Graphics scalability approaches for Series 40 Java ME apps

This article lists methods for scaling app graphics to support different screen resolutions and orientations, and links to the Series 40 Java ME examples which utilise those techniques.

20 Jan 2013

Introduction

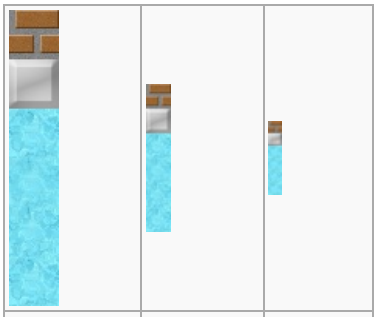
Series 40 devices come in a number of different display resolutions, and may also support both portrait and landscape modes. This can pose compatibility, performance and resource problems for app developers unless handled with care. Luckily there are ways to overcome that complexity effectively, while still providing an optimised experience for each resolution and orientation.

This article evaluates (from image quality, performance and when to use point of view) a number of techniques for graphics scalability and points to the Series 40 Java ME examples utilising those techniques. Typically a combination of approaches is used; the best approach depends on the type of app in question.

The approaches evaluated include; different graphics assets for different resolutions, runtime scaling of images, drawing graphics with code, filling in spare space, using large background and utilizing vector graphics.

Different graphics assets based on the resolution

Scalability can be achieved by creating different graphics assets for all supported Series 40 resolutions, and then loading the appropriate set in runtime based on the resolution of the device the application is running on. The image below shows an example of different tile images of BattleTank game example for three different resolutions:



32 x 192 px	16 x 96 px	8 x 48 px

If the images are not highly detailed, it is usually sufficient to create a single set for the highest resolution and then just scale the images smaller for lower resolutions. For detailed images it may be necessary to create a different set for the lowest resolution. There are no absolute guidelines for this; the best way is to simply see how the scaled-down images look on lower resolution devices.

When implementing this approach, it is good to encapsulate the assets so that each set is in its own folder, e.g. "`/res/high/`", "`res/med/`" and "`/res/low/`", and have matching file names for the images regardless of the resolution. Another option would be to have a prefix or postfix in the file names to describe the resolution of the set, e.g. "`/res/tile-high.png`" and "`/res/tile-med.png`". In the BattleTank example the aforementioned approach is used.

In the code the images are loaded runtime based on the width and height of the screen. First, find out what the screen resolution is. For instance, GameCanvas returns its size with `getWidth()` and `getHeight()`. You can use the values returned to determine which set you want to load the images from:

```
public static final int MEDIUM_THRESHOLD = 320;
public static final int HIGH_THRESHOLD = 640;

private String resourcePath;
public final int gridSizeInPixels;

/**
 * Constructor.
 */
public Resources(int width, int height) {
```

```
    final int max = Math.max(width, height);

    // Check what is the size of the resources to be loaded
    if (max < MEDIUM_THRESHOLD) {
        resourcePath = "/low/";
        gridSizeInPixels = 4;
    }
    else if (max < HIGH_THRESHOLD) {
        resourcePath = "/medium/";
        gridSizeInPixels = 8;
    }
    else {
        resourcePath = "/high/";
        gridSizeInPixels = 16;
    }
    ...
```

Using different graphics assets is fast since it does not involve complex algorithms, but on the downside it increases the size of the .jar file. You can, of course, do the selection at build time to optimise the jar size and create separate binaries for each resolution, but that is a compromise — instead of a larger .jar file your maintenance burden is heavier.

Runtime scaling

By runtime image scaling we refer to scaling an image resource to a different size when the application is running with an algorithm implemented in the application code. This approach is widely used and is suitable for most cases. In general, the runtime scaling is recommended for applications with a reasonable number of graphics assets, and in cases where the size of the images is not in your control, e.g. when loading assets from the web.

When feasible, it is recommended to do the scaling during the start-up. Users are used to the applications taking some time starting up, and you can display a splash screen and possibly some kind of a progress indicator like a progress bar. Note, however, that your app needs to start in a reasonable time. If the scaling of the assets takes a long time, you should consider an alternate approach, or implement the scaling so that it is done in the background.

Some of the algorithms are quite complex, especially if you are not familiar with this area. Luckily, several algorithms can be found online and furthermore, there's a Nokia Developer example providing three different algorithms for Series 40: see Image Scaling example. The algorithms included in the example are linear interpolation, bilinear interpolation and pixel mixing. The table below demonstrates the quality of the scaled images:

Size	Linear interpolation	Bilinear interpolation	Pixel mixing
70 %			

160 %			

As is usual with algorithms, there's a price to pay for quality and that is time. A comparison between the algorithms was made and you can see the results here. The summary of the comparison is presented in the table below.

	Linear interpolation	Bilinear interpolation	Pixel mixing
Quality	Poor.	Great. Beats Pixel mixing when scaling images bigger.	Great. Beats bilinear interpolation in most cases when scaling images smaller. However, anti-aliasing problems may be visible but this depends on the qualities of the original image.
Performance	Excellent.	Poor.	Good.
When to use	When performance is the key. This could be used as the first phase before applying a better algorithm. The aforementioned method may improve the user experience, i.e. a scaled image is shown right away, while better scaling is done in the background to follow.	When scaling images larger and quality is the key.	The most suitable algorithm in most cases since both quality and performance are great.

Drawing graphics with code

Drawing simple graphics, like basic shapes and gradients, is recommended to be done with code rather than with images. This is because the methods of Graphics class are well optimised, and the result usually provides better performance than using image resources.

Drawing basic shapes is very easy and seldom requires more than few lines of code. Unless you need anti-aliasing there's really no reason not to implement them with code. Drawing gradient is somewhat more complex, but fortunately there's this great wiki article which covers the case.

Other techniques

Sometimes you may have a need to keep the aspect ratio of your graphics, for instance, if you have a fixed size game level that fits on the screen. In these cases you might end up with excess area on the screen real-estate. To get the best out of this unfortunate situation, you can **fill the spare area with either a background colour or background image**. You could also consider utilising the space otherwise if feasible. Both Frozen Bubble and Sudokumaster examples use this technique.

In some games, for instance those belonging to real-time strategy genre, **the levels are bigger than what fits on the screen**, and the viewport is often scrollable. If the resolution of the targeted devices does not vary too much, you could manage with only one set of graphics. Thus, the level of zoom might only look different on different resolutions. For example, in Racer game the track is a single image, but only part of it is shown on the display.

Finally, an obvious technique of scaling graphics is using **vector graphics**. Images in **SVG format** are supported by Series 40 and they are scalable by default if created properly. Java Developer's Library contains simple example MIDlets demonstrating the use of Scalable 2D Vector Graphics API (M2G API). You can even use SVGs in localising your app!

Examples

The following table shows the example and the corresponding graphics scalability techniques used:

Example (and link to the project)	Techniques used (and links to project wiki pages)
BattleTank	▪ Different graphics sets for different screens and orientations ▪ Tile based UI, full UI constructed by repeatedly utilising smaller graphical elements
Explonoid	▪ Scaling Runtime (start-up time) scaling of graphics ▪ Filling edges with background colour if different screen aspect ratio
Frozen Bubble	▪ Runtime (start-up time) scaling of graphics. Both pixel mixing (for downscaling) and bilinear interpolation (for upscaling). ▪ Filling edges with background colour if different screen aspect ratio
Image Scaling example	▪ An example focused purely on threaded runtime image scaling featuring three different scaling algorithms
Picasa Viewer	▪ Runtime scaling of graphics
Sudokumaster	▪ Different graphics sets for different screens and orientations ▪ Runtime (start-up time) scaling of graphics ▪ Tile based graphics, full UI constructed by repeatedly utilising smaller graphical elements ▪ Filling edges with background colour See how: http://github.com/nokia-developer/sudokumaster-jme/wiki#Handlingdifferentresolutions
Tic-Tac-Toe for Series 40	▪ Runtime scaling of graphics
WeatherApp	▪ Different graphics sets for different screens and orientations

Summary

Remember that a combination of the techniques may be more powerful than any of them individually. There is no silver bullet in graphics scalability, but with careful planning you can achieve the best result regardless of the type of the application you are developing. The table below summarises the use cases for each technique and their pros and cons.

Approach	When to be used	Pros	Cons
Different graphics assets	When targeting many different resolutions and where the difference between the resolutions is high.	No runtime penalty. Guaranteed compatibility.	The size of the .jar file is increased unless the different sets are managed under different builds. This naturally adds complexity to managing the project. Either way the maintenance burden is increased. Depending on the case there may be a need for creating assets individually for very different resolutions.
Runtime scaling	With reasonable amount of graphical assets.	No increase in `.jar` file size. Guaranteed compatibility with targeted devices, and very good dynamical adaptation.	Depending on the number of images to scale, the runtime penalty may be high or you need to spend more time implementing threaded solution for scaling the assets. Loss of quality may occur.
Drawing with code	With simple shapes and gradients.	Smaller `.jar` file size. Better performance compared to approach using images. Usually simple to implement. Possible to dynamically adapt to device theme colours etc.	Complex items take longer to implement.

Filling spare space	When aspect ratio of, for example, a game level needs to be fixed.	Easy to implement.	May result in poorer user experience.
Large backgrounds	Usually with games that have a large level that is not meant to fit to the viewport.	Dynamic. No extra work with little resolution differences.	Memory consumption may be high, especially if the background is a single image instead of a tile map. When the resolution difference is high, individual graphics assets or scaling still required. Performance could be an issue, and time used in optimisation may be increased.
Vector graphics	For instance custom item graphics in utility applications.	When done right, supports any resolution of a device family. Future proof. No maintenance burden.	Not as easy to create as the usual image assets.

Testing your app

The emulator serves well when you test how your application looks on various resolutions. The performance evaluation with emulator, however, is approximate at best, and you should not trust the results you get. Few developers have all the different devices at hand, but luckily Nokia offers a free service, Remote device access, for you to test your app with different phones.

Version Hint

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]]