Creating Touchable Custom Map Components for the Maps API for Java ME

This article explains how to create a framework for custom `MapComponents` which respond to touch Events. A series of *touchable* components (scale bar, button, attribution image) are then added using the framework.

17 Jun 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 article is believed to be still valid for the original topic scope.

Introduction

In order to maximize the screen real estate available to display a Map, it is necessary to invoke **Full Screen** mode. This can be done when the `MapCanvas` is initialised, through a call to `MapCanvas.setFullScreenMode(true)`. `Command` buttons cannot be used when a full screen is displayed, and therefore it is necessary to add at least one `MapComponent` to allow the user to interact with the Map. This article proposes a framework for creating *touchable* `MapComponents` and a series of components such as a custom `Command` button bar and a scale bar are added to an example `MIDlet`

TouchableDisplayComponent

The job of the `TouchableDisplayComponent` can be conveniently split into three parts.

- The component must display something (usually an `Image`) when added to the Map
- The component may require a background to be displayed behind the `Image` to avoid the `Image` bleeding into the display.
- The component, a subclass or a nominated delegate needs to be able to handle touch events.

Displaying an Image

Displaying an `Image` is easy, it is merely a matter of overriding the default `paint()` event and calling `Graphics.drawImage()`, further code can be taken from the How to create a marker tooltip with Maps API for Java ME example to add a rounded box with a border around it. An example `paint()` method is shown below, where constants such as `borderColor`. `borderWidth` and `backgroundColor` are set in the constructor.

```
    public TouchableDisplayComponent(Image bitmap, int anchor, int border, int borderColor,
            int backgroundColor) {
        this.bitmap = bitmap;
        this.border = border;
        this.borderColor = borderColor;
        this.backgroundColor = backgroundColor;
        this.anchor = anchor;
...
    }
```

```
public void paint(Graphics g) {

        if (border > 0) {
            g.setColor(borderColor);
```

 (C) Copyright Nokia 2013. All rights reserved.

```
            g.fillRoundRect(getBorderAnchor().getX(), getBorderAnchor().getY(),
                    getImageWidth() + (getBorder() * 2), getImageHeight() + (getBorder() * 2),
                    SMALL_CORNER_ARC, SMALL_CORNER_ARC);
        }

        if (backgroundColor > NO_FILL) {
            // Draw the background.
            g.setColor(backgroundColor);
            g.fillRoundRect(getBackgroundAnchor().getX(), getBackgroundAnchor().getY(),
                    getImageWidth(),
                    getImageHeight(), SMALL_CORNER_ARC, SMALL_CORNER_ARC);
        }

        if (bitmap != null) {
            g.drawImage(bitmap,
                    getBackgroundAnchor().getX(), getBackgroundAnchor().getY(),
                    Graphics.TOP | Graphics.LEFT);
        }

    }
```

The location of the bitmap image to display on the `MapDisplay` can only be defined once the component has been attached to a `map`. This must be calculated in the form of x and y in pixels. For ease of use the `TouchableDisplayComponent` can be attached to any corner of a `MapDisplay` since it uses the enumerated anchor locations defined in the `Graphics` class

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

    protected void calculateBorders(MapDisplay map) {
        int x = getOffset().getX();
        int y = getOffset().getY();
        if ((getAnchor() & Graphics.BOTTOM) != 0) {
            y = map.getHeight() - getImageHeight() - getOffset().getY();
        }
        if ((getAnchor() & Graphics.RIGHT) != 0) {
            x = map.getWidth() - getImageWidth() - getOffset().getX();
        }

        setBorderAnchor(new Point(x - getBorder(),
                y - getBorder()));
        setBackgroundAnchor(new Point(x, y));
    }
```

Creating an EventHandler

The `TouchableDisplayComponent` contains a private class to handle touch events (all other events are ignored), and a `TouchEventListener` to delegate events through a known interface. This `TouchEventListener` is added through the usual system of getters and setters. The `pointerPressed()` method of the private class checks to see if a touch has occurred within the target area, and if it has, calls the `doClick()` method

```
        public boolean pointerPressed(int x, int y) {

            if (x > getBackgroundAnchor().getX()
                    && x < getBackgroundAnchor().getX() + getImageWidth()
                    && y > getBackgroundAnchor().getY()
                    && y < getBackgroundAnchor().getY() + getImageHeight()) {
                this.component.doClick(x - getBackgroundAnchor().getX(), y
                        + getBackgroundAnchor().getY());
                return true;

            }

            return false;
        }
```

The default implementation of the `doClick()` method searches for a `TouchEventListener` and passes thread of control to the listener if one exists.

```
    public boolean doClick(int x, int y) {
        if (listener != null) {
            listener.touchAction(this, x, y);
            return true;
        }
        return false;
    }
```

Example TouchableDisplayComponents

Using the base class described above it is possible to create a variety of *touchable* components

TouchTextComponent

The `TouchTextComponent` extends the `TouchableDisplayComponent` class to display text rather than an image. This is achieved by passing in a `null` image in the constructor and overriding `paint()`. Once the border and background have been painted, the text can be added as shown. **Note** : the width and height of the background are now based on the length of the text. The call to `paint()` is conditional on text being present.

```
public class TouchTextComponent extends TouchableDisplayComponent {
...
    protected int getImageHeight() {
        return font.getHeight() + 4;
    }

    protected int getImageWidth() {
        return font.stringWidth(text) + (2 * TEXT_MARGIN);
```

 (C) Copyright Nokia 2013. All rights reserved.

```
    }

        public void paint(Graphics g) {
        if ("".equals(getText()) == false) {
            super.paint(g);
            g.setColor(textColor);
            g.drawString(
                    getText(), getBackgroundAnchor().getX() + TEXT_MARGIN, getBackgroundAnchor().getY(), Graphics.TOP | Graphics.LEF
        }
    }
}
```

AttributionMapComponent

see also: Map Overlay Example

It is also possible to conditionally display and react to an image, through overriding `paint()` and `doClick()` This example is for use with overlays, and will only display if an overlay is present.

```
public class AttributionMapComponent extends TouchableDisplayComponent {
...
    public void paint(Graphics g) {
        if (getMap() != null && getMap().getAllMapOverlays().length > 0) {
            super.paint(g);
        }
    }
...
    public boolean doClick(int x, int y) {
        if (getMap() == null || getMap().getAllMapOverlays().length == 0) {
            return false;
        }
        return super.doClick( x,  y);
    }
}
```

Tip: For simple `MapComponent` behaviour such as this, it would make more sense to `attach/detach` the `MapComponent` itself when the overlay is added/removed, but this code sample has been created as an example of a **conditional** `MapComponent` which alters its behaviour dependent upon the state of the `MapDisplay`

ScaleBarComponent

The following `TouchableDisplayComponent` creates a scale bar which is sized according to the zoom level. In the example the code below only the first eighteen zoom levels (0-17) are catered for, but this can be expanded if necessary. Since the Map uses the normalized Mercator projection, the length of the scale bar needs to be altered depending on the latitudes displayed on the map. Clicking on the scale bar will switch between Metric and Imperial Measurements. This component is self contained, and doesn't need to delegate its events. It overrides `doClick()` and handles it in its entirety.

```
public class ScaleBarComponent extends TouchTextComponent {

    private int y_coord;
    private int x_coord_start;
    private int x_coord_end;
    private boolean imperial;
    final static int[] SCALE_IN_METRES = new int[]{5000000, 2000000, 1000000,
        600000, 300000, 150000, 75000, 30000, 20000, 10000, 5000, 2000, 1000,
        500, 250, 100, 50, 25};
    final static String[] SCALE_IN_METRES_TEXT = new String[]{"5000km", "2000km", "1000km",
        "600km", "300km", "150km", "75km", "30km", "20km", "10km", "5km", "2km", "1km",
        "500m", "250m", "100m", "50m", "25m"};
    final static int[] SCALE_IN_IMPERIAL = new int[]{4828000, 2414000, 1207000, 643737, 321868, 160934, 80467, 40233, 24140,
        16093, 8046, 3218, 1609, 457, 228, 91, 45, 22};
    final static String[] SCALE_IN_IMPERIAL_TEXT = new String[]{"3000 miles", "1500 miles", "750 miles",
        "400 miles", "200 miles", "100 miles", "50 miles", "25 miles", "15 miles", "10 miles", "5 miles", "2 miles", "1 mile",
        "500 yds", "250 yds", "100 yds", "50 yds", "75 ft"};

   ... etc

    public void mapUpdated(boolean zoomChanged) {


        x_coord_end = getBackgroundAnchor().getX() + getImageWidth() - TEXT_MARGIN;
        y_coord = getBackgroundAnchor().getY() + getImageHeight() - TEXT_MARGIN;
        double zoom = getMap().getZoomLevel();
        zoom = (zoom >= getMap().getMinZoomLevel()) ? zoom : getMap().getMinZoomLevel();
        if (isImperial()) {
            displayImperial(zoom);
        } else {
            displayMetric(zoom);
        }
        calculateBorders(getMap());
    }

    private void displayMetric(double zoom) {
        if (zoom <= SCALE_IN_METRES.length - 1) {
            GeoCoordinate end = getMap().pixelToGeo(new Point(x_coord_end, y_coord));
            x_coord_start = x_coord_end - 10;
            while (getMap().pixelToGeo(new Point(x_coord_start, y_coord)).distanceTo(end) < SCALE_IN_METRES[(int) zoom]) {
                x_coord_start--;
            }
            setText(SCALE_IN_METRES_TEXT[(int) zoom]);
        } else {
            setText("");
        }
    }

    private void displayImperial(double zoom) {
        if (zoom <= SCALE_IN_IMPERIAL.length - 1) {
            GeoCoordinate end = getMap().pixelToGeo(new Point(x_coord_end, y_coord));
            x_coord_start = x_coord_end - 10;
            while (getMap().pixelToGeo(new Point(x_coord_start, y_coord)).distanceTo(end) < SCALE_IN_IMPERIAL[(int) zoom]) {
                x_coord_start--;
            }
            setText(SCALE_IN_IMPERIAL_TEXT[(int) zoom]);
```

 (C) Copyright Nokia 2013. All rights reserved.

```
        } else {
            setText("");
        }
    }

    public void paint(Graphics g) {
        if ("".equals(getText()) == false) {
            super.paint(g);

            g.setColor(000000);
            g.drawLine(x_coord_start, y_coord, x_coord_end, y_coord);
            g.drawLine(x_coord_start, y_coord + 1, x_coord_end, y_coord + 1);

            g.drawLine(x_coord_start, y_coord + 3, x_coord_start, y_coord);
            g.drawLine(x_coord_end, y_coord + 3, x_coord_end, y_coord);
        }
    }
    public boolean doClick(int x, int y) {
        setImperial(!isImperial());
        mapUpdated(false);
        return true;
    }

    public boolean isImperial() {
        return imperial;
    }

    public void setImperial(boolean imperial) {
        this.imperial = imperial;
    }
}
```

Wiring up the Components

Since the scale bar handles its own `doclick()` , it can be added to a map directly as a component

```
 map.addMapComponent(new ScaleBarComponent(Graphics.BOTTOM | Graphics.LEFT, 2, 0x000000,
                0xFFFFFF));
```

Any other `TouchableDisplayComponent` also needs to set its `TouchEventListener`

```
touchText = new TouchTextComponent(Graphics.BOTTOM | Graphics.RIGHT,
                2, 0x000000,
                0xFFFFFF, 0xFF0000);
        touchText.setText(" ON | EXIT");
        touchText.setEventListener(this);
        map.addMapComponent(touchText);
```

When placing multiple components in the same corner, use the `setOffset()` method to stop them overlapping.

```
touchText.setOffset(new Point(5, 40));
```

Extending the UI : Adding Highlighting etc.

The coded example places a simple touchable area on screen. To fully comply with the Full Touch UI Guidelines for buttons, the example must be extended to add highlighting and for actions to be cancelable if the touch is dragged outside of the hit area of the button. A discussion of the necessary framework to do this is beyond the scope of this article, but has been created in the Map Components project. Four videos showing highlightable buttons can be seen below.

The media player is loading...

The media player is loading...

The media player is loading...

Summary

As can be seen from the Worked Example , it is possible to use a framework to avoid duplicating code when working with `MapComponents`. The problem of small screen sizes may be overcome to a certain extent through using the full screen mode. It should be noted however that each `MapComponent` will use up valuable screen real estate, and their use should be kept to a minimum to avoid clutter on the screen.

 (C) Copyright Nokia 2013. All rights reserved.

 (C) Copyright Nokia 2013. All rights reserved.