Creating effects in Java ME

This code example demonstrates how to add a small graphical animation to the `List` item using the MIDP 2.0 API.

Overview

In this example, a small 'running lines' animation is drawn around `List` item images. In order to be able to modify an existing image, it must be 'mutable'. For more information on 'mutable' and 'immutable' images, see the MIDP 2.0 API javadoc.

To get the image held in the `List`, the `List.getImage` method is used. It takes the item index as a parameter. In this example, the last selected item index is passed. Thereby an 'animate on select' function is produced.

In order to start modifying the mutable `Image` object instance, you must get the image's `Graphics`. This can be done by calling the `Image.getGraphics` method.

After a frame has been prepared, you can apply it on the chosen `List` item by calling the `List.set` method and passing the frame-in-image as the parameter.

For more information, see API documentation.

Source file: EffectsThreadObserverIF.java

```
import javax.microedition.lcdui.Image;
/**
 * Interface for object that act as ImageEffectsThread observers.
 */
public interface EffectsThreadObserverIF {
    void imageModified( Image img );
}
```

Source file: ImageEffectsThread.java

```
import javax.microedition.lcdui.Graphics;
import javax.microedition.lcdui.Image;

/**
 * This thread draws four 'running lines' in the image four corners. Eeach line
 * runs parallel to appropriate image side.
 */
public class ImageEffectsThread implements Runnable {

    /**
     * This variable is to true when this thread is running.
     * If the value is set to false thread stops.
     */
    private boolean running;

    /**
     * Object reference that should be notified on each animation iteration.
     */
    private EffectsThreadObserverIF observer;

    /**
     * Reference to immutable image instance which is used in animation when
     * rendering a frame.
     */
    private Image savedImage;

    /**
     * Image buffer that holds the next animation frame.
     */
    private Image modifyImage;

    /**
     * Number of frames in a animation.
     */
    private static final int maxStepCount = 9;

    /**
     * Holds index of next animation frame to be rendered.
     */
    private int step;
    /**
     * Holds amount of pixels wich determines how far from the frame border
     * should the animation be drawn.
     */
    private int offset;
    /**
     * Width of the animation less (2 * offset) value.
     */
    private int newWidth;
    /**
     * Height of the animation less (2 * offset) value.
     */
    private int newHeight;
    /**
     * Vertical 'Running line' length.
```


(C) Copyright Nokia 2013. All rights reserved.

```
     */
    private int hDelta;
    /**
     * Horizontal 'Running line' length.
     */
    private int vDelta;

    public ImageEffectsThread() {
        step = 0;
        running = false;
        observer = null;
        savedImage = null;
        modifyImage = null;
    }

    /**
     * Initilizes both image buffers, calcualtes animation parameters(newWidth,
     * newHeight, ... . See class declaration above.).
     * Set's observer variable to null.
     * Note: the 'offset' value is set to '1' by default.
     * @param restoreImage reference to the image used in animation as a
     * background.
     */
    public ImageEffectsThread( Image restoreImage ) {
        Graphics gr = null;
        savedImage = restoreImage;
        // Creating a copy of image than will be used for animation.
        modifyImage = Image.createImage( restoreImage.getWidth(),
                restoreImage.getHeight() );

        gr = modifyImage.getGraphics();
        gr.drawImage( restoreImage,
                0, 0,
                Graphics.LEFT | Graphics.TOP );
        // Calculating animation parameters.
        offset = 1;
        newWidth = modifyImage.getWidth() - (offset*2);
        newHeight = modifyImage.getHeight() - (offset*2);
        hDelta = newWidth / maxStepCount;
        vDelta = newHeight / maxStepCount;
        running = false;
        observer = null;
    }

    /**
     * Notifies the thread observer that another animation had taken place and
     * passes new frame as a parameter.
     */
    private void repaint() {
        if( observer!= null ) {
            observer.imageModified( modifyImage );
        }
    }

    /**
     * Renders another animation frame and increments the 'step' variable.
     * If 'step' holds value from 'maxStepCount' then 'step' will be set to zero
     * in order to animation to start over again.
     * @param graph holds Graphics for the frame to be rendered.
     */
    private void render( Graphics graph ) {
        if( step == maxStepCount ) {
            step = 0;
        }
        // Clearing the frame.
        graph.drawImage( savedImage, 0, 0, Graphics.LEFT | Graphics.TOP );
        graph.setColor( 0xFFFFFF );
        // Drawing the upper horizontal 'running line'.
        graph.drawLine( offset + (step * hDelta), offset, offset +
                ((step + 1) * hDelta), offset );
        // Drawing the lower horizontal 'running line'.
        graph.drawLine( newWidth - (step * hDelta),
                newHeight,
                newWidth - ((step + 1) * hDelta),
                newHeight );
        // Drawing the right vertical 'running line'.
        graph.drawLine( newWidth, offset + (step * vDelta),
                newWidth, offset + ((step + 1) * vDelta) );
        // Drawing the left vertical 'running line'.
        graph.drawLine( offset,
                newHeight - (step * hDelta),
                offset,
                newHeight - ((step + 1) * hDelta)
                );
        step++;
    }

    /**
     * 'running' value setter.
     * @param runStatus value to set the 'running' variable to.
     */
    synchronized public void setStatus( boolean runStatus ) {
        running = runStatus;
    }

    /**
     * 'running' value getter.
     * @return current 'running' variable value.
     */
    synchronized public boolean getStatus() {
        return running;
    }

    /**
     * Holds main animation cycle. This cycle keeps running until the 'running'
     * variable becomes equal to 'false'. In that case the thread finishes it's
     * execution.
     * If 'running' holds 'true' the the thread executes it's regular
     * animation iteration:
     * - renders next frame;
     * - notifies the thread observer;
     * - waits for 50 milliseconds;
     * - returns to 'running' variable check once again and etc.
     */
    public void run() {
        while( getStatus() ) {
            if( modifyImage != null ) {

                Graphics graph  = modifyImage.getGraphics();
```


(C) Copyright Nokia 2013. All rights reserved.

```
                render( graph );

                repaint();
                try {
                    Thread.sleep( 50 );
                } catch ( InterruptedException e ) {
                    setStatus( false );
                }
            }
        }
    }

    /**
     * Thread observer setter.
     * @param observer
     */
    public void setObserver( EffectsThreadObserverIF observer ) {
        this.observer = observer;
    }

}
```

Extract from Source file: CreatingEffects.java

```
public class CreatingEffects extends MIDlet implements CommandListener,
        EffectsThreadObserverIF {
    private Display display;
    private Form mainForm;

    /**
     * Holds our animation thread implementation.
     */
    private ImageEffectsThread effectsRunnable;

    /**
     * Animation thread object.
     */
    private Thread effectsThread;

    /**
     * Image copy of the currently selected list item.
     */
    private Image image;

    /**
     * List whose items we are going to animate on selection.
     */
    private List imageList;

    /**
     * Command brings mainForm to the foreground and stops animation thread if
     * started.
     */
    private static final Command BACK_COMMAND =
            new Command( "Back", Command.BACK, 0 );
    /**
     * This command is being triggered in case of the imageList item selection.
     * Turns of previously started animation and starts a new one for
     * the selected item.
     */
    private static final Command SELECT_COMMAND =
            new Command( "Select", Command.SCREEN, 1 );
    /**
     * Brings imageList control to the foreground.
     */
    private static final Command EXECUTE_COMMAND =
            new Command( "Start", Command.ITEM, 0 );

    /**
     * Exits the MIDlet.
     */
    private static final Command EXIT_COMMAND =
            new Command( "Exit", Command.EXIT, 0 );

    public CreatingEffects() {
        display = Display.getDisplay( this );
        // Initializing timer ...
        selectedItemIndex = -1;
        image = null;
        effectsRunnable = null;
        effectsThread = null;

        setupMainForm();

        setupImageList();
    }

    /**
     * Inializes imageList control. Loads images from files, makes them mutable
     * and adds them to the imageList as item images.
     */
    private void setupImageList() {
        Image tempImage = null;
        Image tempMutableImage = null;
        imageList = new List( "Images", Choice.IMPLICIT );

        imageList.setSelectCommand( SELECT_COMMAND );
        imageList.addCommand( BACK_COMMAND );
        imageList.setCommandListener( this );

         printString( "Loading list images ..." );
        for( int i = 0; i < 3; i++ ) {
             printString( "Loading image " + i );
            try {
                tempImage = Image.createImage( "/item" + i + "_image.PNG" );
                tempMutableImage = Image.createImage( tempImage.getWidth(),
                        tempImage.getHeight() );
                Graphics g = tempMutableImage.getGraphics();
                g.drawImage( tempImage, 0, 0, Graphics.TOP | Graphics.LEFT );
            } catch ( IOException e ) {
                 printString( e.toString() );
                return;
            }
            imageList.append( "Item " + i, tempMutableImage );
        }
```


(C) Copyright Nokia 2013. All rights reserved.

```
        imageList.setFitPolicy( Choice.TEXT_WRAP_ON );
    }
    /**
     * From CommandListener.
     * Called by the system to indicate that a command has been invoked on a
     * particular displayable.
     * @param command the command that was invoked
     * @param displayable the displayable where the command was invoked
     */
    public void commandAction(Command command, Displayable displayable) {
        if (command == EXIT_COMMAND) {
            // Exit the MIDlet
            exit();
        } else if ( command == EXECUTE_COMMAND ) {
            display.setCurrent( imageList );
        } else if ( command == SELECT_COMMAND ) {

            int selItem = imageList.getSelectedIndex();
            if ( selItem < 0 )
                return;
            printString( "Selecting item " + selItem  );
            if ( selectedItemIndex >= 0 ) {
                printString( "Stopping current animation ..." );
                // Stopping effects timer.
                while( effectsRunnable.getStatus() ) {
                    // Waiting for thread to stop.
                    effectsRunnable.setStatus( false );
                }
                restoreImage();
            }
            printString( "Getting currently selected image ..." );
            selectedItemIndex = selItem;
            imageList.setSelectedIndex( selectedItemIndex, true );
             printString( "Saving currently selected image ..." );
            image = imageList.getImage( selectedItemIndex );
             printString( "Starting effects thread ..." );

            //  start animation
            effectsRunnable = new ImageEffectsThread( image );
            effectsRunnable.setObserver( this );
            effectsRunnable.setStatus( true );
            effectsThread = new Thread( effectsRunnable );

            effectsThread.start();

            printString( "Animation is running ..." );

        } else if ( command == BACK_COMMAND ) {
            if( selectedItemIndex >= 0 ) {
                 printString( "Stopping current animation ..." );
                // Stopping effects timer.
                while( effectsRunnable.getStatus() ) {
                    // Waiting for thread to stop.
                    effectsRunnable.setStatus( false );
                }
                restoreImage();
                // set currently selected item index to -1
                selectedItemIndex = -1;
            }
            display.setCurrent( mainForm );
        }
    }

    /**
     * Restores previously selected list item image to the 'non-animated' one.
     * Used when turning off the animation.
     */
    private void restoreImage() {
        if( image != null ) {
            //  replace animated image with static one
            setItemImage( image );
        }
    }

    /**
     * From EffectsThreadObserverIF.
     * Calls the setItemImage method with value passed in 'img' parameter.
     * @see EffectsThreadObserverIF.
     * @param img
     */
    public void imageModified(Image img) {
        if ( img != null ) {
            setItemImage( img );
        }
    }

    /**
     * Replaces the selected item image with the one held in the 'img' parameter.
     */
    synchronized private void setItemImage( Image img ) {
        // Committing changes to the list item.
        synchronized ( imageList ) {
            int sel = imageList.getSelectedIndex();
            imageList.set( selectedItemIndex,
                imageList.getString( selectedItemIndex ),
                img );
            imageList.setSelectedIndex( sel, true );
        }
    }
}
```

Postconditions

When the MIDlet is started, the main form with a text field is displayed. Press the 'Start' softkey to display the list with images.

To animate a list item image, move the cursor to the list item and press the 'Select' softkey.

Supplementary material

This code snippet is part of the stub concept, which means that it has been patched on top of a template application in order to be more useful for

(C) Copyright Nokia 2013. All rights reserved.

developers. The version of the Java ME stub application used as a template in this snippet is v1.1.

- The patched, executable application that can be used to test the features described in this snippet is available for download at Media:CreatingEffects.zip.
- You can view all the changes that are required to implement the above-mentioned features. The changes are provided in unified diff and colour-coded diff (HTML) formats in Media:CreatingEffects.diff.zip.
- For general information on applying the patch, see Using Diffs.
- For unpatched stub applications, see Example app stubs with logging framework.

COPY-PASTE the desired platforms into the wiki text and save. All relevant version categories should be applied.
Windows Phone: [[Category:Windows Phone]]
[[Category:Windows Phone 7.5]]
[[Category:Windows Phone 8]]

Nokia Asha: [[Category:Nokia Asha]]
[[Category:Nokia Asha Platform 1.0]]

Series 40: [[Category:Series 40]]
[[Category:Series 40 1st Edition]] [[Category:Series 40 2nd Edition]]
[[Category:Series 40 3rd Edition (initial release)]] [[Category:Series 40 3rd Edition FP1]] [[Category:Series 40 3rd Edition FP2]]
[[Category:Series 40 5th Edition (initial release)]] [[Category:Series 40 5th Edition FP1]]
[[Category:Series 40 6th Edition (initial release)]] [[Category:Series 40 6th Edition FP1]] [[Category:Series 40 Developer Platform 1.0]] [[Category:Series 40 Developer Platform 1.1]] [[Category:Series 40 Developer Platform 2.0]]

Symbian: [[Category:Symbian]]
[[Category:S60 1st Edition]] [[Category:S60 2nd Edition (initial release)]] [[Category:S60 2nd Edition FP1]] [[Category:S60 2nd Edition FP2]]
[[Category:S60 2nd Edition FP3]]
[[Category:S60 3rd Edition (initial release)]] [[Category:S60 3rd Edition FP1]] [[Category:S60 3rd Edition FP2]]
[[Category:S60 5th Edition]]
[[Category:Symbian^3]] [[Category:Symbian Anna]] [[Category:Nokia Belle]]

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.