Creating LCDUI Custom Components: CategoryBar

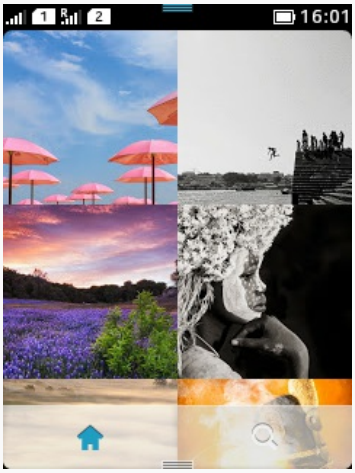
This article explains how to create a translucent custom category bar component that can be deployed in Canvas. The component has a similar API and appearance to the Nokia Asha UI's native component.

01 Dec 2013

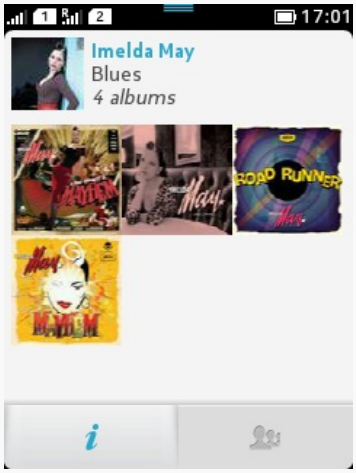
Introduction

The default category bar provided by the Nokia UI API is handy for creating tabbed based user interfaces (UIs) or to have easily accessible actions in the view. It looks nice and the use of the bar for users is very intuitive. On the downside, the component does take a lot of room, reducing the space for the actual content. Changing the opacity of the component would help in some cases, but unfortunately that is not supported by the API.

This article shows how you can create your own custom transparent category bar for implementing a tabbed UI in Canvas (action-based category bars are not fully supported). Familiarity with the native API is helpful but not assumed.



Our translucent custom category bar in the PicasaViewer example.

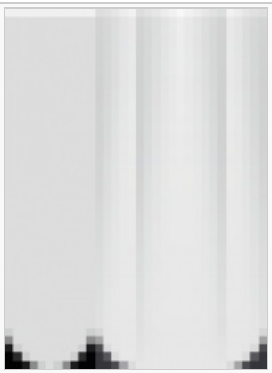


Native CategoryBar in the Nokia Asha Music Explorer example.

Tab bar in the native Gallery app.

Getting graphics assets

Images are are needed for all parts of the category bar in all of its states (ie left and right corners, middle texture, when both selected and unselected). You can use the assets in the design library or screenshots from the emulator. I had an image of a category bar with two tabs of which one was the selected tab. I then extracted the parts I needed to an single image using Gimp:



Graphics assets for the custom category bar in a single image.

The assets image consists of the following bits (from left to right):

1. The left corner of a unselected tab on left-most side of the view
2. One pixel wide texture of a unselected tab
3. The right corner of a unselected tab on right-most side of the view
4. The left corner of a selected tab on left-most side of the view
5. The left corner of a selected tab
6. One pixel wide texture of a selected tab
7. The right corner of a selected tab
8. The right corner of a selected tab on right-most side of the view

The icons for the tabs work just like they do with the native category bar. You only have to provide images for the unselected icons - icons selected items are created dynamically in the code (implementation shown later in this article).

Implementation

Architecture

It would have been really convenient, if the custom component had derived from both the CategoryBar and the CanvasGraphicsItem. However, multi-inheritance is not allowed in Java so another approach had to be taken. Since I still wanted the class to be able to render itself, I put a separate inner class to my `CustomCategoryBar.java` called `CustomCategoryBarRenderer`. This was kind of a workaround; inner classes have the properties of the outer class i.e. they can access the private members of the outer class, but it's not multi-inheritance in the full sense because polymorphism doesn't exist here. The basic structure of the custom category bar class then is as follows:

```
public class CustomCategoryBar extends CategoryBar {
    ...
    private class CustomCategoryBarRenderer extends CanvasGraphicsItem {
    ...
    }
}
```

What is noticeable here is that the main class, `CustomCategoryBar`, has is-a relationsip with the native `CategoryBar` and has the same API and you can associate an ElementListener to it. However, when you want to access the new methods of the custom category bar, it has to be cast into itself:

```
CategoryBar customCategoryBar = CustomCategoryBar.getNewInstance(...); // This works
customCategoryBar.setOpacity(0.5f); // This does not work since CategoryBar does not have a method called setOpacity
((CustomCategoryBar)customCategoryBar).setOpacity(0.5f); // This works
```

Creating the toolbar from the graphics assets

First, we need to load the image asset file. Then we need to process it to get the separate bits and pieces that we need:

```
    /**
     * For rendering our custom category bar.
     */
    private class CustomCategoryBarRenderer extends CanvasGraphicsItem
    {
        // Constants
        private final String ASSETS_IMAGE_URI = "/categorybar-assets.png";

        /*
         * The indeces of ASSETS_START_XS and ASSET_WIDTHS are aligned with the
         * "enumeration" introduced after (SELECTED_TAB_LEFT_EDGE etc.)
         */
        private final int[] ASSET_START_XS = {
            11, 16, 27, 22, 21, // For selected
            0, 6, 5 // For not-selected
        };
        private final int[] ASSET_WIDTHS = {
            5, 5, 5, 5, 1, // For selected
            5, 5, 1 // For not selected
        };

        private static final int SELECTED_TAB_LEFT_CORNER_EDGE = 0;
        private static final int SELECTED_TAB_LEFT_EDGE = 1;
        private static final int SELECTED_TAB_RIGHT_CORNER_EDGE = 2;
        private static final int SELECTED_TAB_RIGHT_EDGE = 3;
        private static final int SELECTED_TAB_TEXTURE = 4;
        private static final int TAB_LEFT_CORNER_EDGE = 5;
        private static final int TAB_RIGHT_CORNER_EDGE = 6;
        private static final int TAB_TEXTURE_FULL = 7;
        private static final int TAB_TEXTURE_WITH_ONE_EDGE = 8;
        private static final int TAB_TEXTURE_WITH_TWO_EDGES = 9;
        private static final int IMAGE_ASSET_COUNT = 10;

        // Members
        private Image _opaqueImages[] = null;
        private Image _translucentImages[] = null;
        private Image _currentImages[] = null;

...

        /**
         * Creates the image assets used by the category bar.
         */
        private void createImages() {
            System.out.println("CustomCategoryBarRenderer.createImages()");
            Image assetsImage = null;

            try {
                assetsImage = Image.createImage(ASSETS_IMAGE_URI);
            }
            catch (IOException e) {
                System.out.println("CustomCategoryBarRenderer.createImages(): Failed to load image!");
            }

            _opaqueImages = new Image[IMAGE_ASSET_COUNT];
            final int assetsHeight = assetsImage.getHeight();

            for (int i = 0; i < ASSET_START_XS.length; ++i) {
                _opaqueImages[i] = ImageUtils.getSubImage(
                    assetsImage, ASSET_START_XS[i], 0, ASSET_START_XS[i] + ASSET_WIDTHS[i] - 1, assetsHeight - 1);
            }

            _tabWidth = WIDTH / _tabCount;
            _edgeImageWidth = _opaqueImages[SELECTED_TAB_LEFT_CORNER_EDGE].getWidth();
            _selectedTabWidthWithoutEdges = _tabWidth - _edgeImageWidth * 2;

            Image temp = _opaqueImages[TAB_TEXTURE_FULL];
            _opaqueImages[TAB_TEXTURE_FULL] = ImageUtils.scale(temp, _tabWidth, HEIGHT);
```

```
            _opaqueImages[TAB_TEXTURE_WITH_ONE_EDGE] = ImageUtils.scale(temp, _tabWidth - _edgeImageWidth, HEIGHT);
            _opaqueImages[TAB_TEXTURE_WITH_TWO_EDGES] = ImageUtils.scale(temp, _tabWidth - _edgeImageWidth * 2, HEIGHT);
            _opaqueImages[SELECTED_TAB_TEXTURE] = ImageUtils.scale(_opaqueImages[SELECTED_TAB_TEXTURE], _selectedTabWidthWithoutEdge

            _currentImages = _opaqueImages;
        }
```

Image manipulation and transparency

As you can see from the previous code snippet, a class named `ImageUtils` is used to retrieve the parts from the main image and scale them. `ImageUtils` is not something provided by the platform, but it comes with the demo attached to this article - you're welcome to use this in your own projects. Two methods from the class are extracted below; `getSubImage()` creates a new Image instance from a part of the given image and `setAlpha()` makes an image transparent based on the given alpha value:

```
    /**
     * Resolves a sub image from the given image and the given region.
     * @param image The source image.
     * @param x0 The X coordinate of the top-left corner of the sub image.
     * @param y0 The Y coordinate of the top-left corner of the sub image.
     * @param x1 The X coordinate of the bottom-right corner of the sub image.
     * @param y1 The Y coordinate of the bottom-right corner of the sub image.
     * @return A newly created sub image.
     * @throws IllegalArgumentException If the arguments are invalid.
     */
    public static Image getSubImage(final Image image,
                                    final int x0,
                                    final int y0,
                                    final int x1,
                                    final int y1)
        throws IllegalArgumentException
    {
        if (image == null || x0 > x1 || y0 > y1
            || x0 < 0 || x1 >= image.getWidth()
            || y0 < 0 || y1 >= image.getHeight())
        {
            throw new IllegalArgumentException("Arguments received: "
                + image + ", " + x0 + ", " + y0 + ", " + x1 + ", " + y1);
        }

        final int imageWidth = image.getWidth();
        final int[] imagePixels = new int[imageWidth * image.getHeight()];
        image.getRGB(imagePixels, 0, imageWidth, 0, 0, imageWidth, image.getHeight());

        final int subImageWidth = x1 - x0 + 1;
        final int subImageHeight = y1 - y0 + 1;
        final int[] subImagePixels = new int[subImageWidth * subImageHeight];

        for (int x = x0; x <= x1; ++x) {
            for (int y = y0; y <= y1; ++y) {
                subImagePixels[(subImageWidth * (y - y0)) + (x - x0)] = imagePixels[(imageWidth * y) + x];
            }
        }

        return Image.createRGBImage(subImagePixels, subImageWidth, subImageHeight, true);
    }
```

```
    /**
     * Sets the alpha of the each pixel in the image based on the given value.
     * @param image The original image.
     * @param alpha The alpha value.
     * @return A newly created image with applied alpha.
     * @throws NullPointerException If the given image is null.
     * @throws IllegalArgumentException If the alpha value is not in [0, 255].
     */
    public static Image setAlpha(final Image image, final int alpha)
       throws NullPointerException, IllegalArgumentException
    {
        if (image == null) {
            throw new NullPointerException();
        }

        if (alpha < 0 || alpha > 255) {
            throw new IllegalArgumentException();
        }

        final int width = image.getWidth();
        final int height = image.getHeight();
        final int[] originalRgb = new int[width * height];
        image.getRGB(originalRgb, 0, width, 0, 0, width, height);

        final int opaqueRgb[] = new int[width * height];

        for (int x = 0; x < width; ++x) {
            for (int y = 0; y < height; ++y) {
                if (originalRgb[(width * y) + x] >>> 24 == 255) {
                    opaqueRgb[(width * y) + x] = originalRgb[(width * y) + x] + (alpha << 24);
                }
                else {
                    opaqueRgb[(width * y) + x] = originalRgb[(width * y) + x];
                }
            }
        }

        return Image.createRGBImage(opaqueRgb, width, height, true);
    }
```

How do we make our custom category bar transparent? That's easy - we already have our opaque assets extracted so we simply create new translucent assets from them. Just to be on the safe side, let's use threading so that we don't accidentally block the UI thread:

```
        /**
         * Creates the translucent image assets based on the original ones.
         * Note that this is done asynchronously.
         * @param opacity The opacity to use.
         */
        private void createAndTakeInUseTranslucentImages(final float opacity) {
            final int alpha = (int)(255 * opacity);
```

```
            if (_translucentImages == null) {
                _translucentImages = new Image[_opaqueImages.length];
                _translucentIcons = new Image[_unselectedIcons.length];
            }

            new Thread() {
                public void run() {
                    for (int i = 0; i < _opaqueImages.length; ++i) {
                        _translucentImages[i] = ImageUtils.setAlpha(_opaqueImages[i], alpha);
                    }

                    for (int i = 0; i < _unselectedIcons.length; ++i) {
                        _translucentIcons[i] = ImageUtils.setAlpha(_unselectedIcons[i], alpha);
                    }

                    _currentImages = _translucentImages;
                    _currentIcons = _translucentIcons;
                    _storedOpacity = opacity;

                    repaint();
                }
            }.start();
        }
```

As mentioned before, the selected icons are created dynamically in the code. For that, we have the `ImageUtils` class to also thank. Since we know the accent color for selected icons, we can modify the RGB values to change the white color into the accent color:

```
for (int i = 0; i < length; ++i) {
    // Highlight color is 0x29a7cc (RGB: 41, 167, 204)
    selectedIcons[i] = ImageUtils.modifyRgb(unselectedIcons[i], -214, -88, -51);
}
```

```
    /**
     * Modifies the RGB values of the given image while maintaining the alpha
     * level of each pixel. You can subtract by giving negative values or add by
     * giving positive ones.
     * @param image The original image.
     * @param r Red value delta.
     * @param g Green value delta.
     * @param b Blue value delta.
     * @return The modified image.
     * @throws NullPointerException If the given image is null.
     * @throws IllegalArgumentException If any of RGB values is invalid.
     */
    public static Image modifyRgb(final Image image,
                                  final int r,
                                  final int g,
                                  final int b)
       throws NullPointerException, IllegalArgumentException
    {
        if (image == null) {
            throw new NullPointerException();
        }

        if (r < -255 || r > 255 || g < -255 || g > 255 || b < -255 || b > 255) {
            throw new IllegalArgumentException();
        }

        final int[] argbDelta = new int[4];
        argbDelta[0] = 0;
        argbDelta[1] = r;
        argbDelta[2] = g;
        argbDelta[3] = b;

        final int width = image.getWidth();
        final int height = image.getHeight();
        final int[] originalRgb = new int[width * height];
        image.getRGB(originalRgb, 0, width, 0, 0, width, height);

        final int newRgb[] = new int[width * height];
        int pixel = 0;
        int[] argb = new int[4];

        for (int x = 0; x < width; ++x) {
            for (int y = 0; y < height; ++y) {
                pixel = originalRgb[(width * y) + x];
                argb[0] = (pixel & 0xff000000) >>> 24;
                argb[1] = (pixel & 0x00ff0000) >>> 16;
                argb[2] = (pixel & 0x0000ff00) >>> 8;
                argb[3] = pixel & 0x000000ff;

                for (int i = 1; i < argb.length; ++i) {
                    final int newValue = argb[i] + argbDelta[i];

                    if (newValue < 0) {
                        argb[i] = 0;
                    }
                    else if (newValue > 255) {
                        argb[i] = 255;
                    }
                    else {
                        argb[i] = newValue;
                    }
                }

                newRgb[(width * y) + x] =
                    (argb[0] << 24) | (argb[1] << 16) | (argb[2] << 8) | argb[3];
            }
        }

        return Image.createRGBImage(newRgb, width, height, true);
    }
```

Deploying the custom category bar

Setting up the custom category bar is very similar to the steps when using the native component. The exceptions include using the factory method `getNewInstance()` with the custom category bar. The factory method is there just for convenience. The constructors have been made private so that they wouldn't be accidentally used since they might lead to misconstructed instances.

```
public class MyCanvas
    extends Canvas
    implements ElementListener
{
    // Constants
    private static final int TAB_COUNT = 3;
...
    // Members
    private CustomCategoryBar _categoryBar = null;

    /**
     * Constructor.
     */
    public MyCanvas() {
        super();
        ...
        Image[] unselectedIcons = new Image[TAB_COUNT];
        ...
        _categoryBar = CustomCategoryBar.getNewInstance(null, unselectedIcons);
        _categoryBar.setElementListener(this);
        _categoryBar.setVisibility(true);
        _categoryBar.setOpacity(0.5f);
        ...
    }
```

Note: Be careful when playing with the opacity. You can easily ruin the user experience (UX) with too translucent components.

The current implementation supports using the Canvas as parent (just give the Canvas instance to the factory method) and if you do, you don't need to paint the category bar explicitly. However, you get more reliable setup when painting the bar yourself. Make sure that you paint everything else before the category bar so that it stays on top of your stuff. In `MyCanvas.paint()`:

```
    /**
     * @see javax.microedition.lcdui.Canvas#paint(javax.microedition.lcdui.Graphics)
     */
    protected void paint(Graphics graphics) {
        graphics.drawImage(_backgroundImage, -_backgroundImageX, 0, Graphics.TOP | Graphics.LEFT);

        graphics.setColor(0xf4f4f4);
        graphics.setFont(_titleFont);
        graphics.drawString(TAB_TITLES[_categoryBar.getSelectedIndex()], MARGIN, MARGIN,
            Graphics.TOP | Graphics.LEFT);
       ...

        _categoryBar.paint(graphics, getHeight() - CustomCategoryBar.HEIGHT);
    }
```

You have to forward the touch events to the custom category bar since it cannot tap into the events itself:

```
    /**
     * @see javax.microedition.lcdui.Canvas#pointerPressed(int, int)
     */
    protected void pointerPressed(int x, int y) {
        _categoryBar.onPointerPressed(x, y);
        repaint();
    }

    /**
     * @see javax.microedition.lcdui.Canvas#pointerDragged(int, int)
     */
    protected void pointerDragged(int x, int y) {
        _categoryBar.onPointerDragged(x, y);
        repaint();
    }

    /**
     * @see javax.microedition.lcdui.Canvas#pointerReleased(int, int)
     */
    protected void pointerReleased(int x, int y) {
        _categoryBar.onPointerReleased(x, y);
        repaint();
    }
```

The method from the `ElementListener` interface that you have to implement interface with the custom category bar similarly as with the native component:

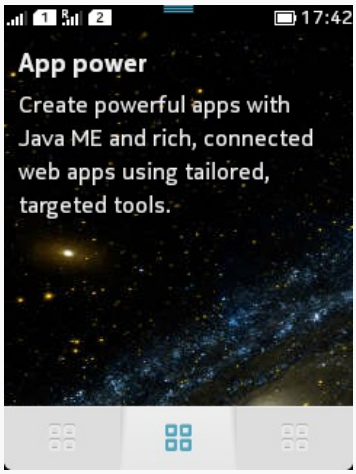
```
    /**
     * @see com.nokia.mid.ui.ElementListener#notifyElementSelected(com.nokia.mid.ui.CategoryBar, int)
     */
    public void notifyElementSelected(CategoryBar bar, int selectedIndex) {
        // Handle the index changed event here
    }
```

CategoryBarDemo

As a result of this case study, an example application was created that provides the custom category bar and sample code how it can be used (and more!). Get the .zip package here: File:CustomCategoryBarDemo-2013-06-27.zip. The demo also implements simple animations which are addressed in another article: Creating Simple Animations with Java ME.

Screenshots

Custom category bar here has the opacity of 0.5.



When pressed the category bar is made opaque. After releasing the touch, the opacity is returned to the previous value.

Known issues

- Only `Canvas` is supported.
- Only tabbed UI solution is supported (i.e. no actions).
- You have to forward the touch events to the custom category bar.
- Using the custom category bar does not have an effect to the value received from `Canvas.getHeight()`. When using the native category bar, the height is reduced by the amount of the height of the bar.
- If you have `Commands` in your `Canvas` and it's not in full screen mode, the command is not hidden in the menu but appears in the bottom of the screen as button. A workaround is presented below.

Workaround for Command usage

Set the `Canvas` in full screen mode and display the status bar by accessing the `Object` trait. This way the `Commands` are placed in the menu which you can open by swiping from the bottom of the screen:

```
setFullScreenMode(true);
LCDUIUtil.setObjectTrait(this, "nokia.ui.canvas.status_zone", Boolean.TRUE);
```

Summary

Canvas-based solutions give you more freedoms in creating custom components. Creating the component itself does not differ much whether you are using a `CustomItem` or `CanvasGraphicsItem` as your base. However, creating the rest of the UI of your application takes more time when using canvas, but if you are willing the invest your time, it is more likely to create better looking UIs with canvas compared to traditional form-based apps.