Maps.Clustering',
    function() {
      clusterLayer = new Microsoft.Maps.ClusterLayer(pushpins, {
        clusteredPinCallback: MakeClusterPins,
        gridSize: 70
    });
    map.layers.insert(clusterLayer);
  });
} // end Button1_Click()
```

The anonymous callback function uses the **ClusterLayer()** constructor to add all the pushpins to the global **clusterLayer** object. The **clusteredPinCallback** property points to a function **MakeClusterPins()** that will define the size and color of the cluster icons. The **gridSize** property is set to 70 pixels, which is one-tenth the width of the map. Alternatively, you can set the grid size programmatically using the **Map.getWidth()** function.

Function **MakeClusterPins()** starts by setting the size of the cluster icons:

```
function MakeClusterPins(clusterDot)
{
  var minRadius = 12;
  var outlineWidth = 7;
  var count = clusterDot.containedPushpins.length;
  var radius = 2.2 * Math.log(count) + minRadius;
. . .
```

Calculating the radius of the cluster icon based on the count of the number of pushpins it represents can be somewhat tricky. The idea is to set an absolute minimum radius, then add a bonus for icons that represent a large number of pushpins. The **Math.log()** function returns the log to the base **e** of its argument. The log of the count of pushpins will be a value of no more than roughly 9.5, and the 2.2 factor was determined by a bit of trial and error. Instead of calculating a variable radius, you can simply use a fixed value.

Next, the color of the cluster icon is determined:

```
var fillColor = null;

if (count >= 100) {
  fillColor = 'red';  // solid red
}
else if (count >= 10 && count <= 99) {
  fillColor = 'rgba(80, 80, 255, 0.45)';  // blue-ish
}
else if (count >= 2 && count <= 9) {
  fillColor = 'rgba(20, 180, 20, 0.45)';  // green-ish
}
```

If you look at the figures at the beginning of this section, you'll see that that blue and green cluster icons are two concentric circles. By using the **rgba()** function and setting the alpha value to 0.45, the outer circle will be a bit transparent. When the inner circle is drawn, it will appear nearly solid. An alternative to the **rbga()** function is the Maps.Color class.

Next, the double-circle icon image is created using SVG:

```
var img = '<svg xmlns="http://www.w3.org/2000/svg" width="' +
  (radius * 2) + '" height="' + (radius * 2) + '">' +
  '<circle cx="' + radius + '" cy="' + radius + '" r="' +
  radius + '" fill="' + fillColor + '"/>' +
  '<circle cx="' + radius + '" cy="' + radius + '" r="' +
  (radius - outlineWidth) + '" fill="' + fillColor + '"/>' +
  '</svg>';
```

Alternatively, we can define a function along the lines of **CreateSvgConcentric()**. Another alternative is to define the icon using an HTML canvas object.

The callback function concludes by creating the cluster icon:

```
. . .
  clusterDot.setOptions({
    icon: img,
    anchor: new Microsoft.Maps.Point(radius, radius),
    textOffset: new Microsoft.Maps.Point(0, radius - 8)
  });
}
```

The **textOffset** property takes into account the height of the count text in the center of the cluster icon.

In summary, you create clustered pushpins by first creating an array of pushpins and a **Layer** object for the pushpins in the usual way. Next, you define an anonymous function in the Maps.Clustering module that calls the **ClusterLayer()** constructor, which in turn optionally sets a callback function that defines the size and color of a cluster icon.

Resources

For detailed information about cluster layer options, see:
https://msdn.microsoft.com/en-us/library/mt712813.aspx.

For further detailed information about the cluster layer class, see:
https://msdn.microsoft.com/en-us/library/mt712808.aspx.

For information about using the Maps.Color class instead of the **rgba()** function, see:
https://msdn.microsoft.com/en-us/library/mt712639.aspx.

4.2 Geosearch

The Bing Maps V8 library has many functions that allow you to query the Bing Maps Spatial Data Services. You can think of the Bing Maps Spatial Data Services as a huge geodatabase that can be queried by using a RESTful URL. The Bing Maps library has wrapper functions that make calling into the Bing Maps Spatial Data Services quite a bit easier than calling the services directly.

The demo web application in Figure 24 loads a map centered near the border of Nebraska and Kansas with an initial zoom level set to 3.

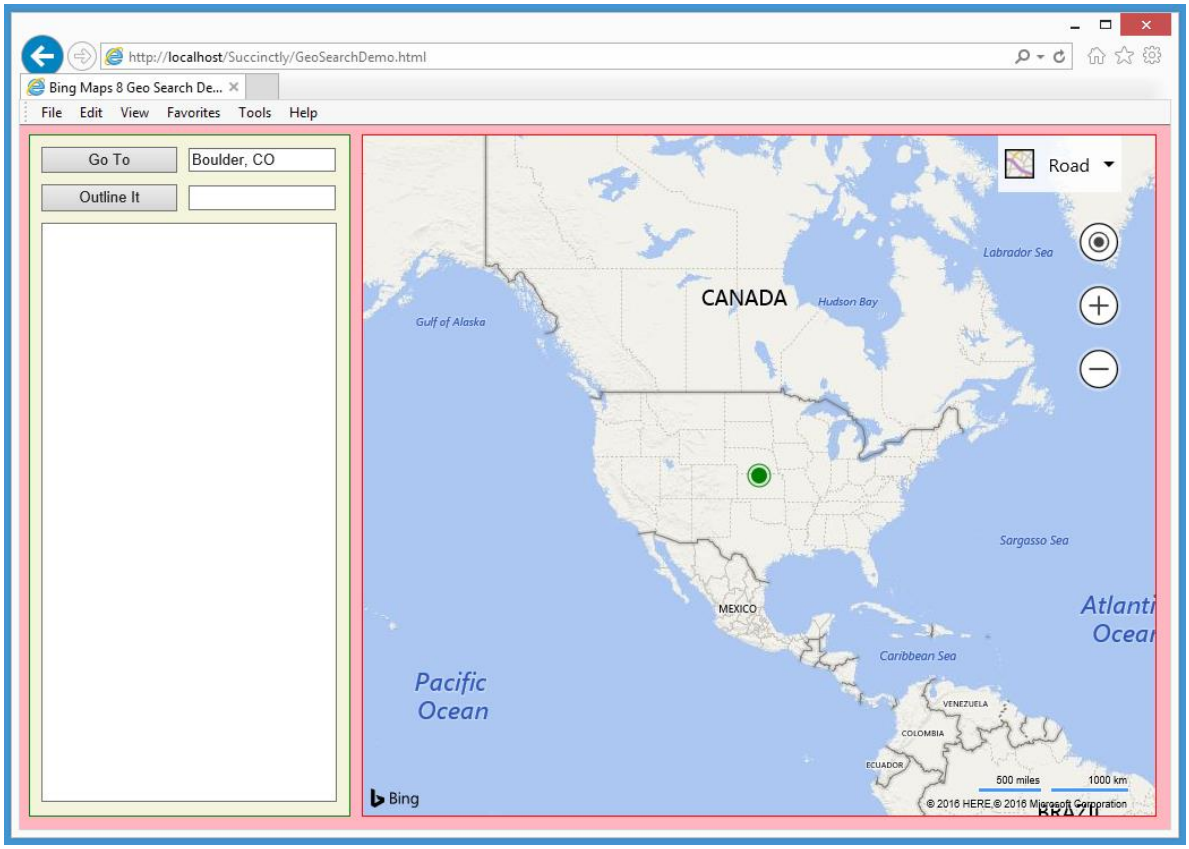

Figure 24: Initial Map View for Geosearch Demo

The application places a default-style, green pushpin at the map center to act as a point of reference. The top-most text box control is prepopulated with “Boulder, CO,” but it is editable.

When the user clicked “Go To,” the application loaded the Map.Search module behind the scenes and sent a request to the Bing Maps Spatial Data Services for information about the specified location.

When the information about Boulder was returned, the application saved the results for later use, then automatically determined a “best view.”

In this example, the user determined the best view for Boulder, CO, to be a zoom level of 12 and a map center location of (40.02, -105.28).

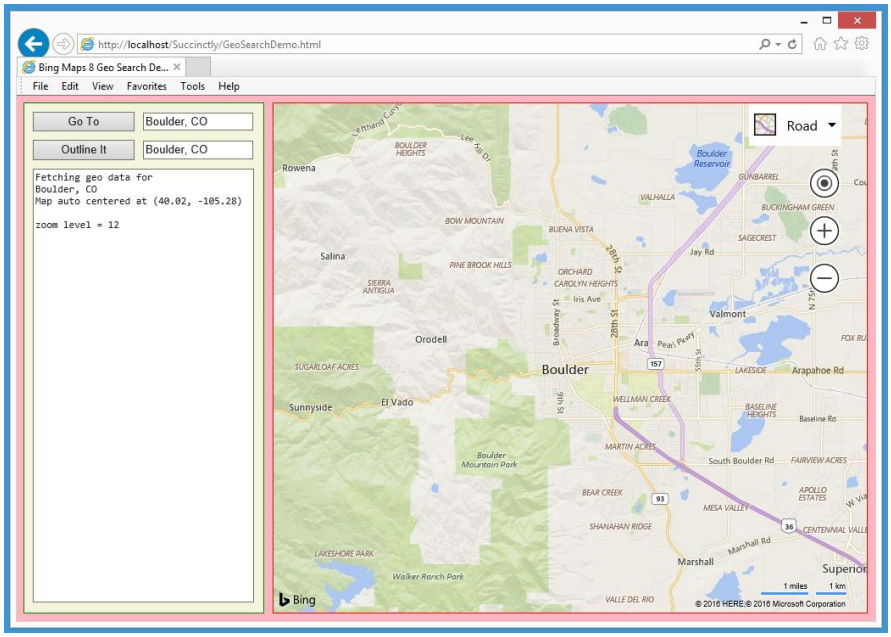

Figure 25: Get Data for a Populated Place

Next, the user clicked “Outline It.” The application loaded the Maps.SpatialDataService module, then used it to color the boundaries of Boulder, CO.

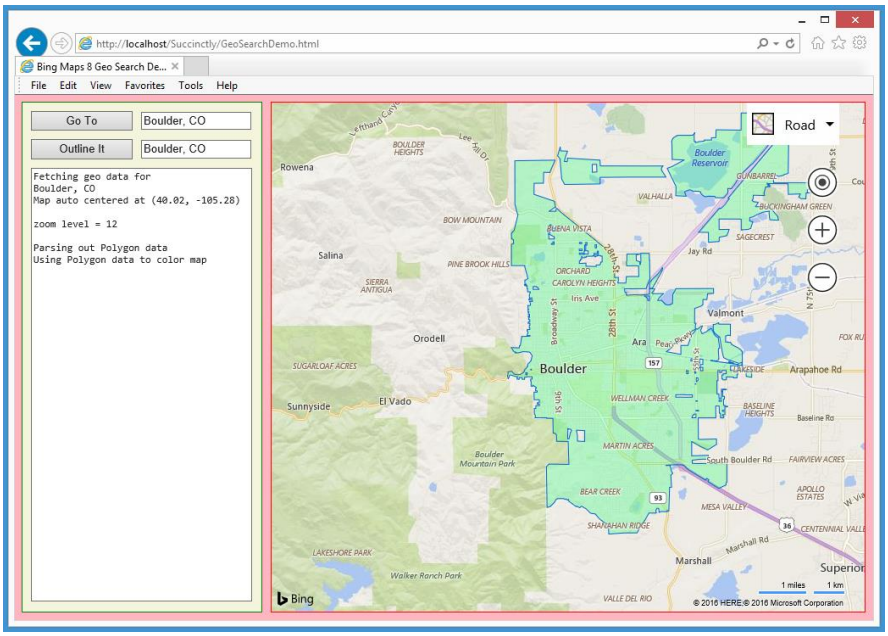

Figure 26: Color Polygon Information for a Populated Place

The demo application is named GeoSearchDemo.html and is defined in a single file.

Code Listing 10: GeoSearchDemo.html

```
<!DOCTYPE html>
<!-- GeoSearchDemo.html -->

<html>
  <head>
    <title>Bing Maps 8 Geo Search Demo</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>

    <script type="text/javascript">

      var map = null;
      var ppLayer = null;
      var geoShapeLayer = null;
      var geoResult = null;

    function GetMap()
    {
      var options = {
        credentials: "Anw _ _ _ 3xt",
        center: new Microsoft.Maps.Location(39.80, -98.60),
        mapTypeId: Microsoft.Maps.MapTypeId.road,
        zoom: 3,
        enableClickableLogo: false,
        showTermsLink: false
      };

      var mapDiv = document.getElementById("mapDiv");
      map = new Microsoft.Maps.Map(mapDiv, options);

      Microsoft.Maps.Events.addHandler(map, 'viewchangeend', ViewChanged);

      ppLayer = new Microsoft.Maps.Layer();
      var cpp= new Microsoft.Maps.Pushpin(map.getCenter(), {color: 'green'});
      ppLayer.add(cpp);
      map.layers.insert(ppLayer);

      geoShapeLayer = new Microsoft.Maps.Layer();
    }

    function ViewChanged(e)
    {
      var z = map.getZoom();
      WriteLn('\nzoom level = ' + z);
    }

    function WriteLn(txt)
    {
      var existing = msgArea.value;
      msgArea.value = existing + txt + "\n";
    }

    function Button1_Click(e)
```

```
{
  var city = textbox1.value;
  textbox2.value = city;  // echo
  WriteLn("Fetching geo data for \n" + city.toString());

  Microsoft.Maps.loadModule('Microsoft.Maps.Search', function() {
    var searchManager = new Microsoft.Maps.Search.SearchManager(map);
    var geoRequest = {
      where: city,
      callback: function(res) {
        geoResult = res;  // Save for use by the outliner function.
        if (res && res.results && res.results.length > 0) {
          map.setView({ bounds: res.results[0].bestView });
          WriteLn("Map auto centered at " +
            LatLonStr(map.getCenter()));
        }
      }
    };
    searchManager.geocode(geoRequest);  // Do an async request.
  });
}

function Button2_Click(e)
{
  var city = textbox1.value;
  WriteLn('\nParsing out Polygon data');
  WriteLn('Using Polygon data to color map');

  var geoOptions = { entityType: 'PpopulatedPlace',
    getAllPolygons: true };

  Microsoft.Maps.loadModule('Microsoft.Maps.SpatialDataService',
    function() {
      var apimgr = Microsoft.Maps.SpatialDataService.GeoDataAPIManager;
      apimgr.getBoundary(geoResult.results[0].location, geoOptions, map,
        function(data) {
          if (data.results && data.results.length > 0) {
            geoShapeLayer.add(data.results[0].Polygons);
            map.layers.insert(geoShapeLayer);
          }
        });
    });
}

function LatLonStr(loc)
{
  var s = "(" + Number(loc.latitude).toFixed(2) + ", " +
    Number(loc.longitude).toFixed(2) + ")";
  return s;
}

</script>
</head>
```

```
  <body onload="GetMap();" style="background-color:lightpink">
    <div id='controlPanel' style="float:left; width:262px; height:580px;
      border:1px solid green; padding:10px; background-color: beige">

      <input id="button1" type='button' value='Go To' style="width:120px;"
        onclick="Button1_Click();"></input>
      <div style="width:2px; display:inline-block"></div>
      <input id="textbox1" type='text' size='16' value='Boulder, CO'>
        </input><br/>
      <span style="display:block; height:10px"></span>

      <input id="button2" type='button' value='Outline It'
        style="width:120px;" onclick="Button2_Click();"></input>
      <div style="width:2px; display:inline-block"></div>
      <input id="textbox2" type='text' size='16' value=''></input><br/>
      <span style="display:block; height:10px"></span>

      <textarea id='msgArea' rows="36" cols="36" style="font-family:Consolas;
        font-size:12px"></textarea>
    </div>
    <div style="float:left; width:10px; height:600px"></div>
    <div id='mapDiv' style="float:left; width:700px; height:600px;
      border:1px solid red;"></div>
    <br style="clear: left;" />

    <script type='text/javascript'
      src='http://www.bing.com/api/maps/mapcontrol?callback=GetMap'
      async defer>
    </script>

  </body>
</html>
```

The demo sets up four script-global objects:

```
var map = null;
var ppLayer = null;
var geoShapeLayer = null;
var geoResult = null;
```

The two **Layer** objects are declared globally even though they're only used by a single function, **GetMap()** and **Button2_Click()**, respectively, to emphasize the idea of map layering. Object **geoResult**, however, is created in function **Button1_Click()**, then accessed in function **Button2_Click()** so that it's legitimately global.

The event handler for the first button control is **Button1_Click()**. The definition of the function begins:

```
function Button1_Click(e)
{
  var city = textbox1.value;
  textbox2.value = city;
  WriteLn("Fetching geo data for \n" + city.toString());
. . .
```

As you'll see shortly, the Bing Maps Spatial Data Services database will be queried for what's called a populated place. The service does quite a bit of intelligent guessing for ambiguous queries. For example, if the user specifies "Paris," the query will return information about Paris, France, rather than the cities named Paris in Arkansas, Idaho, Illinois, Indiana, Iowa, Kentucky, Maine, Michigan, and Texas.

The search is performed this way:

```
. . .
  Microsoft.Maps.loadModule('Microsoft.Maps.Search', function() {
    var searchManager = new Microsoft.Maps.Search.SearchManager(map);
    var geoRequest = {
      where: city,
      callback: function(res) {
        geoResult = res;  // save for use by the outliner function
        if (res && res.results && res.results.length > 0) {
          map.setView({ bounds: res.results[0].bestView });
          WriteLn("Map auto centered at " +
            LatLonStr(map.getCenter()));
        }
      }
    };
    searchManager.geocode(geoRequest);
  });
}
```

There's quite a bit going on here because of the nested callback functions. The first statement can be interpreted as, "Load the Maps.Search module and, when the load has completed, perform the following actions."

The next statement can be interpreted, "Instantiate a new **`SearchManager`** object on the current **`map`** object." When the search executes, the query will look for data that is identified as a city in the Spatial Data Service, and, if any results are returned, save the results in the global **`geoResult`** object, then use the results to reposition and resize the **`map`** object using the **`setView()`** function.

The code in function **`Button2_Click()`** uses the query results to outline and color the target city. The definition begins:

```
function Button2_Click(e)
{
  var city = textbox1.value;
  WriteLn('\nParsing out Polygon data');
```

```
  WriteLn('Using Polygon data to color map');
  var geoOptions = { entityType: 'PopulatedPlace',
    getAllPolygons: true };
. . .
```

The **SearchManager** object returns a lot of information, so the function sets up a **geoOptions** filter to fetch only data tagged as a **PopulatedPlace** and also to get all polygons. (Many cities have complex boundaries that are made up of multiple polygons.)

The data stored in global object **geoResult** is in WKT format that can be tricky to parse, but the **getBoundary()** function makes your life easier:

```
. . .
  Microsoft.Maps.loadModule('Microsoft.Maps.SpatialDataService',
    function() {
      var apimgr = Microsoft.Maps.SpatialDataService.GeoDataAPIManager;
      apimgr.getBoundary(geoResult.results[0].location, geoOptions, map,
        function(data) {
          if (data.results && data.results.length > 0) {
            geoShapeLayer.add(data.results[0].Polygons);
            map.layers.insert(geoShapeLayer);
          }
        });
    });
}
```

In summary, you can use the SearchManager class of the Maps.Search module to query data from the Bing Maps Spatial Data Services rather than use a raw RESTful URL. You can use the GeoDataAPIManager class of the Maps.SpatialDataService class to parse the returned WKT format data.

Resources

For detailed information about the Maps.SpatialDataService module, see: https://msdn.microsoft.com/en-us/library/mt712849.aspx.

For detailed information about the Maps.Search module, see: https://msdn.microsoft.com/en-us/library/mt712846.aspx.

For information about using the QueryAPIManager class to query Bing services, see: https://msdn.microsoft.com/en-us/library/mt712828.aspx.

For information about using the GeoDataAPIManager class to extract polygons, see: https://msdn.microsoft.com/en-us/library/mt712862.aspx.

4.3 Gradient legends

For some geo-applications, it's useful to create a continuous-color gradient, then use the color values defined by the gradient to color shapes and other map features. The Bing Maps V8 library doesn't have any functions that allow you to directly create color-gradient legends, but it's easy to combine an HTML5 canvas with a custom pushpin to create a gradient legend.

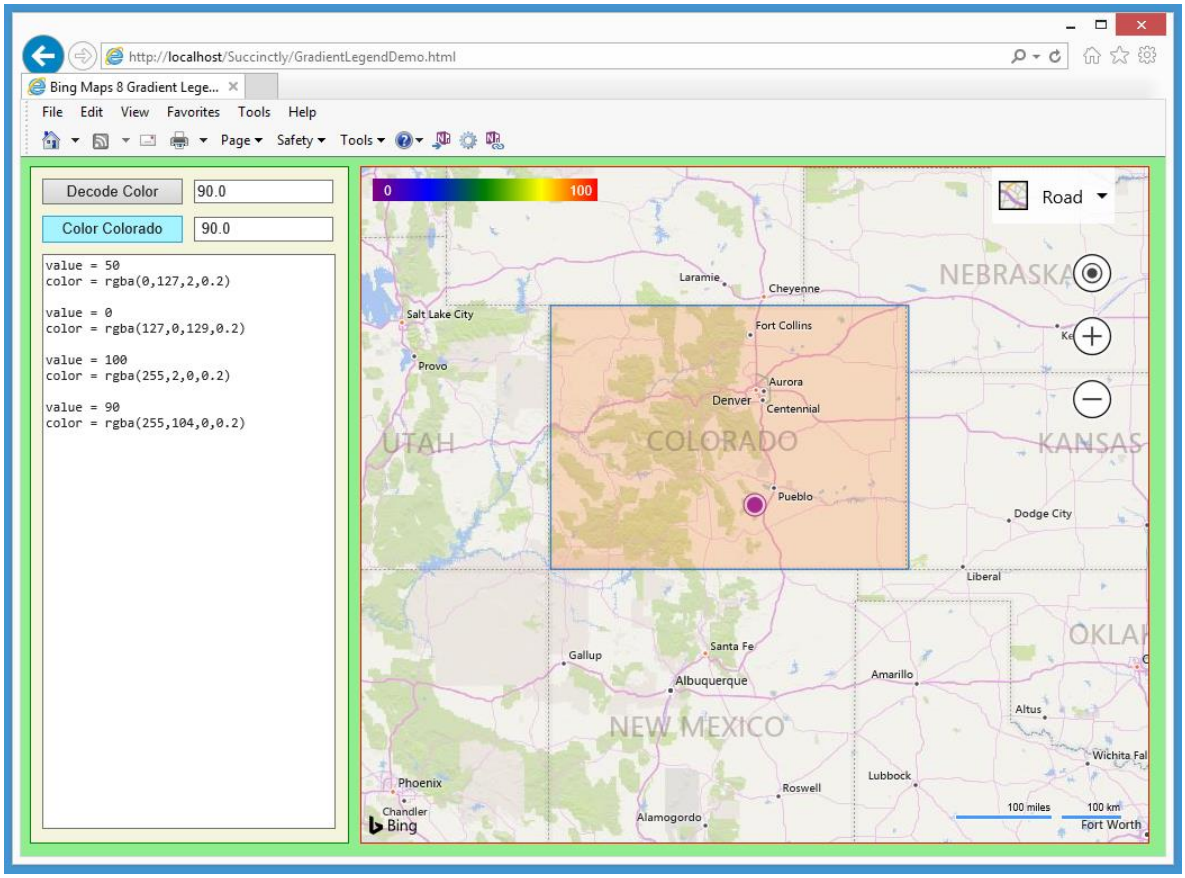

Figure 27: Gradient Legend Demo

The demo web application shown in Figure 27 initially loads a map centered at (38.00, -105.00) and places a default, large purple pushpin at center. Behind the scenes, a color-gradient legend is programmatically created as a custom pushpin and placed in the northwest corner of the map.

The gradient's colors range from purple through green to red, and each color corresponds to a numeric value between 0 (purple) and 100 (red). Whenever the user changes zoom level or scrolls the map, the legend is redrawn in the northwest corner.

The button labeled "Decode Color" displays the color as an RGBA that corresponds to the numeric value in the associated text box control. The button labeled "Color Colorado" does just that, using the color corresponding to the numeric value in the associated text box control.

The demo web application is named GradientLegendDemo.html and is defined in a single file.

Code Listing 11: GradientLegendDemo.html

```
<!DOCTYPE html>
<!-- GradientLegendDemo.html -->

<html>
  <head>
    <title>Bing Maps 8 Gradient Legend Demo</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>

    <script type='text/javascript'>

    var map = null;
    var ppLayer = null;  // pushpin layer
    var legendLayer = null;
    var stateLayer = null;

    var legend = null;
    var legendImageInfo = null;

    var colorado =
      'GEOMETRYCOLLECTION( POLYGON ((-102 41, -109 41, -109 37,' +
      '-102 37, -102 41)) )';

    function GetMap()
    {
      var options = {
        credentials: "Anw _ _ _ 3xt",
        center: new Microsoft.Maps.Location(38.00, -105.00),
        mapTypeId: Microsoft.Maps.MapTypeId.road,
        zoom: 6,
        enableClickableLogo: false,
        showCopyright: false
      };

      var mapDiv = document.getElementById("mapDiv");
      map = new Microsoft.Maps.Map(mapDiv, options);

      Microsoft.Maps.Events.addHandler(map, 'viewchangeend', ViewChanged);

      ppLayer = new Microsoft.Maps.Layer();
      var cpp= new Microsoft.Maps.Pushpin(map.getCenter(), null);
      ppLayer.add(cpp);
      map.layers.insert(ppLayer);

      legendLayer = new Microsoft.Maps.Layer();
      CreateLegend();
      var legendOpt = { icon: legend,
        anchor: new Microsoft.Maps.Point(-10,-10) };
      var bounds = map.getBounds();
      var nw = bounds.getNorthwest();
```

```
    var legendPp = new Microsoft.Maps.Pushpin(nw, legendOpt);
    legendLayer.add(legendPp);
    map.layers.insert(legendLayer);

    stateLayer = new Microsoft.Maps.Layer();
  }

  function ViewChanged(e)
  {
    // Relocate legend.
    legendLayer.clear();
    var legendOpt = { icon: legend,
      anchor: new Microsoft.Maps.Point(-10,-10) };
    var bounds = map.getBounds();
    var nw = bounds.getNorthwest();
    var legendPp = new Microsoft.Maps.Pushpin(nw, legendOpt);
    legendLayer.add(legendPp);
    map.layers.insert(legendLayer);
  }

  function WriteLn(txt)
  {
    var existing = msgArea.value;
    msgArea.value = existing + txt + "\n";
  }

  function Button1_Click()
  {
    // Get color, associate with a value.
    var maxValue = 100;
    var v = parseInt(textbox1.value);
    var color = GetLegendColor(v, maxValue);
    WriteLn("value = " + v);
    WriteLn("color = " + color.toString());
    WriteLn("");
  }

  function CreateLegend()
  {
    // Create, save a legend as png for use as a custom pushpin.
    var c = document.createElement('canvas');
    c.width = 200;
    c.height = 20;
    var ctx = c.getContext("2d");
    var grd = ctx.createLinearGradient(0, 0, c.width, 0);
    grd.addColorStop(0.0, "purple");
    grd.addColorStop(0.25, "blue");
    grd.addColorStop(0.50, "green");
    grd.addColorStop(0.75, "yellow");
    grd.addColorStop(1.0, "red");
    ctx.fillStyle = grd;
    ctx.fillRect(0, 0, c.width, c.height);  // Rectangle fills canvas.
    ctx.font = "12px Arial";
    ctx.fillStyle = "white";
```

```
    ctx.fillText("100", 175, 15);
    ctx.fillText("0", 10, 15);
    legendImageInfo = ctx.getImageData(0, 0, c.width, 1);  // Save pixel info.
    legend = c.toDataURL();  // Save as png for custom pp.
  }

  function GetLegendColor(value, maxValue)
  {
    value = (value > maxValue) ? maxValue : value;
    var idx = Math.round((value / maxValue) * 200) * 4 - 4;
    if (idx < 0) {
      idx = 0;
    }
    return 'rgba(' + legendImageInfo.data[idx] + ',' +
      legendImageInfo.data[idx + 1] + ',' +
      legendImageInfo.data[idx + 2] + ',' + '0.2)';
  }

  function Button2_Click()
  {
    var colorVal = parseInt(textbox2.value);
    var colorRgba = GetLegendColor(colorVal, 100);

    Microsoft.Maps.loadModule('Microsoft.Maps.WellKnownText', function () {
      var geoColl = Microsoft.Maps.WellKnownText.read(colorado,
        { polygonOptions: { fillColor: colorRgba } });

      for (var i = 0; i < geoColl.length; ++i) {
        stateLayer.add(geoColl[i]);
      }
      map.layers.insert(stateLayer);
    });
  }

  </script>
</head>

<body style="background-color:lightgreen">
  <div id='controlPanel' style="float:left; width:262px; height:580px;
    border:1px solid green; padding:10px; background-color: beige">

    <input id="button1" type='button' style="width:125px;"
      value=' Decode Color ' onclick="Button1_Click();"></input>
    <div style="width:2px; display:inline-block"></div>
    <input id="textbox1" type='text' size='15' value=' 50.0 '></input><br/>
    <span style="display:block; height:10px"></span>

    <input id="button2" type='button' style="width:125px;"
      value='Color Colorado' onclick="Button2_Click();"></input>
    <div style="width:2px; display:inline-block"></div>
    <input id="textbox2" type='text' size='15' value=' 50.0 '></input><br/>
    <span style="display:block; height:10px"></span>

    <textarea id='msgArea' rows="36" cols="36" style="font-family:Consolas;
```

```
        font-size:12px"></textarea>
    </div>

    <div style="float:left; width:10px; height:600px"></div>
    <div id='mapDiv' style="float:left; width:700px; height:600px;
      border:1px solid red;"></div>
    <br style="clear: left;" />

   <script type='text/javascript'
     src='http://www.bing.com/api/maps/mapcontrol?callback=GetMap'
     async defer>
    </script>

  </body>
</html>
```

The demo application sets up seven global script-scope objects:

```
var map = null;
var ppLayer = null;
var legendLayer = null;
var stateLayer = null;
var legend = null;
var legendImageInfo = null;
var colorado =
  'GEOMETRYCOLLECTION( POLYGON ((-102 41, -109 41, -109 37,' +
  '-102 37, -102 41)) )';;
```

Object **ppLayer** is a map layer for holding ordinary pushpins. The color-gradient legend is actually a custom pushpin, and a **legendLayer** map layer object is created for it. The **stateLayer** object is for a polygon that outlines Colorado.

The **legend** object is actually a PNG image that will be created programmatically and act as the icon for a custom pushpin. The **legendImageInfo** is an object that holds information about the legend image and is used to map numeric values to color values.

The **colorado** object is a polygon shape defined using WKT syntax. Colorado is quite simple and has just five lat-lon vertices (the last vertex duplicates the first in order to close the shape).

The demo application loads the **map** object asynchronously by calling the program-defined **GetMap()** function in the usual way. The statements that create the color-gradient legend are:

```
legendLayer = new Microsoft.Maps.Layer();
CreateLegend();
var legendOpt = { icon: legend,
  anchor: new Microsoft.Maps.Point(-10,-10) };
```

The key code is contained in program-defined function **CreateLegend()**. The **options** object sets the anchor with an offset of 10 pixels so that the legend won't be flush with the map edge.

The statements that place the gradient legend on the map are:

```
var bounds = map.getBounds();
var nw = bounds.getNorthwest();
var legendPp = new Microsoft.Maps.Pushpin(nw, legendOpt);
legendLayer.add(legendPp);
map.layers.insert(legendLayer);
```

The legend is simply a pushpin in disguise, and it's placed on the map like a normal pushpin. The definition of **`CreateLegend()`** begins with:

```
function CreateLegend()
{
  var c = document.createElement('canvas');
  c.width = 200;
  c.height = 20;
  var ctx = c.getContext("2d");
. . .
```

We use an HTML5 canvas to draw a color gradient, then convert that canvas to a PNG image for use as a custom pushpin icon. As you'll see shortly, using an HTML5 canvas is preferable to using SVG or a static image. Next, a color gradient is created using the built-in **`createLinearGradient()`** function:

```
var grd = ctx.createLinearGradient(0, 0, c.width, 0);
grd.addColorStop(0.0, "purple");
grd.addColorStop(0.25, "blue");
grd.addColorStop(0.50, "green");
grd.addColorStop(0.75, "yellow");
grd.addColorStop(1.0, "red");
```

The arguments to **`createLinearGradient()`** can be tricky. In this case, they define a gradient that goes across the entire width of the canvas rather than from top to bottom. Next, the function fills a rectangle using the gradient information:

```
ctx.fillStyle = grd;
ctx.fillRect(0, 0, c.width, c.height);  // rectangle fills canvas
ctx.font = "12px Arial";
ctx.fillStyle = "white";
ctx.fillText("100", 175, 15);
ctx.fillText("0", 10, 15);
```

The code places labels “0” and “100” at the left and right ends of the gradient. The positioning argument values (175, 15, 10, 15) were determined using a bit of trial and error. The function definition concludes with:

```
. . .
  legendImageInfo = ctx.getImageData(0, 0, c.width, 1);
  legend = c.toDataURL();
}
```

The **getImageData()** function returns an array of RGBA data from the gradient. These values will be used by a program-defined function that maps a number value from 0 to 100 to an RGBA color. The oddly named **toDataURL()** function returns a PNG image for display purposes.

The first button in the control panel area is for informational purposes and displays the RGBA information defined by the color gradient for a specified value. The code is:

```
function Button1_Click()
{
  var maxValue = 100;
  var v = parseInt(textbox1.value);
  var color = GetLegendColor(v, maxValue);
  WriteLn("value = " + v);
  WriteLn("color = " + color.toString());
  WriteLn("");
}
```

All the real work is performed by the **GetLegendColor()** function, which pulls information from the script-global **legendImageInfo** object and is defined as:

```
function GetLegendColor(value, maxValue)
{
  value = (value > maxValue) ? maxValue : value;
  var idx = Math.round((value / maxValue) * 200) * 4 - 4;
  if (idx < 0) {
    idx = 0;
  }
  return 'rgba(' + legendImageInfo.data[idx] + ',' +
    legendImageInfo.data[idx + 1] + ',' +
    legendImageInfo.data[idx + 2] + ',' + '0.2)';
}
```

The index value takes into account that each pixel has four values. The alpha value is hard-coded to 0.2, but you might want to parameterize it. The color value is determined by the **legendImageInfo** information that was created from the HTML5 canvas object. Without this information, mapping a numeric value to a color from a gradient would be quite difficult.

The second button control colors the polygon that outlines Colorado. The code is:

```
function Button2_Click()
{
  var colorVal = parseInt(textbox2.value);
  var colorRgba = GetLegendColor(colorVal, 100);

  Microsoft.Maps.loadModule('Microsoft.Maps.WellKnownText', function () {
    var geoColl = Microsoft.Maps.WellKnownText.read(colorado,
      { polygonOptions: { fillColor: colorRgba } });
```

```
    for (var i = 0; i < geoColl.length; ++i) {
      stateLayer.add(geoColl[i]);
    }

    map.layers.insert(stateLayer);
  });
}
```

The **read()** function of the WKT module parses the WKT data stored in the **colorado** object into an array of polygons because many shapes (such as Hawaii) are made up of several polygons. The color of the polygon is set to the RGBA value that corresponds to a numeric value from 0 to 100. Instead of storing and reading data as WKT, using the GeoJSON format offers an important alternative. For example:

```
var wyoming = { 'type': 'FeatureCollection',
  'features': [{ 'type': 'Feature', 'id': 'WY',
  'properties': { 'name': 'Wyoming' },
  'geometry': { 'type': 'Polygon', 'coordinates':
  [[[-104.05,41],[-104.05,45],[-111.05,45],[-111.05,41],[-104.05,41]]]
} }] };
```

And then:

```
Microsoft.Maps.loadModule('Microsoft.Maps.GeoJson', function () {
  var geoColl = Microsoft.Maps.GeoJson.read(wyoming,
    { polygonOptions: { fillColor: colorRgba } });
  for (var i = 0; i < geoColl.length; ++i) {
    stateLayer.add(geoColl[i]);
  }
  map.layers.insert(stateLayer);
});
```

In summary, one way to create a gradient legend is to draw an HTML5 canvas, convert it to a PNG image, and use the image as the icon for a custom pushpin. This technique allows to you write a function that maps a numeric value to a color in the gradient. The Bing Maps V8 library can work with WKT- and GeoJSON-formatted data.

Resources

For detailed information about the Bing Maps V8 Well Known Text module, see: https://msdn.microsoft.com/en-us/library/mt712880.aspx.

For detailed information about the Bing Maps V8 GeoJSON module, see: https://msdn.microsoft.com/en-us/library/mt712806.aspx.

4.4 Custom Infobox objects

The Bing Maps library allows you to create custom-styled **`Infobox`** objects using ordinary CSS markup combined with the JavaScript string **`replace()`** function. During development, it's often useful to generate random **`Location`** objects. The Maps.TestGenerator module has a **`getLocations()`** function. However, the locations generated are not reproducible. In order to generate reproducible random locations, you can write your own random-number generator.

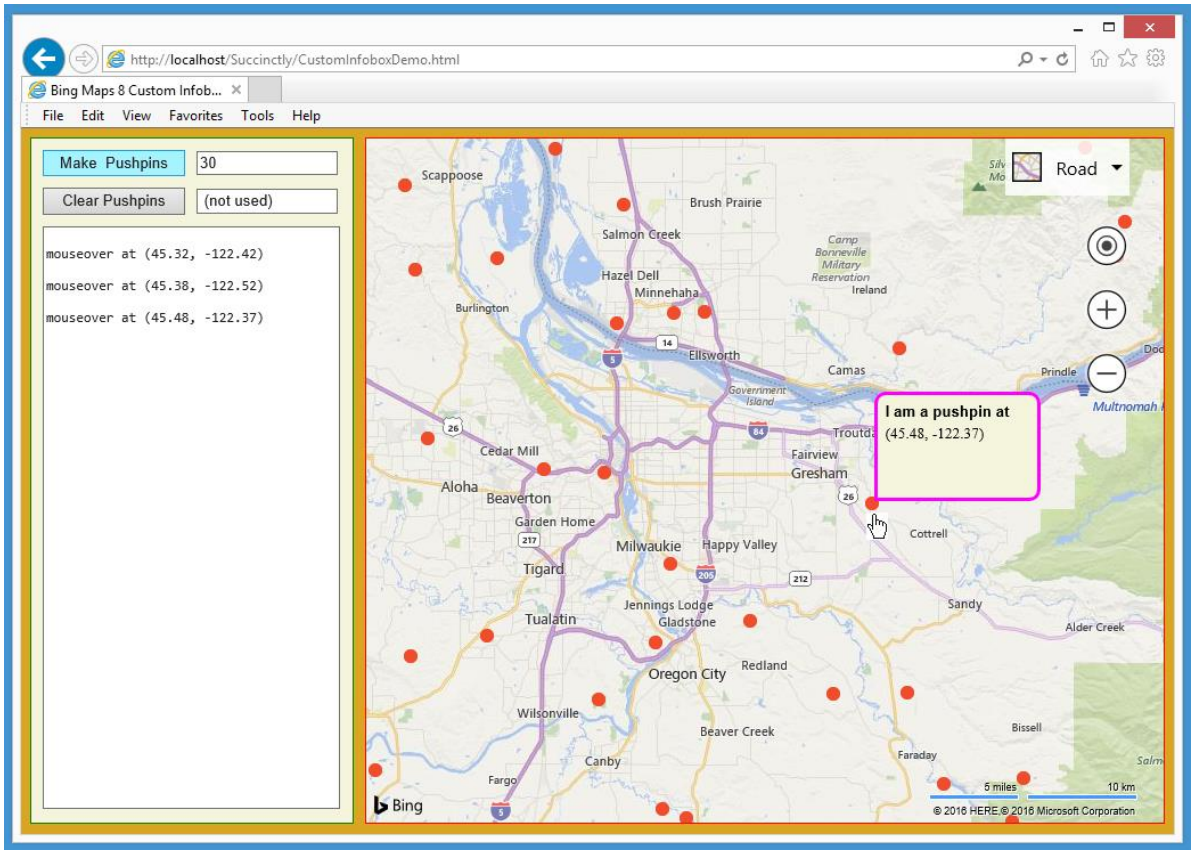

Figure 28: Custom Infobox with Reproducible Random Locations Demo

The demo web application shown in Figure 28 illustrates custom **`Infobox`** objects and custom pseudorandom-location generation. The demo initially loads a map centered at (45.50, -122.50), which is near Portland, Ore. When the user clicked "Make Pushpins," the application used a custom random-number generator to create 30 random **`Location`** objects that are within the map boundaries.

The locations are then used to create 30 orange pushpins. Each pushpin's mouseover event handler triggers the appearance of a custom-styled Infobox that displays the pushpin's location.

The random locations are reproducible, meaning that if the pushpins are cleared away or the webpage is reloaded, clicking "Make Pushpins" will recreate the same 30 locations.

The demo application is named CustomInfoboxDemo.html and is defined in a single file.

Code Listing 12: CustomInfoboxDemo.html

```
<!DOCTYPE html>
<!-- CustomInfoboxDemo.html -->

<html>
  <head>
    <title>Bing Maps 8 Custom Infobox Demo</title>
    <meta http-equiv='Content-Type' content='text/html; charset=utf-8'/>

    <script type='text/javascript'>

    var map = null;
    var ctr = null;
    var pushpins = [];
    var ppLayer = null;
    var infobox = null;
    var ibTemplate = null;

    function GetMap()
    {
      var options = {
        credentials: "Anw _ _ _ 3xt",
        center: new Microsoft.Maps.Location(45.50, -122.50),  // Portland, OR
        mapTypeId: Microsoft.Maps.MapTypeId.road,
        zoom: 10,
        enableClickableLogo: false,
        showTermsLink: false
      };

      var mapDiv = document.getElementById("mapDiv");
      map = new Microsoft.Maps.Map(mapDiv, options);
      ctr = map.getCenter();

      ppLayer = new Microsoft.Maps.Layer();

      ibTemplate = '<div id="ibBox" style="border-radius: 10px 10px 10px 0px;' +
        ' background-color:beige; border-style:solid; border-width:medium;' +
        ' border-color:magenta; min-height: 90px; width: 140px; ">' +
        ' <b id="ibTitle" style="position: absolute; top: 10px; left: 10px;' +
        ' width: 220px; font-family:Arial; font-size:small">{title}</b>' +
        ' <a id="ibDesc" style="position: absolute; top: 30px; left: 10px;' +
        ' width: 220px; font-size:small">{description}</a></div>';

      infobox = new Microsoft.Maps.Infobox(ctr, {
        visible: false,
        htmlContent: ibTemplate.replace('{title}',
          'dummy title').replace('{description}', 'dummy description')
      });

      infobox.setMap(map);
    }
```

```
function ShowInfobox(e)
{
  var loc = e.target.getLocation();
  infobox.setLocation(loc);
  WriteLn('\nmouseover at ' + LatLonStr(loc));
  infobox.setOptions({
    visible: true,
    offset: new Microsoft.Maps.Point(2, 2),
    htmlContent: ibTemplate.replace('{title}',
      'I am a ' + e.targetType + ' at').replace('{description}',
      LatLonStr(loc)) });
}

function HideInfobox(e)
{
  infobox.setOptions({ visible: false });
}

function Button1_Click()
{
  var count = parseInt(textbox1.value);
  var locs = MakeLocs(count, map.getBounds(), 19);

  var orangeDot = '' +
    'AAAAMCAYAAABWdVznAAAAGXRFWHRTb2Z0d2FyZQBBZG9iZSBJbWFnZVJlYWR5cc' +
    'llPAAAAIxJREFUeNpiZICCD766CkCqHogDgFgAJgzEG4C4UWDz5QcgAUaoYgMgt' +
    'R9JIToAaXQEarrACDX5PB7FyJoMmaDOIKSYAaqmngnqZmJBABORpsNtYWIgETBB' +
    'PUMs+MAEDWdiwQaQhkYibQGpaWSCxqAjAU2wiHsA9jQoBkGRAsQL0DR+gIoZQtU' +
    'wAAQYAGgpKDzqLFoIAAAAAElFTkSuQmCC';

  var ppOptions = { icon: orangeDot,
    anchor: new Microsoft.Maps.Point(6, 6) };

  for (var i = 0; i < count; ++i) {
    var pp = new Microsoft.Maps.Pushpin(locs[i], ppOptions);
    Microsoft.Maps.Events.addHandler(pp, 'mouseover', ShowInfobox);
    Microsoft.Maps.Events.addHandler(pp, 'mouseout', HideInfobox);
    pushpins[i] = pp;
  }

  ppLayer.add(pushpins);
  map.layers.insert(ppLayer);
}

function Button2_Click()
{
  ppLayer.clear();
}

function MakeLocs(n, bounds, sd)
{
  Math.myRandom = SetSeed(sd);
  var latHi = bounds.getNorth();
```

```
    var latLo = bounds.getSouth();
    var lonHi = bounds.getEast();
    var lonLo = bounds.getWest();

    var result = [];
    for (var i = 0; i < n; ++i) {
      var lat = (latHi - latLo) * Math.myRandom() + latLo;
      var lon = (lonHi - lonLo) * Math.myRandom() + lonLo;
      var loc = new Microsoft.Maps.Location(lat, lon);
      result.push(loc);
    }
    return result;
  }

  SetSeed = function(seed) {
    if (seed <= 0) {
      alert('Bad seed');
    }

    function ParkMiller() {
      var hi = Math.floor(seed / 127773);
      var lo = seed % 127773;
      seed = (16807 * lo) - (2836 * hi); // global
      if (seed <= 0) {
        seed += 2147483647;
      }
      return seed / 2147483647;
    }

    for (var i = 0; i < 20; ++i) {  // Burn away first 20 results.
      var dummy = ParkMiller();
    }
    return ParkMiller;
  }


  function WriteLn(txt)
  {
    var existing = msgArea.value;
    msgArea.value = existing + txt + "\n";
  }

  function LatLonStr(loc)
  {
    var s = "(" + Number(loc.latitude).toFixed(2) + ", " +
      Number(loc.longitude).toFixed(2) + ")";
    return s;
  }

  </script>
</head>

<body style="background-color:goldenrod">
  <div id='controlPanel' style="float:left; width:262px; height:580px;
```

```
      border:1px solid green; padding:10px; background-color: beige">

      <input id="button1" type='button' style="width:125px;"
        value=' Make  Pushpins ' onclick="Button1_Click();"></input>
      <div style="width:2px; display:inline-block"></div>
      <input id="textbox1" type='text' size='15'
        value='30'></input><br/>
      <span style="display:block; height:10px"></span>

      <input id="button2" type='button' style="width:125px;"
        value=' Clear Pushpins ' onclick="Button2_Click();"></input>
      <div style="width:2px; display:inline-block"></div>
      <input id="textbox2" type='text' size='15'
        value=' (not used) '></input><br/>
      <span style="display:block; height:10px"></span>

      <textarea id='msgArea' rows="36" cols="36" style="font-family:Consolas;
        font-size:12px"></textarea>
    </div>

    <div style="float:left; width:10px; height:600px"></div>
    <div id='mapDiv' style="float:left; width:700px; height:600px;
      border:1px solid red;"></div>
    <br style="clear: left;" />

    <script type='text/javascript'
      src='http://www.bing.com/api/maps/mapcontrol?callback=GetMap'
      async defer>
    </script>

  </body>
</html>
```

The demo sets up six script-global objects:

```
var map = null;
var ctr = null;
var pushpins = [];
var ppLayer = null;
var infobox = null;
var ibTemplate = null;
```

The **Infobox** object is a single Infobox that will be shared by all pushpins stored in the array named **pushpins**. The **ibTemplate** object is a string that will define the appearance of the **Infobox** object. Object **ctr** will hold the center latitude and longitude of the map.

The map is created using function **GetMap()**. The function also instantiates the custom template:

```
ibTemplate = '<div id="ibBox" style="border-radius: 10px 10px 10px 0px;' +
  ' background-color:beige; border-style:solid; border-width:medium;' +
  ' border-color:magenta; min-height: 90px; width: 140px; ">' +
  ' <b id="ibTitle" style="position: absolute; top: 10px; left: 10px;' +
  ' width: 220px; font-family:Arial; font-size:small">{title}</b>' +
  ' <a id="ibDesc" style="position: absolute; top: 30px; left: 10px;' +
  ' width: 220px; font-size:small">{description}</a></div>';
```

The template has three main areas. The first area defines the appearance of the box shape. That means the template is defined using ordinary CSS markup and that there will be a huge number of formatting options available to you.

The second area defines the appearance of the title of the Infobox. Notice the “{title}”—this will be replaced by specific content.

The third area defines the appearance of the description part of the Infobox, where “{description}” will be replaced by specific content. The “{title}” and “{description}” are not special words, so you could use “{foo}” and “{bar}” if you wish.

The global Infobox is created this way:

```
infobox = new Microsoft.Maps.Infobox(ctr, {
  visible: false,
  htmlContent: ibTemplate.replace('{title}',
    'dummy title').replace('{description}', 'dummy description')
});
```

The Infobox is initially positioned at map center but is invisible. The JavaScript **`replace()`** function is used to store dummy text for the title and description. The replace-chaining is convenient and standard practice, but, in my opinion, it’s a bit ugly. An alternative is:

```
var s = ibTemplate.replace('{title}', 'dummy title');
var t = s.replace('{description}', 'dummy description');
. . .
htmlContent: t
```

The built-in JavaScript **`Math.random()`** function cannot accept an initial seed value, which means the results are not reproducible. The demo defines a custom random-number generator that uses the Lehmer algorithm (which, you’ll recall, is sometimes called the Park-Miller algorithm). The code for the custom random-number generator is:

```
SetSeed = function(seed) {
  if (seed <= 0) {
    alert('Bad seed');
  }
```

```
  function ParkMiller() {
    var hi = Math.floor(seed / 127773);
    var lo = seed % 127773;
    seed = (16807 * lo) - (2836 * hi); // global
    if (seed <= 0) {
      seed += 2147483647;
    }

    return seed / 2147483647;
  }

  for (var i = 0; i < 20; ++i) {
    var dummy = ParkMiller();
  }

  return ParkMiller;
}
```

Notice that the function is a declaration (rather than a definition) with a nested function definition that is returned. The code is rather subtle, and you can find a link to more information about the algorithm at the end of this section.

The random-number generator is called by a program-defined function **MakeLocations()** that is defined:

```
function MakeLocs(n, bounds, sd)
{
  Math.myRandom = SetSeed(sd);
  var latHi = bounds.getNorth();
  var latLo = bounds.getSouth();
  var lonHi = bounds.getEast();
  var lonLo = bounds.getWest();
. . .
```

The function accepts the number of **Location** objects to generate as **n**, a Maps Bounds object as **bounds**, and a random-number seed value as **sd**. The random-number generator is created as **Math.myRandom** by using the interesting JavaScript technique of assigning a new member to an existing class on the fly.

The built-in **getNorth()** and **getSouth()** functions return the top and bottom latitude values of the associated **bounds** object. And the **getEast()** and **getWest()** functions return the right and left longitude values. To make 100 random locations within the current map area, using a random seed value of 19, the function could be called this way:

```
var locs = MakeLocs(100, map.getBounds(), 19);
```

The code for **MakeLocations()** concludes with:

```
  . . .
  var result = [];
  for (var i = 0; i < n; ++i) {
    var lat = (latHi - latLo) * Math.myRandom() + latLo;
    var lon = (lonHi - lonLo) * Math.myRandom() + lonLo;
    var loc = new Microsoft.Maps.Location(lat, lon);
    result.push(loc);
  }
  return result;
}
```

The **Math.myRansom()** function returns a pseudorandom value between 0.0 and 1.0. The code idiom to return a value between arbitrary values A and B is **(A-B) * Math.myRandom() + B**.

The **Button1_Click()** function creates random-location pushpins and associates the mouseover and mouseout events with functions **ShowInfobox()** and **HideInfobox()**:

```
function ShowInfobox(e)
{
  var loc = e.target.getLocation();
  infobox.setLocation(loc);
  infobox.setOptions({
    visible: true, offset: new Microsoft.Maps.Point(2, 2),
    htmlContent: ibTemplate.replace('{title}',
      'I am a ' + e.targetType + ' at').replace('{description}',
      LatLonStr(loc)) });
}
```

With custom **Infobox** objects, the **showCloseButton** option property is ignored, which is disadvantageous. Because **Infobox** objects have no click handlers, it's hard to get rid of a visible Infobox and, in many situations, using the mouseout event to set the **visible** property to **false** is a good method. For example:

```
Microsoft.Maps.Events.addHandler(pp, 'mouseout', HideInfobox);
. . .
function HideInfobox(e)
{
  infobox.setOptions({ visible: false });
}
```

In summary, in order to create custom-styled **Infobox** objects, you define a template CSS string and use the JavaScript string-replace function to provide specific title and description content. You can create reproducible pseudorandom **Location** objects by writing a custom function that uses the Lehmer (or Park-Miller) algorithm.

Resources

For detailed information about the InfoboxOptions class, see:
https://msdn.microsoft.com/en-us/library/mt712658.aspx.

For advanced examples of custom **`Infobox`** objects, see:
https://msdn.microsoft.com/en-us/library/mt750644.aspx.

For additional information about the Lehmer random-number algorithm, see:
https://en.wikipedia.org/wiki/Lehmer_random_number_generator.

For a famous article about random-number generators, search the Internet for:
“Random Number Generators: Good Ones are Hard to Find,” 1988, Park and Miller.