

# **Supporting Different Resolutions on Mobile Phones**

## **Recommendations for Mobile Widget Developers**



Vodafone Internet Discovery Services

# Table of Contents

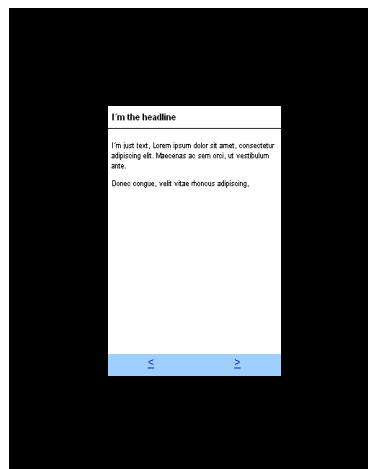
1. Introduction and Scope .....	1
1.1. Terminology .....	2
1.2. Philosophies on creating scalable / resolution aware widgets .....	3
2. Screensize and the widget size .....	5
3. Resolution (ppi) .....	6
3.1. Calculating ppi in theory .....	6
3.2. The difference between theory and practice – or – physical and logical inches .....	7
3.3. Calculating the ppi in practice .....	7
3.4. Effects of different resolutions on widgets .....	8
3.4.1. Fixed sized elements and images .....	8
3.4.2. Readable content .....	8
3.4.3. Touchable elements .....	9
3.5. General techniques to deal with different ppi .....	9
3.5.1. CSS – relative length units and percentage .....	9
3.5.2. CSS – absolute length units .....	11
3.5.3. CSS Media Queries .....	12
3.5.4. Javascript .....	13
3.6. Font size specific techniques .....	14
3.6.1. CSS font specific keywords .....	14
3.6.2. Media queries for handling font sizes .....	15
3.7. Graphics/Images specific techniques .....	16
4. Creating scalable widgets .....	18
4.1. The Newsreader example .....	18
4.1.1. First implementation .....	18
4.1.2. Enhancements with media queries .....	23
4.1.3. Enhancements with SVG .....	25
5. Document History .....	27

# 1. Introduction and Scope

Nowadays, developers of any kind of screen-based application – whether it is a web site, an application for a mobile device, or a desktop application – have to keep in mind that different screens have different properties and features. Beside the obvious differences – form factor and physical size, for example –, the biggest differences are the various screen resolutions and, as a consequence thereof, the different DPI (Dots Per Inch) – or to be precise ppi (Pixels Per Inch) – values in dependence of the real physical dimensions of the viewable area.

A higher ppi is usually seen as a good thing, because it leads to sharper and clearer fonts, images, and so on. However, when display size stays (roughly) the same while the resolution increases, things will be displayed smaller. Consider a bitmap with the size of 200x100 pixels. If you view that image on a screen that has 100ppi, the image will be displayed two inches wide and one inch high. If you view that same image on a screen that has 300ppi, the image will be displayed half an inch wide and one third of an inch high. To prevent this, the developer has to make sure that font sizes, image sizes and so on will scale depending of the ppi of different devices.

If a widget does not handle different resolutions, it most probably will look like the following one:



The widget does not use the full screen, the fonts are too small to be read easily and the navigation is unusable on touch displays because of the small size.

This document deals with the different screen sizes and ppi values on mobile devices that use the Vodafone Apps Manager, which is the runtime that actually launches, displays and controls the widget. In comparison to the previous version of this document, we extended the section about the ppi and added the sections about different resolutions and the effects on widgets, the general techniques and the step by step example.

We start with an introduction on the term “resolution” and discuss different techniques to create scaling widgets. The next chapter covers the screen size and the widget size from a widget developer's perspective. The subsequent chapter provides an in

depth discussion about different resolutions on mobile devices, how they affect widgets and what possibilities exist for developers to handle them. We finish with a step by step example that shows how to use the techniques we have explained.

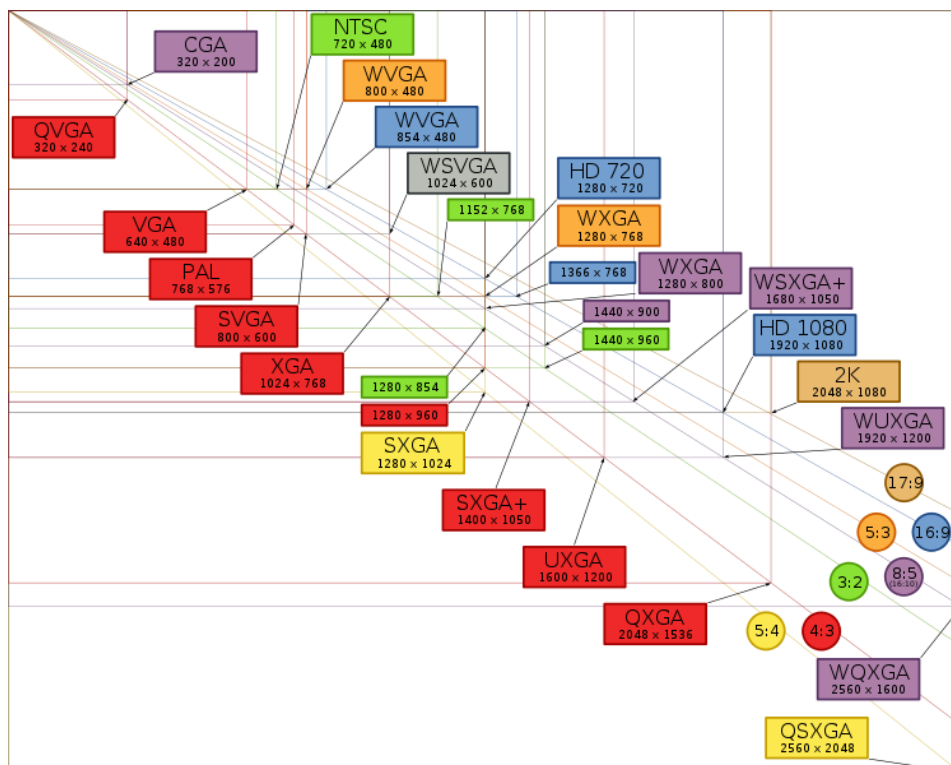
## 1.1. Terminology

The term “resolution” has different meanings, depending on how it is used. It’s especially important to understand the differences and what we mean by the term resolution in this document.

The “**screen resolution**” is defined as the number of pixels that can be displayed in horizontal and vertical dimension. A screen resolution is typically given in the notation:

“horizontal pixel count”x“vertical pixel count”, for example: 800x600

The commonly used screen resolutions were given names, for example “SVGA” for 800x600. The following picture gives an overview of common screen resolutions (not limited to mobile devices, see also Appendix A) used today:



Common screen resolutions. License: Creative Commons Attribution-ShareAlike 3.0, Source: [http://en.wikipedia.org/wiki/Display\\_resolution](http://en.wikipedia.org/wiki/Display_resolution)

Note that the screen resolution is defined in pixels only and does not define the physical dimensions of the display (expressed in length units like mm or inch, something you could measure with your ruler on the device). Hardware manufactures deliver displays with the same screen resolution in different dimensions. For example, a 240x320 screen

can have a diameter of 2 inches or 2.6 inches. This leads to a different “**pixel density**” on the displays, which is expressed as “**pixels per inch**” (ppi). The ppi is also called the “resolution” of a display and this **is what we mean by resolution** in this document.

In our example, the resolution for a screen with 240x320 pixels and a diameter of 2 inches would be 196 ppi, while the resolution for a screen with 240x320 pixels and a diameter of 2.6 inches would be 153ppi. Details for calculations can be found in section 3.1.

## 1.2. Philosophies on creating scalable / resolution aware widgets

Within this document we will present certain techniques that can be used to build scalable widgets. What does “scalable” mean in this context?

Scalable means that the **widget adapts to the device specific characteristics** to give the user the best experience possible on the specific device. How exactly device specific characteristics are used **depends highly on the widgets use-case** and how a designer would like the widget to work on different platforms.

Measures used to deal with different resolutions in one widget might be the wrong ones for a different widget or use-case. Example: Consider you have an RSS reader widget and a game widget. You will most likely want the RSS reader to fill the whole screen of your mobile device in order to show as much content as possible. But the game widget needs a specific aspect ratio (a square) for example. Using the technique described in Chapter 2 to resize the widget to fullscreen, is perfect for the RSS reader, but inappropriate for the game widget.

Moreover, there are two general philosophies how to create scaling widgets. You can create a widget that scales depending on the device characteristics either seamlessly or in defined steps. We’ll call the former scaling to be “**linear**”, the latter one to be “**profile based**”. Which of these you should or can implement is use-case dependent.

In our example above, the RSS reader is a good candidate to be linear scalable, i.e. by making the font appear 2mm regardless of the ppi of the device.

For the game widget it would be better to have two sets of graphics: one with low resolution graphics, the other one with high resolution graphics. Moreover, two font sizes (12px and 16px) can be defined to be used in conjunction with these graphic sets. If the resolution is higher than a defined value, say 200ppi, you want the high resolution pictures to be used and a font size of 16px, otherwise you want the other pictures set and a font size of 12px. That way, you create two “profiles” for your widget. You could extend this with another step, say 150ppi and offer three sets of graphics and further font size, resulting in three profiles: 0 - 150, 150 - 200 and 200 - unlimited ppi.

In practice, there are use-cases where you will find yourself mixing up the linear and profile based approach. For example, you may want the font size to scale linear, but have different profiles for the widgets menus and so on.

Please note that some of the techniques that you will see, like defining font sizes in mm instead of pixels, may be strange to you at first. This is probably due to the fact that we, as developers, have defined font sizes in pixels for years and that's what we know that works and that is what we like to use. Developing an application for current mobile devices with different screen sizes (every one of them still quite limited in space) and the different resolutions (differing much stronger than PCs resolutions from each other) require other approaches.

## 2. Screensize and the widget size

The developer of a mobile widget faces a couple of problems when it comes to the point that their widget should look its best on every mobile phone. One of these problems is the size of the widget itself on-screen. The size of the widget is specified in the `config.xml`-file of the widget. Let's assume that you have set the width and height of the widget to 240x320 pixels. Now the widget is displayed fullscreen on a Nokia N95 (240x320 pixels, about 154ppi), but will only fill about one third of the screen of a Nokia 5800 XpressMusic (360x640 pixels, about 229ppi).

Fortunately, **the height and width values from the `config.xml`-file are only used for initialization of the widget and may be altered programmatically at runtime.** To make sure that your widget is always displayed at fullscreen, you can use a simple javascript function.

```
function resizeWidget() {
    var availWidth = screen.availWidth;
    var availHeight = screen.availHeight;

    window.resizeTo(availWidth, availHeight);
}
```

During the execution of a widget, the resolution of the phone can change. For example, newer mobile phones have sensors to detect whether the phone is held horizontally or vertically. The Vodafone Apps Manager recognizes such changes and fires a `resolution`-event every time there is a change. To make sure your widget is resized every time the resolution changes, you can add an event listener for the `resolution`-event and use the `resizeWidget()` function as event handler:

```
widget.addEventListener('resolution', resizeWidget, false);
```

Additionally, you should call the `resizeWidget`-function when the widget starts, because a user could start the widget in landscape mode. **As the runtime does not raise a `resolution`-event when this happens you have to call the `resizeWidget`-function manually.**

Furthermore, you should check whether your widget is running on the desktop. On the desktop, you do not want to resize your widget to fullscreen, so you do not want to call the `resizeWidget`-function. To make sure, that the widget is only resized when it is not running on a desktop, you should add the following to your code:

```
if (!(widget.widgetMode === "widget")) {
    resizeWidget();
    widget.addEventListener('resolution', resizeWidget, false);
}
```

## 3. Resolution (ppi)

A screen is made up of pixels that are horizontally and vertically aligned. The resolution of the screen is given in pixels per inch (ppi), which basically means how many pixels fit on one inch. On devices with a high ppi, an element with its dimensions defined in pixels (i.e. 100x100 pixel) looks smaller than on a device with a lower resolution.

### 3.1. Calculating ppi in theory

**Note:** You do not need to know the formula to calculate the ppi of a mobile phone to make use of the ppi value in your widgets. This chapter is purely informative and you can skip it if you are not interested in the technical details on how to calculate the ppi of a mobile phone.

Calculating the ppi of a device is fairly easy and can be done in two steps<sup>1</sup>: First, calculate the diagonal resolution in pixels. Second, divide the result from the first step by the diagonal size in inch of the screen.

Here is an example to illustrate: A Nokia N95 has a 2.6" QVGA screen with 240x320 pixels.

w = width resolution in pixel  
h = height resolution in pixel  
 $d_p$  = diagonal resolution in pixel  
 $d_i$  = diagonal size in inches

First, one has to calculate the diagonal resolution in pixels:

$$d_p = \sqrt{w^2 + h^2} = \sqrt{240^2 + 320^2} = 400$$

Second, we divide the result of the first step by the diagonal size in inches of the phone's screen:

$$PPI = \frac{d_p}{d_i} = \frac{400}{2.6} = 153.846154$$

So we see that the Nokia N95 has about 154 pixels per inch.

In general, all phones running on Symbian S60 v3 have 240x320 pixels; all phones running on Symbian S60 v5 have 640x360 pixels.

---

1 [http://en.wikipedia.org/wiki/Pixels\\_per\\_inch](http://en.wikipedia.org/wiki/Pixels_per_inch)

## 3.2. The difference between theory and practice – or – physical and logical inches

Unfortunately, we can not implement the formula mentioned in the previous paragraph. This is solely due to the fact, that as a developer, you don't have generic access to the **physical** dimensions of the display.

Fortunately, a widget runtime allows you to define height and width of elements by using inch as unit. How does that work? Basically, it's an assumption and an approximation: The *system* defines a number of pixels it believes are shown as an inch on the display (this inch is the **logical** inch). The difference between a logical and physical inch is the interesting part. If we take a look at regular PCs, the ppi is by default set to 96. This is mainly because of exchangeable monitors, which can have different physical ppis the operating system does not know about. Therefore, the operating system just makes this assumption, which can result in quite different real inches on the screen.

As mobile devices have fixed screens, it is possible for the runtime to define one fixed ppi value that best fits into one real inch on the screen. Testing this on different mobile platforms shows that the logical and the physical inch match is so good that the difference can be neglected. This means to you as a developer that you have the possibility to define, for example, elements with 1 inch edge length, and can be sure that it will be one real inch high and one real inch wide on each mobile device as well.

For further reading on this sometimes difficult to grasp topic, [www.emdpi.com](http://www.emdpi.com) is recommended.

## 3.3. Calculating the ppi in practice

As mentioned in 3.2, the widget runtime allows you to create elements with dimensions defined in inches. The following JavaScript function creates an element of one logical inch width and reads the number of pixels that were used by the runtime to create the element.

```
function getPPIOfCurrentDevice() {
    var DOM_body = document.getElementsByTagName('body')[0];

    var DOM_div = document.createElement('div');
    DOM_div.style.width = "1in";
    DOM_div.style.visibility = "hidden";

    DOM_body.appendChild(DOM_div);
    var ppi = document.defaultView.getComputedStyle(DOM_div,
    null).getPropertyValue('width');
    DOM_body.removeChild(DOM_div);
}
```

```
        return parseInt(ppi);  
    }
```

With this function, you can easily get the ppi of the current device and process it as needed within javascript.

## 3.4. Effects of different resolutions on widgets

The fact that hardware platforms vary greatly in the number of pixels they can display per inch on a screen makes it hard for developers to create a widget that looks its best on any platform. For example, a widget with its dimensions defined in pixels that looks great on the desktop (approximately 96ppi) will look very small on a mobile phone with 640x360 pixels and a 3.5" screen (around 210ppi).

In the next section we will take a look at different visual elements a widget can contain and how they are affected by different resolutions.

### 3.4.1. Fixed sized elements and images

Usually, widgets provide some kind of general layout elements like menu buttons, differently styled and positioned content areas and so on. As said in the introduction of this chapter, elements with dimensions defined in pixels will look very/too small on high resolution display.

Another problem for mobile widget developers that makes it hard to let the widget look optimal on different mobile phones concerns the images used in a widget. The most widespread way of using images while developing for the world wide web is by including bitmap graphics, e.g. .jpg, .png, .gif and so on. Unfortunately, because of the strong variation in ppi on mobile phones, with mobile widgets the situation is slightly more complicated. An image that looks perfect on a Nokia N95 (QVGA, 240x320 pixels, about 154ppi) will probably look too small on a Nokia N97 (360x640 pixels, about 210ppi). And an image that looks perfect on a Nokia N97 might look fuzzy on a Nokia N95. This is due to the fact that bitmap graphics are not scalable. You could resize them, but that usually results in blurred images – most probably this is unacceptable.

### 3.4.2. Readable content

This not only applies to elements with dimensions defined in pixels, but is also true for font sizes defined in pixels. They look smaller on high resolution displays than on low resolution displays. Recent displays offer a very high resolution which makes fonts defined e.g. with 12px (which is common on the world wide web) very hard to read.

A question that often arises: When is a font readable? Sadly, there is no set of rules you can adhere to and be sure, that your font is readable. Even on classic print media,

this question is influenced by subjective factors, for example the age of the reader. Nevertheless, there is some rule of thumb: **a font is readable if it's size is at least 2mm, or 0.07 inches respectively.**

### 3.4.3. Touchable elements

As recent mobile phones offer a touch screen as the only mean to navigate, another problem arises: On high resolution devices, the touchable elements become too small to be touched comfortably. As with the font size, there also arises a similar question: When is an area touchable? Again, a lot of different factors are involved, like the thickness of the finger or the physical characteristics of the touch pad, which make this question hard to be answered in general. But as a rule of thumb: **a square area with an edge length of at least 7mm, or 0.27inches, is touchable.**

## 3.5. General techniques to deal with different ppi

Widget developers can use different techniques to deal with the ppi. The following sections describe some general technical possibilities that are available to you as a widget developer to deal with different ppis. **It is especially important knowing when to use which techniques and even more important, when not to use them.**

### 3.5.1. CSS – relative length units and percentage

Use-cases:

- size elements according to the font size
- size elements relative to their parent elements

The W3C CSS 2.1 specification (<http://www.w3.org/TR/CSS2/>) makes a distinction between relative and absolute length units. This section covers the relative **em** unit while chapter 3.5.2 deals with absolute length units.

“Relative” means that the size is calculated using a reference size. The em unit is defined as follows: “The 'em' unit is equal to the computed value of the 'font-size' property of the element on which it is used.” So the em unit uses the font size as a reference size.

For example, you want to create a notification box. It should be three times as high as the font-size and the message should be displayed in the middle. The HTML is simple:

```
<body>
  <div class="relative_sized_box">
    <span>My notification text</span>
  </div>
```

```
</body>
```

The CSS is more interesting:

```
body {
  font-size: 12px;
}

.relative_sized_box span {
  display: block;
  height: 1em;
  line-height: 1em;
  border: 1px solid red;
  padding: 1em;
}
```

The body's font size is 12px, which will be used as reference. For the notification box, we define the span-element to be as high as the font size (12px). We add a padding with the size of the font size. For the top and bottom we get two times the font size (24px), which results in a total height of 36px. The addition of a one pixel width red border makes it look nice.

Changing the font size in the body would result in a differently sized notification box. By wrapping this notification box within another element that has already a defined font-size, the referenced font-size will change to the one of the outer element.

Another possibility to define a size in relation to another size is by using a **percentage** value. You may want the notification box be half of the screen width and have it appear centred. Moreover, the font size should be twice as large as the regular (referenced) one. We enhance the CSS declaration with:

```
.relative_sized_box {
  font-size: 200%;
  width: 50%;
  margin-left: 25%;
}
```

The width property references the width of the parent object (which is the body tag and, in our case, the whole widget width) and makes the notification box half of that. To make it appear centred, we adjust the notification box by defining a left margin of 25%. This way, we get 25% free space on the left and right and the box takes exactly 50% of the whole width.

The percentage for the font size property uses the font size defined in the parent object, the body, as the reference value. 200% doubles that value, resulting to a font size of 24px. The em values defined previously now reference this value to make their calculations, which leads to a total height of 72px (plus border) for the notification box.

This example shows that when using percentages it's especially important to know which reference sizes are used for the calculations. Using em or a percentage value allows you to create widgets that adapt dynamically to given values, the scaling is **linear**. For further reading on this topic, see <http://www.w3.org/TR/CSS2/>.

### Excursion: Pixel as a relative unit

As said, the CSS specification makes a distinction between relative and absolute units. The px unit is classified as a relative unit in the specification. At first glance, this may seem strange. But the explanation is easy: dimensions defined in pixels are relative to the resolution of the viewing device.

## 3.5.2. CSS – absolute length units

Use-cases:

- Elements should be displayed with the same dimensions on each device (for example touchable areas) – regardless of the ppi and screen dimensions.
- Fonts should be displayed with the same size on each device – regardless of the ppi and screen dimensions.

**Not** a use-case:

- **pixel defined layouts**
- **elements and fonts used in space restricted, pixel defined environments (like text in fixed sized buttons, menu bars, ...)**
- define the whole widget size / fullscreen (see Chapter 2)
- elements that should take a relative space (for example the rest/half of the screen)

As mentioned in 3.2, the widget runtime allows you to define the dimensions of an element in inch. The definition is done via CSS. Example:

```
<body>
  <div>text</div>
</body>
```

Now let's define the div to be 1 inch high and 2 inches wide. Here is the CSS code for it:

```
div {
  height: 1in;
  width: 2in;
}
```

Let's go through how the widget runtime interprets and renders this example. First,

the widget runtime looks at the unit specified in the style definition. In our example it's inch. Now, the runtime multiplies these values with the ppi (it knows) and receives a length in pixels, which are then rendered. On different resolutions you get, of course, a different number of pixels.

Exactly the same happens if you define the dimensions of font sizes or images in inch. The units are calculated to pixels which are used to render. Example:

```
div {  
    font-size: 0.2in;  
}
```

A typical use-case for such a definition is the continuous text of a news ticker widget. You can ensure the same font size on every resolution, the scaling is **linear**.

Besides inch, CSS allows you to define the dimension using the following absolute units:

- **cm**: Centimeter
- **mm**: Millimeter
- **pt**: Point, defined as 1/72nd of an inch
- **pc**: Picas, defined as 12 points

### 3.5.3. CSS Media Queries

Use-cases:

- depending on the resolution, you want other styles to be applied to your widget
- defining style profiles depends on device characteristics

Media queries are a powerful mechanism to adapt your widget to the device/media characteristics. In general, media queries allow you to do much more, but we'll focus on it's use of the ppi. Let's start with an example:

```
/* style.css */  
@media all and (min-resolution: 200dpi) {  
    /* all definitions are only for minimum 200ppi devices */  
    body { font-size: 24px; }  
}
```

In this example, we define a font size of 24px for the body tag for all devices with a ppi of at least 200. While parsing the CSS definitions, the widget runtime uses the dpi value (we use dpi and ppi interchangeable here) given in the media query and compares it with it's ppi value (it knows). Only if the resolution of the device is at least 200ppi, the style definitions within the curly brackets will be applied, otherwise they won't.

As you can encapsulate any CSS definition in such a media query, you are not limited to pixel units as in our example. You could also have used inch or cm units as described in 3.5.2. In 3.6.2 you will see extended examples of how to use media queries to manage the font size.

There is one more interesting way to use media queries. Instead of declaring the media specific rules directly within the curly brackets, you could put them in a separate file and make this file imported with a media query. Example:

```
/* style.css */
@import url("200dpi.css") all and (min-resolution: 200dpi);

/* 200dpi.css */
body { font-size: 16px; }
```

For testing purposes, you can directly include the different style definitions and see how they affect your layout.

With media queries, you define styles for ranges of ppi values, so the scaling is **profile based**. For further reading on media queries, we recommend <http://www.w3.org/TR/css3-mediaqueries/>.

### 3.5.4. Javascript

Use-cases:

- general purpose ppi handling, especially things you can not do with CSS or media queries

In 3.3, we showed how to calculate the ppi with Javascript. As a developer, you know that you can use Javascript to access the DOM and manipulate its elements. That said, it's obvious that you can also manipulate the DOM depending on the ppi. Let's do rewrite the example in 3.5.3:

```
var ppi = getPPIOfCurrentDevice(); // defined earlier
var DOM_body = document.getElementsByTagName('body')[0];
if (ppi >= 200){
    DOM_body.style.fontSize = '16px';
}
```

Even if this self-explanatory example does the same thing you could have achieved by using media queries, Javascript allows you to do much more than just dealing with the element styles. For example, based on the ppi you could show different texts or shuffle your elements into another order – things that are not possible with CSS.

```
var ppi = getPPIOfCurrentDevice(); // defined earlier
```

```
var DOM_body = document.getElementsByTagName('body')[0];
DOM_body.innerHTML = 'Your device has PPI value of '+ppi;
```

Using Javascript, you may use the ppi value as you like, so you can implement a **linear scaling** as well as a **profile based** one.

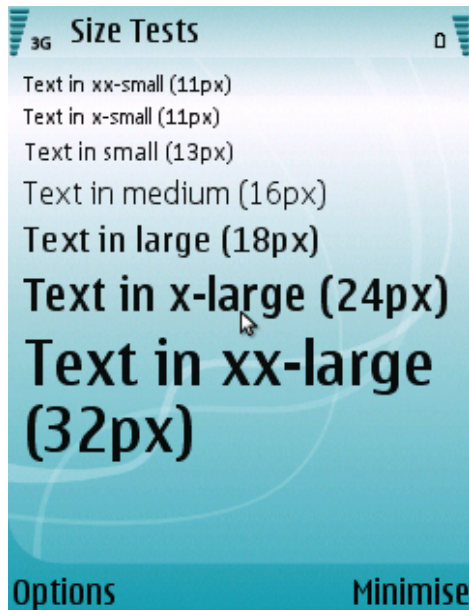
## 3.6. Font size specific techniques

The Vodafone Apps Manager's default font size for the <body>-tag of a widget's index.html-file is 16px. Just like in all other HTML/CSS environments, the developer can change the default font size by altering the <body>'s CSS font-size attribute. We will show how to change this and how to define relative values to it, which is useful for headlines etc.

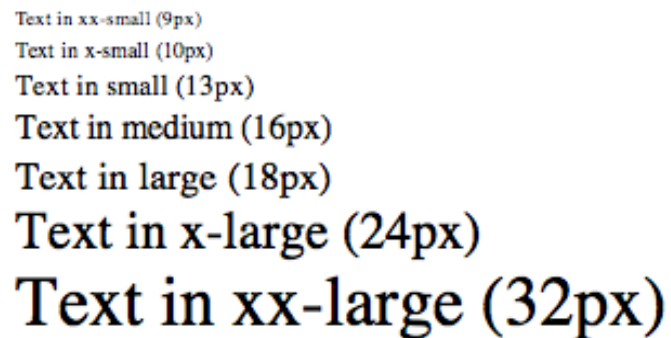
### 3.6.1. CSS font specific keywords

Within CSS, there are seven fixed keywords to manipulate font sizes: xx-small, x-small, small, medium, large, x-large, and xx-large. Within these seven keywords, the medium keyword functions as a reference. All other font sizes are derived from the size of the medium font size by either in- or decreasing it. The size for the medium font size itself is derived from the user's browser settings. It can be altered by the developer by setting the font size for the <body>-tag using CSS.

Screenshot 1 shows that the minimal font size within the Vodafone Apps Manager is 11px. Comparing font sizes within a widget and on the desktop (using Opera Browser for Desktop, see Screenshot 2), xx-small renders fonts with 11px (on desktop: 9px) and x-small renders fonts with 11px, too (on desktop: 10px).



Screenshot 1: Nokia N95



Screenshot 2: Desktop

In general, you could use CSS keywords for managing font sizes. Unfortunately, you may not know how different keywords are interpreted by the runtime. It could be that certain keywords are not recognized by a runtime or that they are treated differently, just as you can see from the two screenshots above, where xx-small and the x-small keyword are being treated differently.

### 3.6.2. Media queries for handling font sizes

A developer can use a media query to determine at what resolution the current device is running. Depending on the resolution, the developer creates any number of layout profiles in which the font sizes are defined as fixed values in pixels. Using this approach, the developer can be sure that their fonts will have exactly the needed height in pixels.

A more advanced approach is to use a media query to determine at what resolution the current device is running, but in contrast to the previous approach, the developer does not define all font sizes in pixel values, but only the font-size of the <body>-tag. All other font sizes are then defined by **relative** values – like percentage (%), em-values or the keywords mentioned in the previous section – to this font size. The <body>-tag's font size is taken as a reference for all other font sizes.

Here is an example:

```
<body>
  <div>
    <p>This is some sample text</p>
  </div>
```

```
</body>
```

Having this short HTML section, we could use the following CSS definitions to handle font sizes for different devices:

```
body { font-size: 12px; }

@media all and (min-resolution: 200dpi) {
  body { font-size: 16px; }
}

div { font-size: 100%; }
p { font-size: 80%; }
```

A major advantage over the first approach is that in the future, when new devices and/or other resolutions will be available, developers would just have to add another media query and alter the font size for the `<body>`-tag. All other font-sizes would then be altered automatically, taking the `<body>`-tag's font size as reference. This would allow the developer to adjust their widget with the least amount of work.

### 3.7. Graphics/Images specific techniques

Similar to first described approach in section 3.6.2, a developer can use a media query to determine at what resolution the current device is running. Depending on the resolution, the developer uses a different set of graphics. Using this approach, the developer can be sure that the images will look sharp on every device. Of course, this approach requires more work because the developer has to create different sets of the same graphics. Needless to say, it also takes much more time and effort to maintain such a widget.

How to overcome the effort to create different graphic sets? As the runtime allows you to explicitly size the graphics, media queries could be used to size the images according to the ppi. A first approach would be, to resize your bitmap graphics:

```
<body>
  
</body>
```

A CSS could look like this:

```
img.icon { width: 95%; } /* This can be any value you want */
```

You could, of course, have used pixels to define the icon size. The important thing to remember is that this will probably lead to blurry or distorted images as bitmap based

graphics do not scale without a quality loss.

A more advanced approach is by using Scalable Vector Graphics (SVG). SVGs are the perfect tool for resolution independent graphics, because they can be resized to whatever size the developer wants to without looking blurred or distorted in some other way.

Using SVGs in combination with media queries is fairly easy. First, the developer uses a media query to determine at what resolution the current device is running. After that, the SVGs may be resized using the width and height-attributes.

Here is an example:

```
<body>
  <object class="icon" data="images/mySVG.svg"
  type="image/svg+xml"></object>
</body>
```

Having this short HTML section, the developer could use the following CSS definitions to handle image sizes for different devices:

```
object.icon { width: 16px; height: 16px; }

@media all and (min-resolution: 200dpi) {
  object.icon { width: 32px; height: 32px; }
}
```

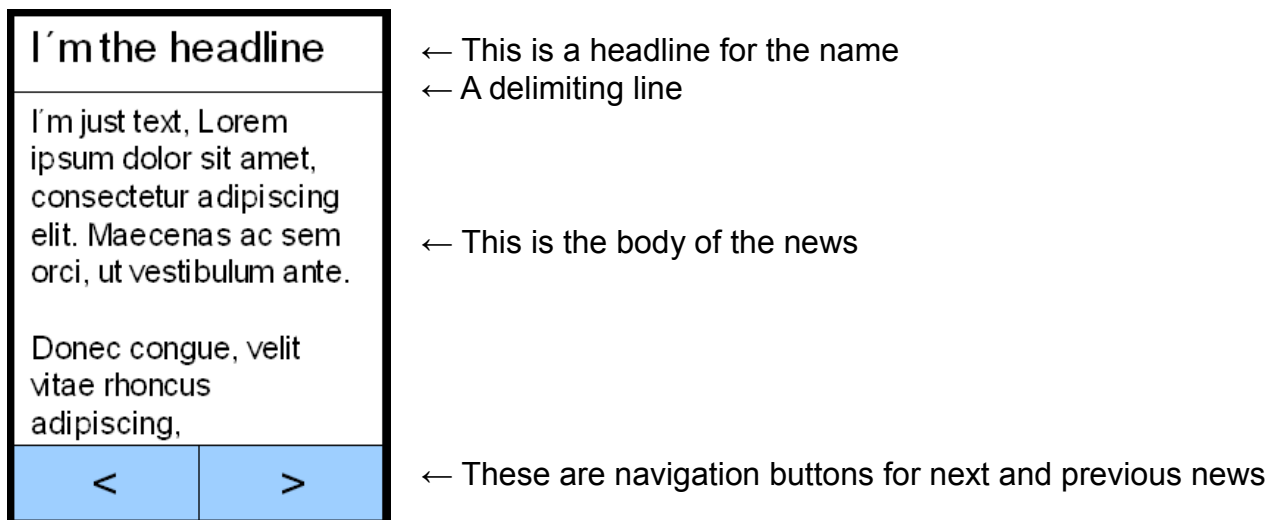
Just like when managing font sizes, this approach has a major advantage: In the future, when new devices and/or other resolutions will be available, developers would just have to add another media query and alter the width- and height-attributes for the SVGs.

## 4. Creating scalable widgets

The previous two chapters described how the screen size and the resolution can affect the look and feel of your widget and what technical possibilities exist to make your widgets look optimal on various platforms. Let's see how you can create scalable widgets.

### 4.1. The Newsreader example

Let's say you would need to implement the following wireframe of a newsreader widget that should run on touch devices with any resolution:



Screenshot 3: Newsreader wireframe

The layout consists of three parts: Headline, body and navigation. Headlines can take multiple lines and should appear in a bigger font size than the body. A small line is used as a visual delimiter between the headline and the news body, The body itself should have a font size that is easily readable. The navigation buttons will be touched by the user to navigate to the previous or next news.

#### 4.1.1. First implementation

Let's have a look at the important parts of the basic HTML and CSS code for such an application.

HTML:

```
<body>
  <div id="news_headline">I'm the headline</div>
```

```
<div id="news_body">
  <p>I´m just text, Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Maecenas ac sem orci, ut vestibulum ante.</p>
  <p>Donec congue, velit vitae rhoncus adipiscing,</p>
</div>
<div id="news_navigation">
  <a id="button_previous" class="navigation_button"
href="#">&lt;</a>
  <a id="button_next" class="navigation_button" href="#">&gt;
  </a>
</div>
</body>
```

This is quite straight forward. We create 3 DIV elements within the body, add some example content and use anchor tags for the navigation.

CSS:

```
html, body {
  background-color: #FFFFFF;
  margin: 0;
  font-family: sans-serif;
}

#news_headline {
  font-size: 2.8mm;
  font-weight: bold;
  padding: 0.5em;
  border-bottom: 1px solid #000000;
}

#news_body {
  font-size: 2.2mm;
  padding-bottom: 7mm;
  padding-left: 0.5em;
  padding-top: 0.5em;
  padding-right: 0.5em;
}

#news_navigation {
  position: fixed;
  bottom: 0px;
  height: 7mm;
  width: 100%;
}

.navigation_button {
  width: 50%;
```

```
    height: 100%;  
    float: left;  
    background-color: #99CCFF;  
    text-align: center;  
}
```

**Basics:** In `html/body` we set the basic font family and background color for the widget. We also remove any margins set by `html` or `body`, so we have the complete space for our widget.

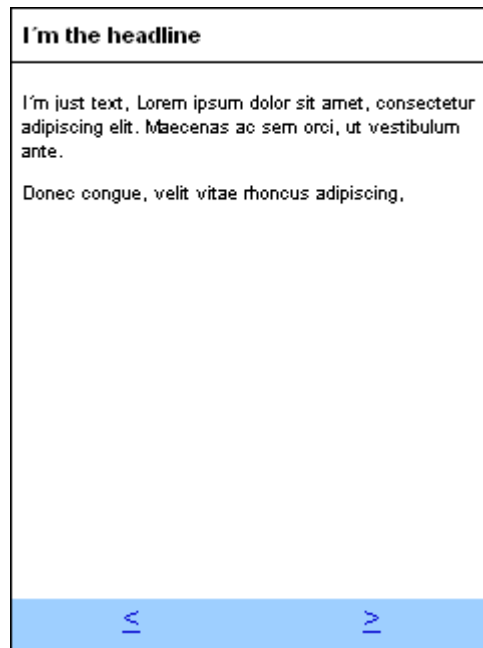
**Header:** The font size is given in mm, we do **NOT** use pixels here, that way we ensure that it is displayed in the same height, even on high resolution displays. We set the font size to bold, add some padding in em and add a border. This border is defined in pixels, which is okay as we want it to be displayed as fine as possible. We do **NOT** define a height. That way, this element can grow in length, which is needed for headers that have multiple lines.

**Body:** The font size is defined in mm and no height is defined for the same reasons as in the header. The paddings are also defined in em, like in the header. The exception is the bottom padding. On the bottom, we need to make sure that there is enough place for the navigation. That's why we use the same padding here as the height of the navigation.

**Navigation:** We position the navigation to be always on the bottom. Additionally, we want it to be 7mm high so it's **high enough to be touchable** on any resolution. As with the font size, using pixels would result in different sizes on different resolutions.

The buttons have a simple styling too: We want them to take half of the whole widget's width each. That's why we use a percentage of 50 here. The height is given as 100%, which is 7mm as the `news_navigation` element is the used parent object. We float the buttons to the left in order to let them appear next to each other, give them a background color and make the contained text centred.

Let's have a look what we have got so far:

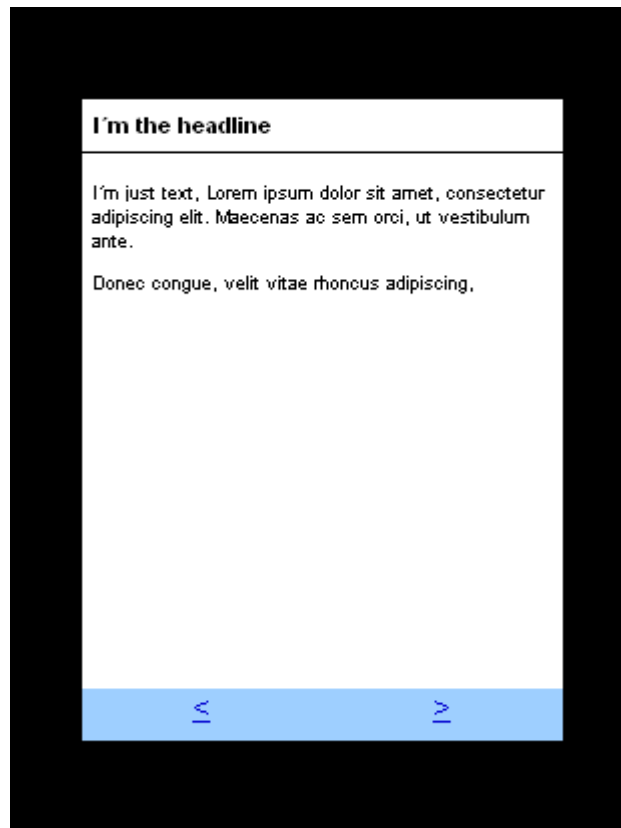


Screenshot 4: Newsreader, first implementation

It looks good so far, but we haven't looked at the config.xml file:

```
<?xml version="1.0" encoding="utf-8"?>
<widget id="http://localhost/newsreader_demo">
  <widgetname>News reader</widgetname>
  <description>News reader demo</description>
  <width>240</width>
  <height>320</height>
</widget>
```

We define the widget height and size in pixels here. Look how it would look like on devices with a larger screen:



Screenshot 5: Newsreader, first implementation on larger screen

We remember from Chapter 2 that the widgets height and width defined in the config.xml can be overwritten at runtime. Let's add the Javascript to resize the widget to fullscreen on start-up and a change in orientation:

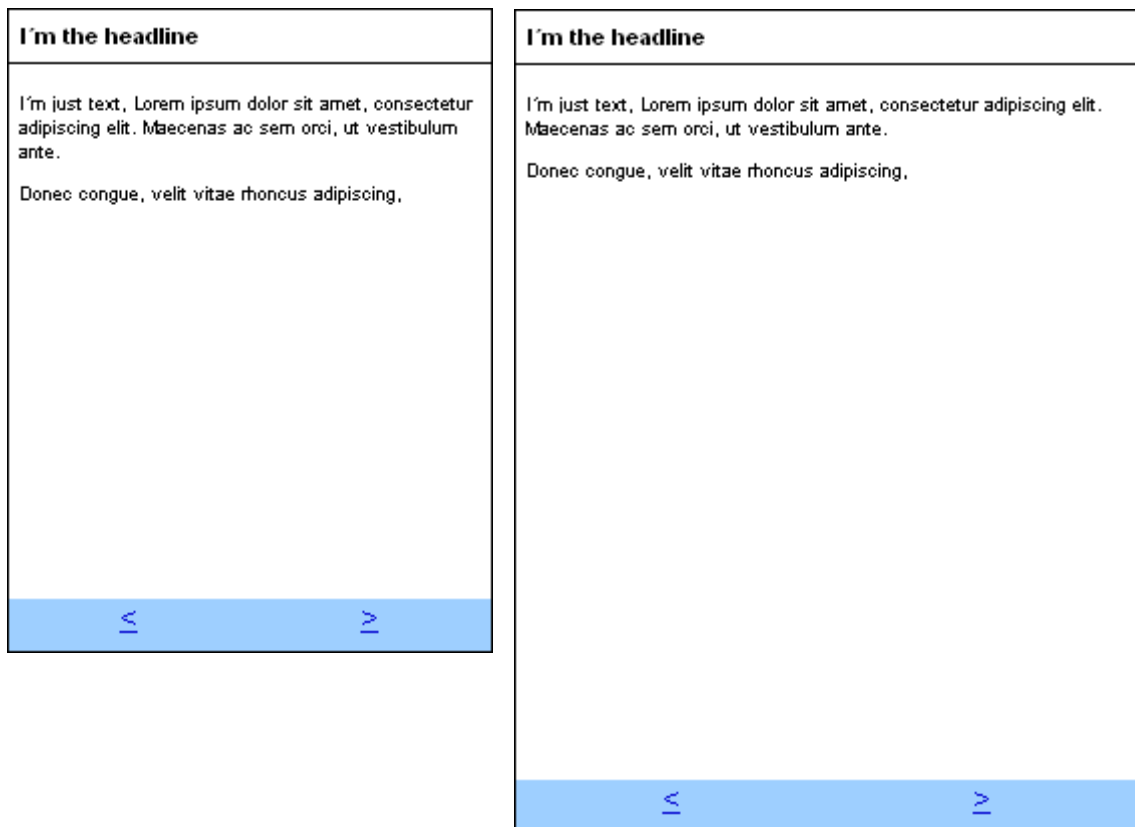
```
function resizeWidget(){
    var availWidth = screen.availWidth;
    var availHeight = screen.availHeight;
    window.resizeTo(availWidth, availHeight);
}

function init(){
    widget.addEventListener('resolution', resizeWidget, false);
    resizeWidget();
}
```

We need to register the init method to be called when the document has been loaded:

```
<body onload="init();">
```

Let's make a comparison of this widget on different screen sizes and resolutions:



Screenshot 6: Newsreader, comparison on different screen size

Note the following:

- The font size is of equal height on both
- The navigation buttons expand and scale to the full width
- The navigation buttons are of equal height and are easy to touch
- No space is left unused (no black areas)

### 4.1.2. Enhancements with media queries

Let's consider the following requirement: If there is more than 5cm space in width, add two buttons to be able to jump to the first and to the last entry. These two buttons are not crucial to the functionality as the same could be done by using the previous and next buttons, so it's okay to hide them on smaller displays.

First, we need to enhance the HTML for this:

```
<div id="news_navigation">
  <a id="button_first" class="navigation_button"
href="#">&lt;&lt;</a>
  <a id="button_previous" class="navigation_button"
```

```

href="#">&lt;</a>
  <a id="button_next" class="navigation_button" href="#">&gt;</a>
  <a id="button_last" class="navigation_button"
href="#">&gt;&gt;</a>
</div>

```

Again, this is straight forward. The tricky part is adapting the widget depending on the screen width. Basically, there are two things we need to do:

- hide two of the buttons if the width is smaller than 5cm
- make all buttons have a width of 25% if the width is greater or equals 5cm

Actually, this one is quite easy by using media queries. Let's enhance our CSS (added to the bottom of the file) with:

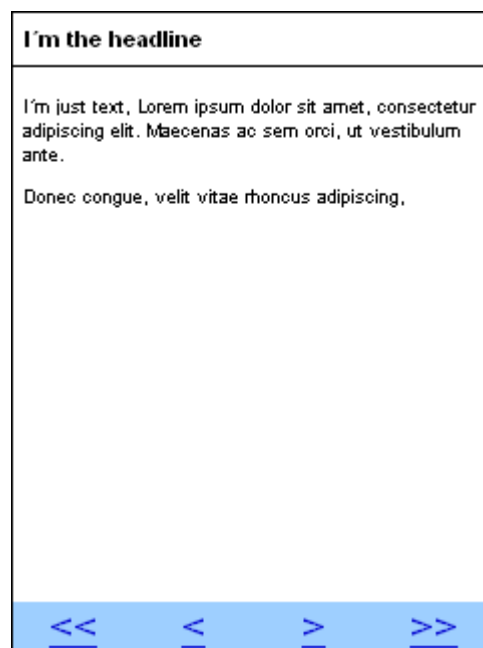
```

@media all and (max-width: 50mm) {
  #button_first, #button_last {
    display: none;
  }
}

@media all and (min-width: 50mm) {
  .navigation_button {
    width: 25%;
  }
}

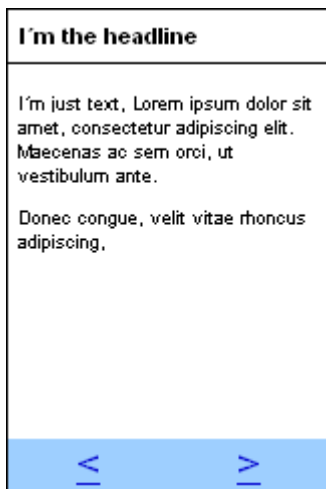
```

Result:

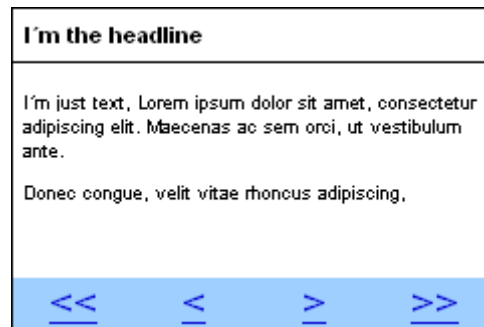


Screenshot 7: Newsreader with 4 buttons

Trying the same widget on a smaller display in both horizontal and vertical mode shows the effect of the code even better:



Screenshot 9: Newsreader, 4 buttons, small display, vertical



Screenshot 8: Newsreader, 4 buttons, small display, horizontal

### 4.1.3. Enhancements with SVG

One thing that could look nicer are the arrows on the buttons. We created them by just using greater-than (“>”) and smaller-than (“<”) as the text. Let’s use some images here, to enhance the look of the widget.

As we have a well defined and fixed height of 7mm, we could use gif, jpg or png images here. But to be more flexible - probably the 7mm will be changed to 14mm one day - we’ll directly use svg images here. Within the HTML for the navigation, we exchange the &gt; etc. with svg object tags:

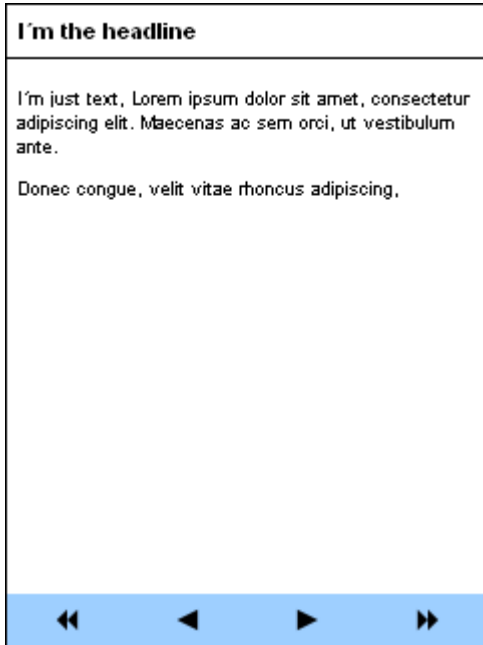
```
<div id="news_navigation">
  <a id="button_first" class="navigation_button" href="#">
<object data="images/arrow_left_double.svg" type="image/svg+xml">
</object>
  </a>
  <!-- and so on -->
</div>
```

In the CSS, we define that these object tags should use the whole available space:

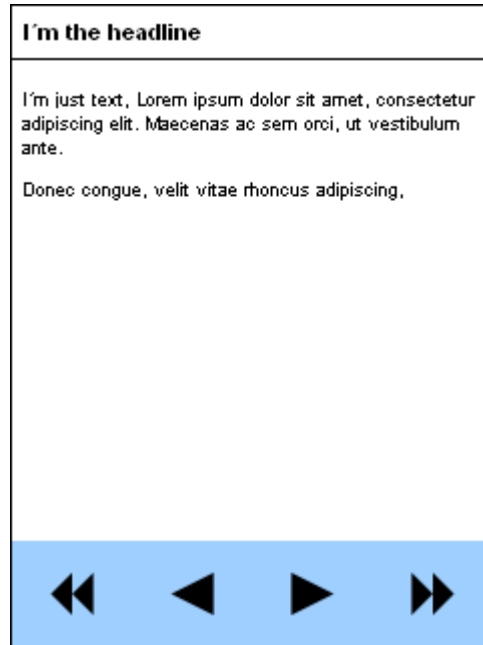
```
.navigation_button object {
  height: 100%;
  width: 100%;
```

}

For every button, we have a specific svg file. The 7mm and 14mm version will look like this:



Screenshot 10: Newsreader, SVG buttons, 7mm



Screenshot 11: Newsreader, SVG buttons, 14mm

## 5. Document History

Revision	Description	Date	Author(s)
0.1	Initial Draft	05.06.09	Stefan Kolb
0.2	Included suggestions made by Daniel Herzog & Jochen Cichon	10.06.09	Stefan Kolb
0.3	Added introduction, ppi information, ppi implications, general techniques part, step by step example	31.08.09	S.B.
0.3.1	Switched to "images/svg+xml", supported by all newer browsers. Added CSS defining height and width for svg images in the newsreader example. Added Appendix A, table with common mobile resolutions.	07.09.09	S.B.
0.3.2	Updated common mobile resolutions table	07.09.09	S.B.
0.3.3	Updated Javascript PPI calculation function to be crossplatform compatible.	14.09.09	S.B.
0.3.4	Updated common mobile resolutions table	15.09.09	S.B.

## Appendix A: Common mobile screen resolutions

<b>Resolution</b>	<b>Physical Size</b>	<b>PPI (rounded)</b>	<b>Example Devices</b>
240x320	2.0-2.8in	200-143	S60v3 (Nokia N95)
240x400	3.2in	146	Vodafone 360 M1 by Samsung
320x480	3.2in	180	Android (HTC Magic)
360x640	3.2-3.5in	229-210	S60v5 (Nokia N97)
480x800	3.5in	267	Vodafone 360 H1 by Samsung