How to create a marker tooltip with Maps API for Java ME

This article explains how to associate **tooltip** text to a `MapObject`. This means that when the `MapObject` is over the centre of the screen, the **tooltip** text is visible.

15 Apr 2012

Archived: This article is **archived** because it is not considered relevant for third-party developers creating commercial solutions today. If you think this article is still relevant, let us know by adding the template {{ReviewForRemovalFromArchive|user=~~~~|*write your reason here*}}.

The latest 1.3 release of the HERE Maps API for Java ME now includes an integrated touchable component framework. This also includes the code for the Tooltip component - see HERE Map Code Examples. These components are fully backwards compatible for older phones . The API has been integrated as a plug-in into the Asha SDK 1.0.

Introduction

A **tooltip** is a GUI element that appears when the focus lies over an associated point of interest. It usually contains some text with more information about the item concerned, typically a short descriptive label. This leaves the screen uncluttered by hiding these summaries unless the point of interest has the focus. Given that a typical mobile screen is quite small, the usage of the screen real estate on mobile devices is at a premium. Avoiding displaying additional texts on the map unless required should make the underlying map more readable and therefore increase the usability of the application.

Definition of the Issue

The issue can be split into three separate problems:

- The need to know when a `MapObject` holds the focus - i.e. when it is centered on the `MapDisplay`.
- The need to know what text (if any) is associated with the `MapObject`.
- The need to display a **tooltip** at an appropriate point on the `MapDisplay`, with the text wrapping in a readable manner.

Creating a Custom MapComponent

How to tell when a marker holds the focus

In order to interact with map events, a custom `MapComponent` must be created. The `MapComponent` must respond to the `mapUpdated()` method in order to see if a `MapObject` can be found at the current centre of the `MapDisplay`. Eventually we'll also need to implement a `paint()` method to display the **tooltip**. Implementations of the `attach()` and `detatch()` methods are also required to remember

(C) Copyright Nokia 2014. All rights reserved.

the `MapDisplay`. The other required methods can take default values. Since this `MapComponent` does not fire any events itself, the `EventListener` can be returned as `null`.

```
public class TooltipComponent implements MapComponent {

    private MapDisplay map;
    private static final String ID = "TooltipComponent";
    private static final String VERSION = "1.0";

    public void attach(MapDisplay map) {
        this.map = map;
    }

    public void detach(MapDisplay map) {
        this.map = null;
    }

    public String getId() {
        return ID;
    }

    public String getVersion() {
        return VERSION;
    }

    public EventListener getEventListener() {
        return null;
    }

    public void mapUpdated(boolean zoomChanged) {
        MapDisplay currentMap = map;
        if (currentMap != null) {
            MapObject focusMO = getCurrentFocus(currentMap);

 ....
        }
    }

    public void paint(Graphics g) {
...
    }
}
```

Associating tooltip text to a marker

Not all `MapObjects` will have a **tooltip** associated with them. It is therefore necessary for the `TooltipComponent` to hold a list of `MapObjects` with a **tooltip**, and to keep track of the latest **tooltip** text. Internally, the items are held in a `Hashtable`, and accessed using the current `MapObject` as the key. The methods `setTooltip()` and `removeTooltip()` merely hide the implementation of the `Hashtable`.

```
public class TooltipComponent implements MapComponent {
...
    private final Hashtable tooltipTexts;

    public TooltipComponent() {
```


(C) Copyright Nokia 2014. All rights reserved.

```
        tooltipTexts = new Hashtable();
    }

    public void setTooltip(MapObject mo, String tooltip) {
        if (mo != null ){
               tooltipTexts.put(mo, tooltip);
        }
    }

    public void removeTooltip(MapObject mo) {
        tooltipTexts.remove(mo);
    }

    public void mapUpdated(boolean zoomChanged) {
        MapDisplay currentMap = map;
        if (currentMap != null){
            MapObject focusMO = getCurrentFocus(currentMap);

             if (focusMO != null && tooltipTexts.containsKey(focusMO)) {
                 renderer.setTooltip(getAnchor(currentMap, focusMO), (String)
tooltipTexts.get(focusMO));
             }
     else {
                renderer.clearTooltip();
            }
        }
    }


[...]


    /**
     *
     * @param currentMap
     * @return the MapObject at the center of the map.
     */
    private MapObject getCurrentFocus(MapDisplay currentMap){
        Point center = new Point(currentMap.getWidth() / 2, currentMap.getHeight() / 2);
        return currentMap.getObjectAt(center);
    }


}
```

This means that a **tooltip** can be added to a `MapObject` with the following code:

```
tooltips = new TooltipComponent();
map.addMapComponent(tooltips);
marker = mapFactory.createStandardMarker(new GeoCoordinate(51.477811, -0.001475,
Float.NaN));
tooltips.setTooltip(marker, "Royal Observatory, Greenwich");
```

Creating the Tooltip Text

(C) Copyright Nokia 2014. All rights reserved.

How to create readable wrap around text

When creating the **tooltip** text, the object needs to know three things: the text to be displayed, the font to use and the maximum width of the `Display`. Using these values it is possible to calculate if the text will display comfortably over a single line or not. If a single line of text would not be entirely visible, it is necessary to split the text over several lines. Initially the text can be split into words using an implementation of a `String.split()` function, and then the list of the words recombined adding line breaks where necessary.

```
public class TooltipText {

    private final String[] texts;
   private final Font font;
...

    public TooltipText( String tooltipText, Font font, int maxTooltipWidth) {
        this.font = font;
        texts = splitTextIntoLines(tooltipText,  maxTooltipWidth);
...
    }

   private String[] splitTextIntoLines(String text,  int maxLineWidth) {
        if (font.stringWidth(text) < maxLineWidth) {
            return new String[]{text};
        }
        return multipleLinesOfText(text, maxLineWidth);
    }

  private String[] multipleLinesOfText(String text,  int maxLineWidth) {
        Vector lines = new Vector();


        StringBuffer buf = new StringBuffer();
        String[] words = splitIntoWords(text);

        for (int i = 0; i < words.length; i++) {

            if (font.stringWidth(buf.toString() + words[i]) > maxLineWidth) {
                lines.addElement(buf.toString());
                buf = new StringBuffer(words[i]);
            } else {
                buf.append(words[i]);
            }
            buf.append(" ");
        }
        lines.addElement(buf.toString());


        String[] textArray = new String[lines.size()];
        lines.copyInto(textArray);

        return textArray;
    }

    private String[] splitIntoWords(String tooltipText) {
        Vector words = new Vector();
```


(C) Copyright Nokia 2014. All rights reserved.

```
        int index = tooltipText.indexOf(" ");
        while (index >= 0) {
            words.addElement(tooltipText.substring(0, index));
            tooltipText = tooltipText.substring(index + 1);
            index = tooltipText.indexOf(" ");
        }

        words.addElement(tooltipText);
        String[] wordsArray = new String[words.size()];
        words.copyInto(wordsArray);

        return wordsArray;
    }
}
```

How to display tooltip text on screen

The lines of text can be displayed on screen through an implementation of a `paint()` method. Note that the calling method supplies the position on the screen and the anchor, the text just renders itself sequentially over a series of lines.

```
 public void paint(Graphics g,  int x, int y, int anchor) {
        for (int i = 0; i < texts.length; i++) {
            g.drawString(texts[i], x, y + (i * font.getHeight()), anchor);
        }
    }
}
```

It will be useful for other objects to know how much of the screen will be covered by the tooltip text. The `getWidth()` and `getHeight()` methods can supply this information, they are used by the `TooltipRenderer` class defined below. The values are calculated in the constructor.

```
public class TooltipText {

...
    private final int width;
    private final int height;

    public TooltipText( String tooltipText, Font font, int maxTooltipWidth) {
...
        width = calculateWidth();
        height = font.getHeight() * texts.length;
    }

    private int calculateWidth(Font font) {
        int maxWidth = 0;
        for (int i = 0; i < texts.length; i++) {
            int textWidth = font.stringWidth(texts[i]);
            maxWidth = (maxWidth > textWidth) ? maxWidth : textWidth;
        }
        return maxWidth;
    }

    public int getWidth() {
        return width;
```


(C) Copyright Nokia 2014. All rights reserved.

```
    }

    public int getHeight() {
        return height;
    }
}
```

Rendering the Complete Tooltip

A proper **tooltip**, consists of more than just text on the map. For an aesthetically pleasing display, the **tooltip** should place the text on a rectangle of a specified background colour, and optionally surround the rectangle with a border. Additionally options should be available to alter the margins, width of the borders and the `Font` of the rendered text. These requirements can be fulfilled through a separate class responsible for the **tooltip** "look-and-feel" which in turn delegates the rendering of the **tooltip** text rendering to the `TooltipText` class.

As you would expect, all the constants for the "look-and-feel" need to be set up in the constructor. A default render style can also be supplied, the values here specify blue text on a white background with a black border, but can be altered as you wish.

```
public class TooltipRenderer {

...
    private final int margin;
    private final int borderColor;
    private final int background;
    private final int textColor;
    private final int borderAndMargin;
    private final int maxWidth;
    private final Font font;

...
    public TooltipRenderer(int maxWidth, int border, int margin, int borderColor,
            int background, int textColor, Font font) {
        this.margin = margin;
        borderAndMargin = border + margin;
        this.borderColor = borderColor;
        this.background = background;
        this.textColor = textColor;
        this.font = font;
        this.maxWidth = maxWidth;
    }

    public static TooltipRenderer getDefaultRenderer(int maxWidth) {
        return new TooltipRenderer(maxWidth, 2, 2, 0x000000, 0xFFFFFF, 0x00007F,
Font.getDefaultFont());
    }
```

The width and height of the current **tooltip** will vary as the text of the **tooltip** is altered. The x and y offsets for the border, background and **tooltip** textbox are held in class variables, and altered whenever the `setTooltip()` method is called

```
public class TooltipRenderer {

    private static final int SMALL_CORNER_ARC = 5;
    private static final int Y_OFFSET = 2;
```


(C) Copyright Nokia 2014. All rights reserved.

```
...

    private Point borderAnchor;
    private Point backgroundAnchor;
    private Point backgroundRect;
    private Point borderRect;
    private TooltipText textElement;


    public void setTooltip(Point anchor, String tooltipText) {

        textElement = new TooltipText(tooltipText, font, maxWidth);


        this.textAnchor = new Point(anchor.getX() - (textElement.getWidth() / 2),
                anchor.getY() + Y_OFFSET + borderAndMargin);
        this.borderAnchor = new Point(anchor.getX() - (textElement.getWidth() / 2) -
borderAndMargin,
                anchor.getY() + Y_OFFSET);
        this.backgroundAnchor = new Point(anchor.getX() - (textElement.getWidth() / 2) -
margin,
                anchor.getY() + Y_OFFSET + borderAndMargin - margin);

        this.backgroundRect = new Point(textElement.getWidth() + (margin * 2),
textElement.getHeight() + (margin * 2));
        this.borderRect = new Point(textElement.getWidth() + (borderAndMargin * 2),
textElement.getHeight() + (borderAndMargin * 2));
    }

    public void clearTooltip() {
        textElement = null;
    }
}
```

The complete **tooltip** is rendered by a call to its `paint()`} method.

```
public void paint(Graphics g) {

        if (textElement != null) {

            g.setColor(borderColor);
            g.fillRoundRect(borderAnchor.getX(), borderAnchor.getY(),
                    borderRect.getX(), borderRect.getY(), SMALL_CORNER_ARC,
SMALL_CORNER_ARC);

            g.setColor(background);
            g.fillRoundRect(backgroundAnchor.getX(), backgroundAnchor.getY(),
backgroundRect.getX(),
                    backgroundRect.getY(), SMALL_CORNER_ARC, SMALL_CORNER_ARC);

            g.setColor(textColor);
            textElement.paint(g,  textAnchor.getX(), textAnchor.getY(),  Graphics.TOP |
Graphics.LEFT);
        }
    }
```


(C) Copyright Nokia 2014. All rights reserved.

Linking up the TooltipComponent display the tooltip

In order to display a proper **tooltip**, the `TooltipComponent` needs to create a Tooltip renderer, set and clear the **tooltip** text and delegate the painting of the tooltip. The necessary plumbing code is shown below:

```
public class TooltipComponent implements MapComponent {

...
    private TooltipRenderer renderer;
...

    public void attach(MapDisplay map) {
        this.map = map;
        renderer = TooltipRenderer.getDefaultRenderer(map.getWidth());
    }

    public void mapUpdated(boolean zoomChanged) {
        if (map != null) {
            Point center = new Point(map.getWidth() / 2, map.getHeight() / 2);
            MapObject focusMO = map.getObjectAt(center);

            if (focusMO != null && tooltipTexts.containsKey(focusMO)) {
                renderer.setTooltip(getAnchor(focusMO), (String)
tooltipTexts.get(focusMO));
            }
            else {
                renderer.clearTooltip();
            }
        }
    }

    private Point getAnchor(MapDisplay currentMap, MapObject mapObject) {

        if (mapObject instanceof MapStandardMarker) {
            return currentMap.geoToPixel(((MapStandardMarker)
mapObject).getCoordinate());
        }
        else if (mapObject instanceof MapMarker) {
            return currentMap.geoToPixel(((MapMarker) mapObject).getCoordinate());
        }

        return currentMap.geoToPixel(new GeoCoordinate(
            mapObject.getBoundingBox().getBottomRight().getLatitude(),
            mapObject.getBoundingBox().getCenter().getLongitude(), 0f));
    }

    public void paint(Graphics g) {
        renderer.paint(g);
    }
}
```

Extending the Tooltip Concept to a KMLResultSet

(C) Copyright Nokia 2014. All rights reserved.

Keyhole Markup Language (**KML**) is an XML notation for geographic applications. It allows both two-dimensional maps and three-dimensional maps to display additional information overlaying the basic map. The Maps API for Java ME is able to read and render **KML** files in the form of the `KMLResultSet` class. This class also offers a `KMLEventListener` interface to fire a `onFocusChanged()` event when highlighting occurs, so we merely add an additional overload for `TooltipComponent.setTooltip()` which will add **tooltip** text to the `MapObject` at the center of the current `MapDisplay`

1) Add in the hook to the `KMLEventListener`, associate the `<Feature>` text as required.

```
public void onFocusChanged(Feature placeMark) {
...
            tooltip.setTooltip(placeMark.getName());
    }
```

2) When the overloaded `setTooltip()` method is requested **without** a `MapObject` being passed into the function, find the `MapObject` at the center of the attached `map` and add it to the list of objects with tooltips.

```
   public void setTooltip(String tooltip){
        if (map != null){
            Point center = new Point(map.getWidth() / 2, map.getHeight() / 2);
           MapObject mo = map.getObjectAt(center);
            setTooltip(mo, tooltip);
        }
    }
```

Summary

A **tooltip** renderer for `MapObjects` can be created via a custom `MapComponent`. The associated Code Example contains all the necessary details to describe how it can be done. The custom `MapComponent` can also be extended to handle a `KMLResultSet` and display **tooltips** for **KML** data. The code behind this example and other custom map components can be found in the Map Components project.

The media player is loading...

(C) Copyright Nokia 2014. All rights reserved.