RealityMobile

RealityVision - Secure Maps
Integration Guide

With the release of RealityVision 3.4 the ability to use any WMTS compliant mapping service became possible. Additionally, the map tiles are now served to the client devices through a secured proxy. This has many benefits, but the biggest change that can affect the user experience is the ability for organizations to add their own assets and information which have associated locations. Examples of such information can be, but are not limited to, a mobile base of operations and regions on the map that provide additional context for users either for a short time or on a permanent basis.

To make the integration of your unique data into the RealityVision system as smooth as possible, this document provides a combination of necessary and useful information on how to display your data. The following topics will be discussed:

- The underlying architecture of the Secure Maps implementation
- Creation of custom mapping layers
- Adding basic objects to a layer
- How to periodically consume basic data and update the map
- Demonstrate the addition of custom pop-ups for mapping layer objects
- Additional resources

To support shallow integration efforts by programmers who may have only limited JavaScript experience, this document sometimes takes a moment to describe relatively basic JavaScript concepts. For readers with more substantial experience in the language it is suggested to scan through the example scenarios and then reference the resources section at the bottom of this document. If you have prior OpenLayers experience, at least read through the last example on Pop-ups since there are some specific coding patterns required to have them work alongside the RealityVision pop-up implementation. The examples discussed below have been provided in a ZIP file for convenience, along with this document.

Secure Maps Architecture

The RealityVision Secure Maps implementation was built on top of OpenLayers (a JavaScript mapping API) to provide a tested and documented foundation for access to mapping services. Each client accesses the RealityVision Secure Map JavaScript when it requests a new map display. Upon instantiation of the Secure Map object a Customer Layer initialization routine is called. Within this function access to the OpenLayers map object is accessible. With it you can also access the Secure Maps API when needed. Because the RealityVision Secure Maps API has handled the creation of the map object, tile server and basic navigation controls, integration of your data is generally limited to the creation of custom layers and deciding how to display your data on them.

The JavaScript you will create in the following examples will execute on your client devices. Therefore, consideration of memory constraints, power consumption and client functionality (touch vs. mouse for example) can be significant considerations in your design. It also means that you will want to test your changes on all platforms that your organization runs RealityVision on.

Creating a Custom Layer

So, what exactly is a layer? When rendering data on the map it is useful to put data of different types into their own containers. This allows the end user to easily toggle on or off the display of a given type of data. Without modification, RealityVision provides layers for the various types of video a user might consume and the location of users. Each category of item on the map can be shown or hidden based upon the user's needs or interests at a given moment. Each layer displays on top of the layers below it as shown in Figure 2 below:

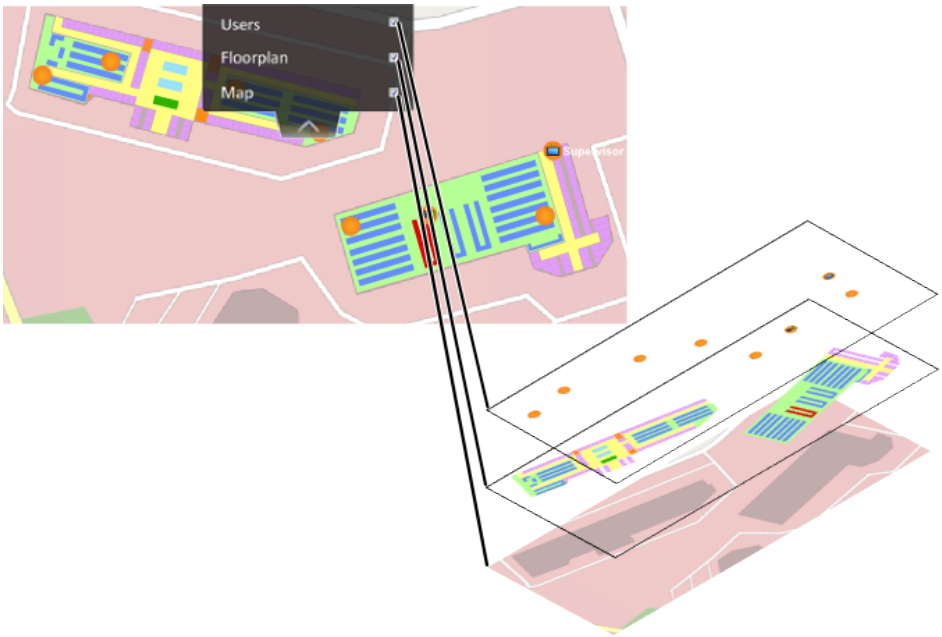

Figure 2: Layer Visualization

To begin building a new customer layer requires modifying the empty CustomerLayer.js file that is placed on the server when RealityVision is installed. With a typical installation this file can be found on the RealityVision host server at **C:\inetpub\rvroot\MTPS\JavaScript\CustomerLayer.js**. It will contain a single empty JavaScript function that reads:

```
SecureMaps.InitializeCustomerLayers = function() {
    // Insert custom layer OpenLayers code here. Refer to API
    documentation for more details.
};
```

To create a layer you will need to access OpenLayers and allocate a new layer object, for example:

```
var myLayer = new OpenLayer.Layers.Vector("My Layer");
```

A vector layer allows for the creation of polygon regions, lines or map markers. It is a useful layer to create when you wish to have direct control of the content rendered on the map. To make this layer visible you need to tell OpenLayers about it. To do so call:

```
map.addLayer(myLayer);
```

This places myLayer at the very top of the displayed layers. To have your new layer appear below the RealityVision layers, but above the map tiles, call:

```
map.setLayerIndex(myLayer,1);
```

The setLayerIndex call places the layer at the specified layer index, in this case just above the map tiles ,which are on layer 0, but below all RealityVision layers, which start on layer 1 (when no Satellite layer is available, see below). The layer that was layer 1 becomes layer 2, layer 2 becomes layer 3, and so on. When deciding where to put your layers you can reference Chart 1 below, which lists the default layers added by RealityVision on startup. Higher numbered layers display on top of lower numbered layers, and, as mentioned above, any new layer added is always placed at the top until a new location is specified in the setLayerIndex call.

Default Layer Index			
Bing Maps	**All Other Maps**	**Layer Name**	**Shown in Layer Switcher**
13	12	Users	Yes
12	11	Favorites	Yes
11	10	Cameras	Yes
10	9	Screencasts	Yes
9	8	Video Files	Yes
8	7	My Location	Yes
7	6	Users – Labels	No
6	5	Favorites – Labels	No
5	4	Cameras – Labels	No
4	3	Screencasts - Labels	No
3	2	Video Files – Labels	No
2	1	My Location – Labels	No
1		Satellite	Yes
0	0	Streets	No

Chart 1: Default Layer Indexes

Although the layers used in this document are all Vector layers, additional layer types are available with OpenLayers. See the Resources section at the bottom of this document for how to find the OpenLayers documentation and learn what other layer types are available.

Creating a rectangular region on the map

You are now ready to add content to the map. This initial example shows how to drop a small rectangular region around Times Square in New York. The end result will look something like this:

Figure 3: Bounding Region Drawn Around Times Square

The lower left corner will be at Latitude and Longitude of 40.758546,-73.985359, and the upper right corner at 40.759373,-73.984555. To create the boundary object call:

```
var timesSquareBoundary = new OpenLayers.Bounds(
    -73.985359, 40.758546,
    -73.984555, 40.759373);
```

Note that Bounds accepts the coordinates in Longitude, Latitude order, so when calling Bounds they were reversed. The first pair was the lower left coordinate; the second was the upper right.

The Bounds object created by the code above uses coordinates in what is commonly called the WGS 84 (EPSG:4326) system. When integrating with OpenLayers, if you do not know what projection system your coordinates are in, it is likely they are in WGS 84. OpenLayers uses a different coordinate projection system for its display, so conversion into its address space is typically necessary. To perform this coordinate translation, the first step is to create a WGS84 projection object. OpenLayers uses the alternate naming for this: “EPSG:4326”. The JavaScript looks like this:

```
var projWGS84 = new OpenLayers.Projection("EPSG:4326");
```

To convert into the maps coordinate system, tell the boundary object to transform its points by providing it our projection (the coordinate space of our latitude, longitude coordinates) and that of the map, which OpenLayers provides a utility method for. To perform this action call:

```
timesSquareBoundary.transform( projWGS84,
    map.getProjectionObject() );
```

Many OpenLayers objects have a transform method similar to the above. If the object you are using does not, see if there is another object type that is used to construct the object, and if it has a transform

function instead. Although individual points can be converted (See OpenLayers LonLat object) it is not typically the most efficient means of adjusting projections.

With the boundary now in the map's coordinate space you can now create an OpenLayers polygon from the bounds object like this:

```
var timesSquareRectangle = timesSquareBoundary.toGeometry();
```

This polygon can be used to create a Feature on the map. To do that you need to decide how you want the rectangle to appear. This is done by declaring what JavaScript calls an anonymous object to use as a style. The values that are allowed are defined in the OpenLayers documentation. If you do not specify a style attribute, its default value is used. The attributes used for this example are set like this:

```
var rectStyle = {
        fillColor: '#00ff00'     // Fill with green
        fillOpacity: 0.2,        // 20% opacity for the fill
        strokeColor: '#00ff00',  // A green border
        strokeWidth: 2,          // Set border to 2 pixels wide
        strokeOpacity: 0.6,      // 60% border opacity
    };
```

Colors in JavaScript are in RGB (red, green blue) format. Many tools are available to create "hex" colors in this format and one such example online is the HTML Color Picker listed in the resources section of this document.

The style you created above gives you the last piece you need to create a feature and add it to the map. To add it to the map you will make these two calls:

```
var timesSquareFeature = new OpenLayers.Feature.Vector(
    timesSquareRectangle, {name:"Times Square"}, rectStyle);
myLayer.addFeatures([timesSquareFeature]);
```

The anonymous object that specifies the name "Times Square" for the feature allows you to identify it in the future if you need to remove or modify it. It works as the object's ID. You should also note the brackets "[ ]" around the feature when addFeatures is called. The addFeatures function, by its pluralized name, is designed to add multiple features at the same time, with each feature object's name separated by a comma. If there is only one feature to add, as in this case, then you still need to create an array, but with just the single item. The completed InitializeCustomerLayer function will look like this:

```
SecureMaps.InitializeCustomerLayers = function () {

    var myLayer = new OpenLayers.Layer.Vector("My Layer");
    map.addLayer(myLayer);
    map.setLayerIndex(myLayer,1);
```

```
        var timesSquareBoundary = new OpenLayers.Bounds(
            -73.985359, 40.758546, -73.984555, 40.759373);

        var projWGS84 = new OpenLayers.Projection("EPSG:4326");
        timesSquareBoundary.transform( projWGS84,
            map.getProjectionObject() );

        var timesSquareRectangle = timesSquareBoundary.toGeometry();

        var rectStyle = {
            fillOpacity: 0.2,
            strokeColor: '#00ff00',
            strokeWidth: 2,
            strokeOpacity: 0.6,
            fillColor: '#00ff00'
        };
        var timesSquareFeature = new OpenLayers.Feature.Vector(
            timesSquareRectangle,{name:"Times Square"}, rectStyle);
        myLayer.addFeatures([timesSquareFeature]);
    };
```

If you save the file, over-writing the original CustomerLayer.js file, and launch the Management Console, you should be able to quickly find the green rectangle hovering over Times Square. If you pull down the Layer Switcher, you will also find your new layer in the list and can toggle the rectangle on and off.

Creating Polygons

Geographic features and regions of interest are often not rectangular, or if they are they do not line up in a perfectly north-south or east-west direction. OpenLayers provides a number of geometry objects in its OpenLayers.Geometry namespace that can be used to create more complex map features. In this example the LinearRing and Polygon objects will be demonstrated. The following regions will be created in this example to demonstrate this:

Figure 4: Multiple Polygons Example

As before, you will create an IntializeCustomerLayer function. This time, however, to provide an example of how you can create more than one object using the same code, this example builds a pair of reusable functions to avoid code duplication.

The first of these two functions will also double as a container for a shape. The function will be called CustomerShape and accept a name, color and a set of points for the polygon. Here is the completed function which is discussed below:

```
// A utility method that creates a new customer shape object
// with a given name, path and color
CustomerShape = function(name, path, color) {
    // Create an array that will dynamically grow as
    // points are added
    var listOfPoints = [];
    var pointIndex = 0;
    for (var index=0; index < path.length; index += 2) {
        // Create a new Point object and add it to the array
        listOfPoints[pointIndex++] = new
            OpenLayers.Geometry.Point(path[index+1],path[index]);
    }

    // Create a linear ring (first point closes with last
```

```
        // point in list)
        var ringOfPoints = new OpenLayers.Geometry.LinearRing(
            listOfPoints);

        // Transform into map coordinates
        var projWGS84 = new OpenLayers.Projection("EPSG:4326");
        ringOfPoints.transform(
            projWGS84, map.getProjectionObject() );

        // Create object members that can be accessed later


        // Treat the name as the object's ID
        this.id = name;

        // Save a newly created polygon with the list of points.
        this.polygon = new
            OpenLayers.Geometry.Polygon([ringOfPoints]);

        // Set the object's style
        this.style = {
            fillOpacity: 0.2,
            strokeColor: color,
            strokeWidth: 2,
            strokeOpacity: 0.6,
            fillColor: color
        };
    };
```

As stated above, CustomerShape is declared as a function which takes three values: name, path and color. JavaScript handles figuring out what the types of these are when they are passed to the method. The function begins with the opening brace '{', and ends with the closing brace '}'.

The function creates a polygon from the given points, draws a slightly translucent border around them, and fills the region inside with a more translucent version of the same color. This is created with a pair of OpenLayers objects called a LinearRing and a Polygon. This is just one example of the types of shapes that you can create. Alternative versions of this method could be created to render lines, curves or multiple related polygons from a single method call. See the OpenLayers link at the end of this document for information on additional geometry options.

To build the polygon the first thing this function does is create two variables using the "var" keyword that will store the list of points and an index counter into the list. This is necessary since it is intended that the path be a simple list of latitude, longitude pairs that are not separated as individual objects. The

code that follows will read two array items at a time and create single points from them that are placed into this list.

The for loop performs this conversion by walking the input array of Latitude, Longitude pairs, reading two items per pass within the loop, and converting them into OpenLayers Point objects that are placed into the point array created above. This is a necessary operation because OpenLayers' LinearRing object only accepts arrays of points when created. As with functions, the start and end of loops are contained within braces similar to other languages like C, Java or C#. When adding the items to the array, pointIndex++ is used, which automatically bumps the value of the index immediately after its value is read. The value of pointIndex then specifies the index that will be used on the next pass through the loop, but the value used to access the array was the value at the time the call was made before it was incremented. The result is that the two input values are read (latitude,longitude) from the input list, swapped (this is why the order of "path[index+1], path[index]" is given to the Point) and then the new point is added to the end of the array.

Once the loop is finished, you can see where the code creates the LinearRing:

```
var ringOfPoints = new OpenLayers.Geometry.LinearRing(
    listOfPoints);
```

This object ensures that an additional line is drawn from the last point in the array back to the first point in the array. Similar to the first example, the two lines of code that follow perform a translation from the WGS 84 projection to the map's projection model:

```
// Transform into map coordinates
var projWGS84 = new OpenLayers.Projection("EPSG:4326");
ringOfPoints.transform(
    projWGS84, map.getProjectionObject() );
```

At the end of the function are three "member" variables. The use of the "this" keyword tells JavaScript to store what is assigned to these variables as part of the object created when this function is accessed with the "new" keyword. Using the "new" keyword turns any function in JavaScript into a constructor which creates a new object instance. In this case a new CustomerShape instance would be created and it will store the ID for the object, which can be used to uniquely identify it, the polygon that is created from the LinearRing, and a style for the polygon.

The polygon's style requires additional discussion. By saving the style in the object, each polygon created with this function can then have its own style; in this example you will only apply different colors to the polygons. To see how the color is set, take note of how this code assigns the input parameter "color", to the strokeColor and fillColor attributes, instead of directly assigning the '#00FF00' style number string, as was done in the boundary rectangle example. This is what allows the caller of this routine to customize the style for each CustomerShape object created. To add additional customization options, you can extend this function by adding additional parameters and setting style

appropriate style properties with them. See the OpenLayers documentation reference, in the Resources section at the bottom of the document, for a list of additional attributes you can set. How far you wish to extend this routine will be based upon the level of flexibility you require to best display your data on the map layer.

Before you use CustomerShape to create your polygon you need to first define its path. As discussed earlier, CustomerShape expects an array of location values in latitude and longitude order, repeated as pairs. So create a new array called timesSquareAreaPath like this:

```
var timesSquareAreaPath = [
        // lat   , lon
        40.757782,-73.985684,
        40.757908,-73.985684,
        40.757997,-73.985627,
        40.758704,-73.985357,
        40.759424,-73.985126,
        40.760131,-73.984868,
        40.760799,-73.98452,
        40.760476,-73.983761,
        40.757957,-73.985517
    ];
```

With that path you are ready to create an object for placement on a layer of your choosing, allocate a new instance of CustomerShape using the JavaScript “new” keyword:

```
var timesSquareRegion = new CustomerShape("Times Square",
    timesSquareAreaPath, '#00ff00');
```

As mentioned briefly before, this creates a new object which stores the polygon and color ready to be rendered to the map once it is added to a layer.

To add it to the map this example shows how to create a second method; this one will be a part of the CustomerShape object using another JavaScript concept called a “prototype”. Using prototype when defining a function gives you access to the data stored in the object created when the “new” keyword was used earlier. To create the prototype function that will add a shape to a specified layer as a new feature add this code to your CustomerLayer.js file after the CustomerShape function:

```
// Creates a new feature object and puts it into
// the specified map layer
CustomerShape.prototype.AddToLayer = function(layer) {
    var feature = new OpenLayers.Feature.Vector(
        this.polygon, {name: this.id}, this.style);
    layer.addFeatures([feature]);
};
```

This creates a new routine called AddToLayer that takes a layer as its input and is only legal to call on a CustomerLayer object allocated by using new. The routine is fairly simple, but does a few key things. By have used the “this” keyword to create the CustomerShape this function has access to the three values we stored earlier: this.polygon, this.id and this.style. When a new OpenLayers.Feature.Vector is created, these members are used instead of the fixed values as in the boundary rectangle example. The call to addFeatures then adds the new feature to the layer. Just remember that addFeatures requires an array, so even if only a single object is given, as in this case, you still need to surround it with ‘[‘ and ‘]’.

All of the pieces are now in place to create two objects in the InitializeCustomerLayer function:

```
SecureMaps.InitializeCustomerLayers = function () {
    // Create our demo layer
    var myLayer = new OpenLayers.Layer.Vector("My Layer");
    map.addLayer(myLayer);
    map.setLayerIndex(myLayer,1);

    // Create a green wedge shape around Times Square, NY
    // Plus one block to the south and 3 more to the north
    var timesSquareAreaPath = [
        // lat   , lon
        40.757782,-73.985684,
        40.757908,-73.985684,
        40.757997,-73.985627,
        40.758704,-73.985357,
        40.759424,-73.985126,
        40.760131,-73.984868,
        40.760799,-73.98452,
        40.760476,-73.983761,
        40.757957,-73.985517
    ];
    var timesSquareRegion = new CustomerShape(
        "Times Square", timesSquareAreaPath, '#00ff00');
    timesSquareRegion.AddToLayer(myLayer);

    // Creates a blue rectangular shape, but not one that is
    // exactly north-south oriented, around Bryant Park, NY
    var bryantParkPath = [
        // lat   , lon
        40.753556,-73.985043,
        40.754864,-73.984124,
        40.753483,-73.980862,
        40.752199,-73.981828
    ];
    var bryantParkRegion = new CustomerShape(
```

```
            "Bryant Park", bryantParkPath, "#0000ff");
        bryantParkRegion.AddToLayer(myLayer);
};
```

As in the first example, a layer is created first. Second, this code creates a path to surround a series of blocks near Times Square, but this time, in a wedge shaped block along Broadway and 7th Ave. Then it allocates a CustomerShape and gives it the name “Times Square” and the color Green. Finally, the new object is added to the layer.

To complete the demonstration, that more than one object can be created using these routines without conflict, a second object is created surrounding Bryant Park. This is a rectangle, but as the code comment states, it is not in a fixed north-south aspect, it is instead aligned with the slanted roads. To have it stand out from the Times Square region, the color blue has been assigned to the object. Once the file has been saved to the server, the next time a client is launched the new JavaScript will be received and render the two regions in New York as seen in Figure 4 at the top of this example.

Placing a Custom Marker

Location information is frequently associated with a single point of interest. In such cases a region is not the best method to display your data. An icon or pin works better. This example will provide you with the basic tools necessary to place one or more pins on the map and to update their locations as time passes.

You will create a marker at one corner of Bryant Park, and proceed to update its position to that of the next corner, and then update it to the next in a continuous loop. The icon used for the marker ships with RealityVision, but could be any icon you choose, and deploy, to the RealityVision server. In fact, using functions to create the markers, similar to those in the polygon creation example above, would allow each marker, or type of marker, to utilize its own icon.

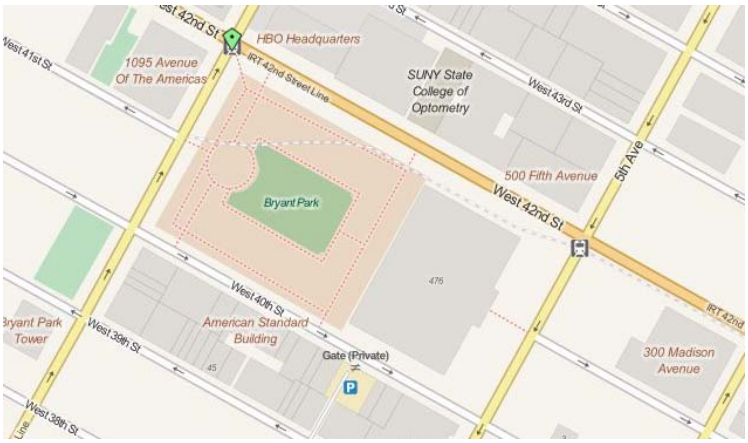

Figure 5: Map Markers

As before, this example will utilize some JavaScript methods. In this scenario it is actually necessary to do so because of the use of a timer. Timers require two things: a function to call, and how frequently to call it. The call cannot receive any data, so for it to access the existing context it must be tied to an object. This example creates a CustomLayerManager to be that object. It looks like this:

```
CustomerLayerManager = function (olMap) {
    // core secure map objects exposed for interaction
    // with OpenLayers and the RV Mapping support.
    this.map = olMap;
    this.secureMap = olMap.secureMap;

    // Layers
    this.pinLayer = null;

    // allocate layers
    this.pinLayer = new OpenLayers.Layer.Vector("Pin Layer");
    map.addLayer(this.pinLayer);
    map.setLayerIndex(this.pinLayer, 1);

    // Create demonstration data around Bryant Park, NY
    this.listOfLonLat = [
        this.secureMap.ConvertWGS84toWebMercator(
            -73.985043, 40.753556),
        this.secureMap.ConvertWGS84toWebMercator(
            -73.984124,40.754864),
        this.secureMap.ConvertWGS84toWebMercator(
            -73.980862,40.753483),
        this.secureMap.ConvertWGS84toWebMercator(
            -73.981828,40.752199)
    ];

    // Periodically fire off a position update (every 10 seconds)
    var layerManager = this;
    this.updatePinsTimer = setInterval(
        function () { layerManager.UpdatePins() }, 10000);

    // Perform an initial UpdatePins call to have the pin
    // display on startup.
    this.UpdatePins();
};
```

The comments added to this function perform a reasonably clean explanation of what is happening here. The following should help clarify a few points, however. The function is defined, as before, to actually be an object. That makes this function what is typically called a constructor. It accepts an

instance of the map object, available to the InitializeCustomerLayer function which, without being passed in, would be unknown to this object. When we create additional functions to work with this routine, we'll need that knowledge.

A list of points around Bryant Park is created after the layer is built:

```
this.listOfLonLat = {
        …
};
```

You should note the use of a utility method that is a part of the Secure Maps object: ConvertWGS84toWebMercator. This is a utility method that accepts a Longitude, Latitude pair and converts it into the maps projection mode. One of the reasons for storing the reference to the secure map object in the layer manager is so that if you convert this example into one that does make calls in the timer's callback function, you will have access to this conversion method.

The next item of interest is the timer routine itself. You will note the creation of a local variable called layerManager:

```
var layerManager = this;
```

At first glance this looks to be unnecessary, since it is only used to make the call to UpdatePins() in the setInterval function. What is tricky about this code pattern is the un-named function call:

```
function() { layerManager.UpdatePins() }
```

This creates an ambiguous function that is executed with no knowledge of the CustomerLayerManager object instance that establishes it. By creating the local variable, and assigning "this" to it, a memory reference to the CustomerLayerManager is created and is given to the anonymous function on its creation that retains knowledge of the CustomerLayerManager instance when the timer goes off.

Another point of interest in the setInterval call is the time value supplied. It is in milliseconds, so 10,000 means 10 seconds. If you implement a web service call mechanism, it is suggested to have the web service "wait" and only respond once new position data is available, instead of using a local timer like this, to conserve power on mobile devices. In that pattern you will want to periodically have the service respond with no updates to avoid timeouts, but keep that cycle as long as possible. That said, if you need a timer, also try to have its interval be set as long as is useful to avoid having devices constantly running JavaScript and consuming battery power.

The next step is to build the UpdatePins routine. Here is its code:

```
CustomerLayerManager.prototype.UpdatePins = function() {
    // Request updated data from a web service
```

```
        // Loop through the data, updating map feature
        // objects for items with new locations
        // or add items that were not previously received.

        // For purposes of this example the set of four
        // locations stored in this will be used instead.
        if (this.examplePin == null)
        {
            // Initialize a position counter used to track where
            // we are in the list.
            this.locationIndex = 0;

            var metaData = {
                iconUri: "JavaScript/OL_theme/img/marker-green.png",
                iconWidth: 21
            };

            // Create a CustomerMarker object – it uses a geometry
            // point object for its initial position instead of
            // an OpenLayers.LonLat object
            this.examplePin = new CustomerMarker(
                this.secureMap, "pin1",
                new OpenLayers.Geometry.Point(
                    this.listOfLonLat[this.locationIndex].lon,
                    this.listOfLonLat[this.locationIndex].lat),
                this.pinLayer, metaData );

            // Increment the index so the pin moves on
            // the first timer event.
            this.locationIndex++;
        }
        else {
            // The marker exists, so update its position instead
            // of creating a new one.
            this.examplePin.marker.move(
                this.listOfLonLat[this.locationIndex++]);

            // When the fourth value has been assigned,
            // the first value is used again.
            if (this.locationIndex > 3) {
                this.locationIndex = 0;
            }
        }
    };
```

By using the prototype syntax this function has access to the values defined in the CustomerLayerManager constructor illustrated above. The “if” statement allows this function to perform two actions: create a new marker if one does not exist, or update the location of an existing marker. These are the same two actions needed if a service routine is utilized to retrieve the data, the only difference would be that a list of known objects would need to be accessed to see if the object was present before making that decision. Null is used to check the value, if this.examplePin has been set to an object, its value will be non-null and the test will fall through to the “else” behavior which performs an update of the pin’s position.

The null case will only happen the very first time the function is called, so it is used here as a convenient way to initialize the counter for going through the list of four positions. In non-example code the first-time initialization check may look very different, as may its initialization code. However, this example should provide a solid foundation to build from, as any implementation of a mobile marker will have the need to initially create, and then update the marker.

The block for metaData can be expanded to store additional data depending on your needs. The next example will demonstrate this. For now, we set the address of the icon to be displayed, relative to the web server’s root path, and the width of the icon used. If the icon is smaller or larger than the size given OpenLayers will scale the image provided to be that width and adjust its height by the same percentage.

The call to create a new CustomerMarker object is similar in concept to the polygon example, above, where a CustomerShape was created. A discussion of the function, and its parameters, is given below. For now, the important item to understand is the code that supplies the location to the call:

```
new OpenLayers.Geometry.Point(
                this.listOfLonLat[this.locationIndex].lon,
                this.listOfLonLat[this.locationIndex].lat)
```

This is necessary since the CustomerMarker expects an OpenLayers.Geometry.Point object, but we want to keep the array as LonLat objects since OpenLayers expects the data in that format when updating positions of existing markers, as is discussed next.

The update behavior is triggered on each subsequent timer event after the first call. Moving an existing object is fairly simple, the actual vector object created has a move() routine that accepts an OpenLayers.LatLon object. As mentioned above, this is why the original list was created as LatLon objects provided by the secureMap.ConvertWGS84toWebMercator() call. If you are utilizing a method for retrieving data from a service, usage of this.secureMap.ConvertWGS84toWebMercator() can be performed here, because an instance member for the secureMap object was added in the constructor.

The final function needed for this example is the CustomerMarker object’s constructor:

```
CustomerMarker = function (secureMap, uniqueId, point, maplayer,
    metaData) {
```

```
        if (secureMap.listPopupMarkers[uniqueId] != null) {
            throw new Error("CustomerMarker(): Error: Marker with
        same unique ID already exists ('" + uniqueId + "')");
        }

        secureMap.listPopupMarkers[uniqueId] = this;

        this.secureMap = secureMap;
        this.uniqueId = uniqueId;
        this.layer = maplayer;
        this.lonlat = point;
        this.popupManager = this;   // Required for hiding errors
            // triggered by the secure map API in the next example.

        var marker_style = new OpenLayers.Util.extend({}, null);
        marker_style.graphicWidth = metaData.iconWidth;
        marker_style.externalGraphic = metaData.iconUri;
        this.marker = new OpenLayers.Feature.Vector(this.lonlat,
        null, marker_style);
        this.marker.metaData = metaData;
        this.marker.lonlat = this.lonlat;
        this.marker.popupMarker = this;

        this.layer.addFeatures([this.marker]);
    };
```

The parts you are most likely to be interested in are the setting of the graphicWidth and externalGraphic in the marker_style object. The OpenLayers.Util.extend routine is used to create a default object that OpenLayers understands and can perform its own operations upon. The code than gives it the first two values it will need to know to create a vector object.

There is some additional code in this routine that is important for the next example. To avoid revisiting this function later, the code is provided in its complete form now. You can ignore those portions and accept that those lines serve a later purpose.

The last step is once again to add the feature you created to the map by adding it to the assigned layer. So here's the InitializeCustomerLayer call needed to make this example run:

```
    SecureMaps.InitializeCustomerLayers = function () {
        var layerManager = new CustomerLayerManager( map );
    };
```

Because the layer code was moved to the CustomerLayerManager all the initialization call needs to do is create a new instance of one and provide it with a copy of the map.

What was presented here was a fairly basic example; using a timer to pull locations from a list. It was discussed that a more sophisticated approach would be to access a web service hosted by the RealityVision web site running as its own application. If you do so, have it run as part of the RealityVision site to simplify cross-domain request issues and SSL host Certificate acceptance complications.

To complete the example and see it in action, save the file and launch a client. The placemark should start to move around Bryant Park, once every 10 seconds.

Displaying Popups

Objects placed on the map by RealityVision display popups with information about user state, camera status, etc. The objects you place on the map can display their own data as well. To lay out that data requires a bit of HTML, CSS and some new JavaScript functions. This example starts with the JavaScript created for creating custom markers, shows how to add data for display, by modifying the existing functions you have, and then provides the new functions that will allow your custom markers to display and close popups as the user clicks or touches them. The results will look like this:

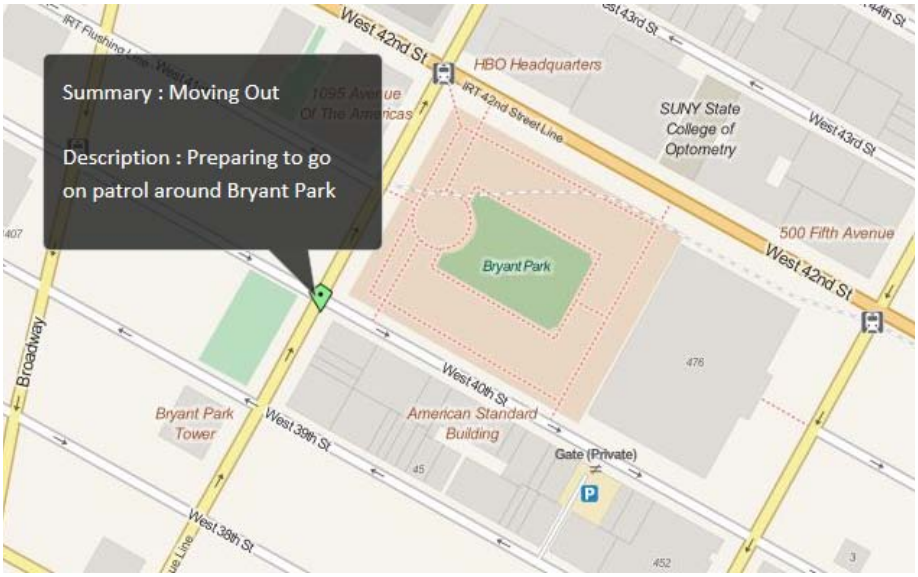

Figure 6: Initial popup display

The first task is to consider the content to be displayed. It should be kept in mind that RealityVision is designed for mobile devices. As such, consideration for smaller phone displays can be important. Having said that, the only limitation on what you can put in the popup is based on your skill with HTML, CSS and your access to the necessary content to display. Another thing to keep in mind is, if you put in a link that is not handled directly by the HTML you create, it will take you outside of the RealityVision client and spawn a separate device browser instance.

In this example the data to be added will be a summary and a description for the marker. To add this data we will need a new function that can update the pin. The CustomerMarker function from the last example is already complete. You just need to add some new functionality to make popups possible. The first thing to do is extend CustomerMarker by adding an Update routine:

```
CustomerMarker.prototype.Update = function(
        lonlat, summary, description ) {

    // Update the metadata.
    this.marker.metaData.popupSummary = summary;
    this.marker.metaData.popupDescription = description;

    // Move the marker on the map
    this.lonlat = lonlat;
    this.marker.move(lonlat);

    // If the popup is currently open, update it as well.
    // If the popup is currently open, update it as well.
    if (this.marker.popup != null)
    {
        // Update the popup's contents
        this.marker.popup.contentHTML = this.GetContent();

        // Set its location
        this.marker.popup.lonlat = this.lonlat;

        // Have the popup display the updated contents
        this.marker.popup.draw();
    }
};
```

The Update routine takes the three parameters: a location (lonlat), the summary and a description. To modify the customer marker with the updated summary and description, just over-write the earlier contents added when it was created, like so:

```
// Update the metadata.
this.marker.metaData.popupSummary = summary;
this.marker.metaData.popupDescription = description;
```

The marker's position is updated next; you will remove it from the UpdatePins call later. The code also saves off the new lonlat value, for use when the user clicks the marker to know where to place the popup:

```
// Move the marker on the map
this.lonlat = lonlat;
this.marker.move(lonlat);
```

Because the move is now performed by the control itself, the call no longer requires the examplePin reference as it did when in the updatePins call.

Next, the code checks to see if the popup is currently open. If it is, then it replaces its contents with a call to GetContent(), which is defined below, and then updates the popup's position and tells it to draw itself. The draw call triggers OpenLayers to re-read the assigned metadata and the existing popup appears at the new location with the updated text.

With Update complete it is time to specify the look of the content. As alluded to previously, an OpenLayers popup is designed to receive a block of HTML to display. The GetContent routine provides a single creator of that content that the CustomerMarker uses, so updates and initial creation do not duplicate this markup text:

```
CustomerMarker.prototype.GetContent = function () {
    var innerHtml =
        '<div class="olFramedCloudPopupCustomerMarker">' +
        '<div class="olFramedCloudCustomerPopupText">' +
            'Summary : ' +
                this.marker.metaData.popupSummary +
                '<br/><br/>' +
            'Description : ' +
                this.marker.metaData.popupDescription +
                '<br/>' +
        '</div>';

        return innerHtml;
};
```

The two div objects break up the HTML into an outer container and an inner container. The reason to do this is to ensure that the markup leaves a border around the text and control the popup's size. If raw HTML is given without CSS styles, OpenLayers will auto-determine the layout, which may not draw on screen in a manner you like. The two styles used here are: olFramedCloudPopupCustomerMarker and olFramedCloudCustomerPopupText. If you navigate to the JavaScript folder where the CustomerLayer.js file lives, you will find a sub-folder named OL_theme. There is a "style.css" file in this folder which you will want to add the two new styles used in this example.

The two styles look like this:

```
.olFramedCloudPopupCustomerMarker {
    width: 220px;
    height: 120px;
    text-align: left;
    background-color: transparent;
    overflow: hidden;
    color: #FFFFFF;
    pointer-events: none;
}

.olFramedCloudCustomerPopupText {
    margin: 6px;
    width: 202px;
    height: 108px;
    text-overflow: ellipsis;
    white-space: wrap;
    overflow: hidden;
    line-height: 140%;
}
```

The olFramedCloudPopupCustomerMarker sets the overall width and height desired for popups. It also specifies that:

- The contents will be left aligned.
- The text rendered on the popup will not render a background so the translucent gray look used by RealityVision will be retained.
- Any text that overflows the popup will be hidden.
- It will display all text as white.

The olFramedCloudCustomerPopupText class sets the inner section and further constrains the contents in this way:

- There will be a margin of 6 pixels around the text.
- Its height will be 108 pixels (12 pixels smaller, allowing for the 6 pixel margin, at top and bottom, from the above style)
- Its width should again be 12 pixels smaller than the outer style, as with the height.
- If text exceeds the size of this inner box, it will be truncated automatically and an ellipsis will be shown (i.e. "…").
- The text will wrap on white space, allowing the description to wrap over multiple lines.
- The overflow is marked as hidden, but the ellipsis to some degree covers this.
- The line-height value sets the spacing between lines, making it a little more spread out and easier to read.

This text formatting will have the display match that of the other RealityVision pop-ups.

If your content requires more space, adjust the height and/or width of both of these classes appropriately to make space. If your layout is more complex, additional div sections may be required to allow you to place your content as desired. You can also freely add bold, italic and other tags as appropriate, for formatting the content to be displayed.

To link the popup data to the pins the metadata object from the UpdatePins function in the last example requires modification. The GetContent function above assumed two values existed in the metadata object to display the summary and description. To provide this data, add popupSummary and popupDescription attributes to the existing metaData object. It should now look something like this:

```
var metaData = {
    popupSummary: "Moving Out",
    popupDescription:
        "Preparing to go on patrol around Bryant Park",
    iconUri: "JavaScript/OL_theme/img/marker-green.png",
    iconWidth: 21
};
```

The order doesn't actually matter, but if you place them after iconWidth, be sure to put a comma after the 21 or the JavaScript will not execute correctly. These two elements will be used in the new functions that follow.

You will recall that earlier you added the code to move the pin to the new Update function, so in the else block of the UpdatePins function, change the line for moving the marker to this:

```
// The marker exists, so update its position instead of
// creating a new one.
this.examplePin.marker.move(
    this.listOfLonLat[this.locationIndex++]);
this.examplePin.Update(
    this.listOfLonLat[this.locationIndex++],
        "On Patrol",
        "Currently on patrol around Bryant Park");
```

This completes the process of moving the responsibility for updating the marker to the marker itself. Additionally, the Update routine, when called, will change the contents of the summary and description text items. This is generally good coding practice and will help keep the JavaScript from becoming overly interdependent which could make it hard to update. Alternatively, you could pass in a new metadata object which you parse in the Update routine. For purposes of this example, this is just a little clearer.

Now that you have the popup designed, OpenLayers needs to know about your layer so that it can trigger events to select and de-select your features. OpenLayers provides a control specifically for this purpose that RealityVision adds its own layers to. Unfortunately, the InitializeCustomerLayer call is made slightly before the RealityVision layers are added to this control. If you add yours first, the RealityVision code will override your request to receive events. So, a second timer is needed to ensure that your layers are added after RealityVision's. We will set that up later, so for now the timer event's function will be:

```
CustomerLayerManager.prototype.SetSelection = function () {
    // Find the feature selection control
    for (var index in this.map.controls) {
        var control = this.map.controls[index];
        if (control.displayClass == 'olControlSelectFeature') {
            // Add any new layers to the OpenLayers
            // SelectFeature control to support item pop-ups
            var selectLayers =
                [this.pinLayer].concat(control.layers);
            control.setLayer(selectLayers);

            // The custom layers are now added to the
            // selection control, so stop the timer.
            window.clearInterval(this.setSelectionTimer);
        }
    }
};
```

This routine accesses the OpenLayers map object, and tries to find the SelectFeature control. If it is not there, it exits and waits for the next timer event. Once it is found, it retrieves the current list of layers from the control, appends them to an array of your layers that contain features you want click/touch events to be sent to. Then this routine stops the timer since the feature selection will have been initialized. If for some reason you add additional layers at a later time, you will need to call this routine, or one like it, again to add the additional layers. Just make sure that all layers that need to receive events are all added at the same time, including RealityVision layers, every time setLayer is called, otherwise portions of the UI will break. The line you need to modify is:

```
var selectLayers = [this.pinLayer].concat(control.layers);
```

The brackets '[]' define the array of layers you are adding. If you have more than one, this.pinLayer should be replaced with a comma separated list like this:

```
[this.layer1, this.layer2, this.layer3]
```

Do not add layers where click and touch events are not needed since that will just create extra work for the JavaScript to attempt to look for popup display content on the layer objects that will not be found.

Once your layer has been added to the feature selection control, you need to add the click event handler:

```
CustomerMarker.prototype.evtOnClick = function (evt) {
    if (!this.secureMap.popupsEnabled) {
        return;
    }

    if (this.marker.popup != null) {
        this.ClosePopup();
    }
    else {
        innerHtml = this.GetContent();

        this.marker.popup = new OpenLayers.Popup.FramedCloud(
            "PopupID",
            this.marker.geometry.getBounds().getCenterLonLat(),
            null, innerHtml, null, false,
            null);

        map.addPopup(this.marker.popup);
    }
};

CustomerMarker.prototype.ClosePopup = function () {
    var popup = this.marker.popup;
    if (null == popup) {
        return;
    }

    this.secureMap.map.removePopup(popup);
    popup.destroy();
    this.marker.popup = null;
};
```

These two routines handle opening and closing popups as the user clicks them. evtOnClick first checks to see if RealityVision has set the popupsEnabled flag in the SecureMaps layer. This flag is cleared for certain views where popups were not helpful for the RealityVision markers. If you desire to have your popups show for those views, you can remove this check and your popups will show in all views.

The rest of the function creates the actual popup with your content by calling the GetContent routine defined previously, or closes it, if the popup is already open, by calling ClosePopup defined below the evtOnCLick method. The ClosePopup routine checks to see if a popup is open, and if so, releases the popup's memory and sets the marker's popup object to null to let evtOnClick know there is no popup

open the next time it is called. Since these two routines are very general it is unlikely you will need to modify them. You can save them as helpful utility methods to be used when you need them.

The last remaining modification you need to make is to set up the timer that triggers SetSelection mentioned earlier. Add the timer to the last line of the CustomerLayerManager function just below the UpdatePins() call. The new code is:

```
// A second timer used to initialize OpenLayers with
// the layers used
this.setSelectionTimer = setInterval(
    function () { layerManager.SetSelection() }, 100);
```

If you recall, the CustomerLayerManager uses a timer for moving the marker on the map. When doing so it initialized the layerManager object, so you can use that again here without redefining it. This second timer triggers the attempt to set the selection handler once every 100 milliseconds, or 1/10th of a second. This ensures that the selection control is fully initialized before the user has a chance to see and select a control on the map.

As before, you can now save the file and run a client to try out the new additions. When you touch the marker it will pop up its bubble. If you do so before the first time it moves, you will see the initial text. Once it starts moving the bubble's text will change and orient itself to fit in the display, possibly scrolling the map to allow it to show.

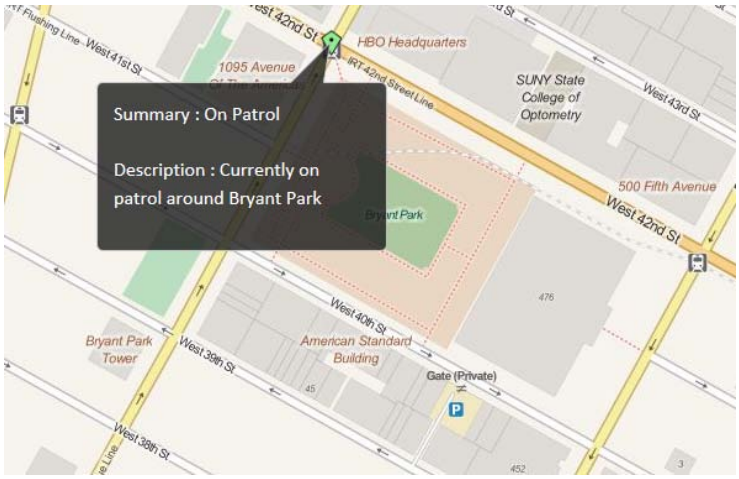

Figure 7: Popup after first move

Summary

Hopefully this guide has given you the basic tools necessary to start to integrate your data with RealityVision in a new and powerful manner. There are even more objects OpenLayers allows you to add to the mapping layers. By leveraging your organization's knowledge of where its events, resources or even external data to which it has access, you can begin to bring a whole new level of context into the hands of your RealityVision user base.

To help you expand on this initial introduction, please utilize the following resources to expand your understanding of what is possible.

Resources

HTML Color Picker: http://www.w3schools.com/tags/ref_colorpicker.asp

OpenLayers API Documentation: http://dev.openlayers.org/releases/OpenLayers-2.13.1/doc/apidocs/files/OpenLayers-js.html

JavaScript Language Documentation: http://www.w3schools.com/js/js_intro.asp