How to design apps on Tizen Wearable devices with multi-resolution

Webapp UI layout guide

Table of Contents

Overview

This document demonstrates how to design your application layout so that your application can run on multiple Tizen devices. It is targeted for 3rd party developers.

The Web engine provides the basic mechanisms for fitting user content to the target screen. However, to ensure that the application runs well on multiple devices, you must understand those mechanisms in detail.

A relative layout is a safer choice than a fixed layout, because a relative layout helps to maintain usability even when the application runs on unexpected devices. To enhance usability even further, use different layouts for each device category.

1. Introduction

Although the Web was originally designed to structure documents for large-scaled Internet world, with CSS (Cascading Style Sheet), JavaScript, and various device APIs, it is quickly evolving to a complete runtime environment for applications. In addition, its W3C open development model and the flexibility of its CSS-based presentation make it every developer's choice as the next development platform.

CSS is the key technology for flexible presentation, and it is used to describe the look and formatting of HTML documents. The role of CSS is to determine how the logical structure of the document is displayed to the user.

This document explains how, using some CSS techniques and Tizen Web engine mechanisms, you can design your Web application UI layout so that it displays properly on multiple devices. The document content includes:

- Explanation of the auto-fitting mechanism of the Tizen Web engine. The document provides practical tips for using the mechanism to create a single layout that covers multiple devices.
- Description and sample code for creating different layouts for each device category.
- Summary of the useful tips.

2. Single Layout for Multiple Devices

A typical application developer designs an optimized UI for a target device with a fixed layout and high-quality resources. This approach usually results in a nice look for the end user, but can sometimes show an ugly layout, which makes the application unusable. And this unfortunate result can occur merely due to a change in the device form factor.

Since the presentation and business logic are strictly separated in a Web application, it handles the device form factor change better than native applications. In this section, you learn how just one layout, taking advantage of Web technology, can ensure usability even on multiple devices with different form factors.

Viewport and Other Meta Tags

The viewport is a screen area that the Web engine displays in the UI and, in the Web world, the viewport meta tag is used to inform the Web engine that this content is written for a specific form factor, such as device width. The viewport meta tag was introduced by Apple to fill the screen resolution gap between initial Smartphone (for example, 320 px) and PC (for example, 980 px). Basically, the viewport meta tag helps the Web engine to determine the scale factor for the content on the current device.

To use the tag in an HTML file, set its name and content:

```
<meta name=”viewport” content=”XXX”>
```

As shown in the following code snippet, use the viewport meta tag to tell the Web engine which device width is targeted by the application. The Web engine can estimate the scale factor based on the target content size and the real screen width. For example, usually almost all mobile applications have following viewport meta tag:

```
<meta name=”viewport” content=”width=device-width,user-scalable=no”>
```

`which sets the viewport width to the appropriate size which depends on each device.` if you want to set your content layout to the target width of 320 px on every mobile devices, use the following meta tag:

```
<meta name=”viewport” content=”width=320”>
```

The viewport meta tag has other properties, such as height, initial-scale, minimum-scale, maximum-scale, and user-scalable. However, do not use those properties unless you understand their exact meaning. Wrong values can cause the Web engine to incorrectly fit the user content to the current device.

Relative Layout

The relative layout concept means that the element size of all content is set as a relative unit (such as percentage), and not as absolute values (such as pixels or points). In responsive Web design, this concept is also known as “fluid grid” (for more information, google “responsive web design”). To achieve a layout that works on multiple devices, you only need to know about relative layout, not all the other concepts of responsive Web design.

Implementing a relative layout is easy. You only have to set the width and height of each element as a percentage, as shown in the following example. In the example (shown in Figure 1), the width and height of the number_pad element is set to 100% and 70%, not 320 px and 224 px.

```
number_pad { width: 100%; height: 70%; }
```

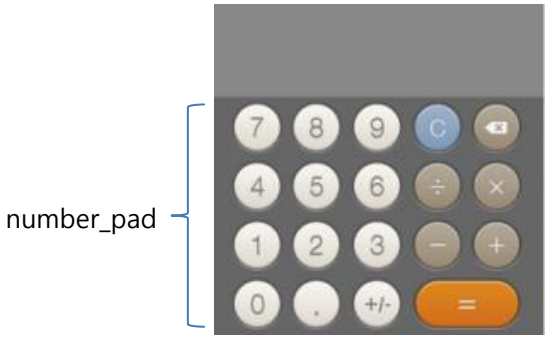

Figure 1. 320x320 calculator

An application with this relative layout is auto-fitted when it runs on an unexpected device, such as the device with 360x480 resolutions shown in Figure 2. Even though this adjusted layout may not be as good as the original, it is good enough, since it shows a usable application UI.

Figure 2. 360x480 calculator with a relative layout

Figure 3 shows what happens if you use an absolute 320x320 layout. The presentation is definitely ugly (with an empty white space at the bottom) and usability reduced, although the calculation logic itself still works.

```
number_pad { width: 320px; height: 224px; }
```

Figure 3. 360x480 calculator with an absolute 320x320 layout

Header and Footer Position

While the content layout itself becomes the application UI layout on a Web page, a typical mobile application layout consists of a header area, content area, and footer area. With this trend, many mobile Web applications are designed by separating the three areas explicitly, as shown in Figure 4.

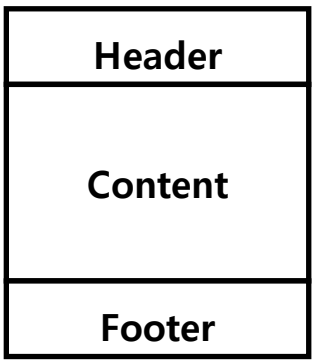

Figure 4. Typical mobile Web application layout

The most common mistake you can make is not to set the place of the header and footer areas clearly. In case of the header, the side-effect is relatively small. However, a wrongly defined footer area can be quite visible and lead to poor usability. Figure 5 shows the original layout of a pedometer application that consists of a header, content, and footer, with a Stop button set in the footer area.

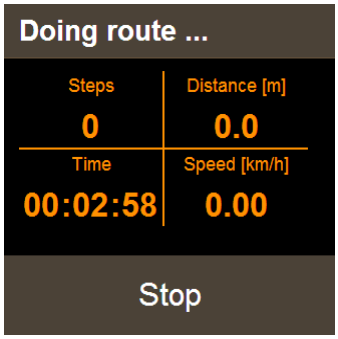

Figure 5. 320x320 sample Web application

In this case, if the position of the footer is not defined explicitly or if the position and bottom properties do not have proper values, an ugly layout can be displayed on an unexpected device, such as the 360x480 screen shown in Figure 6.

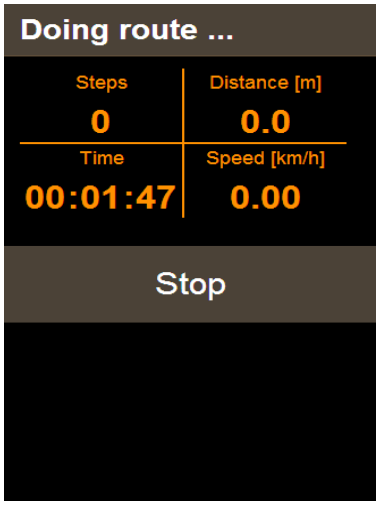

Figure 6. Layout on 360x480

To make the layout correct, define the position property as fixed and set the bottom property explicitly as 0 px:

```
.footer { position: fixed; bottom: 0px; }
```

Figure 7 shows the same Web application running on the unexpected device after the footer properties are defined properly. With the correct values, the Web application is usable even on a new target, not planned during the development phase. Note that if a relative layout is still applied to the content area, the result is an even better layout and further improved usability.

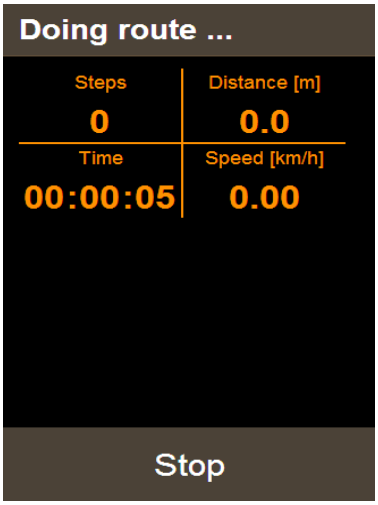

Figure 7. Usable layout on 360x480

Flexible Media

One of key elements used in the content area is a media element, including image and video. As all other elements, the media elements also must be placed within a relative layout to allow the application flexibly handle resolution changes. When such relative sizing is applied to a media element, it is called “flexible media” in responsive Web design.

The main issue when making media element size relative is to ensure that the usable ratio does not change. Figure 8 shows a video element that works on both 320x320 and 360x480 resolutions, even though the aspect ratio of the screen is changed and scaled up. The opposite is shown in Figure 9, where usability decreases as the video element is simply scaled up based on the aspect ratio change.

Basically, when handling media elements, you want to keep the size or ratio of the element the same as in the initially targeted device, even when displayed using a different resolution and aspect ratio (as shown in Figure 8). To achieve this, set the width and height properties of the media element with percentage and the auto keyword.

```
.video iframe { width: 100%; height: auto; }
```

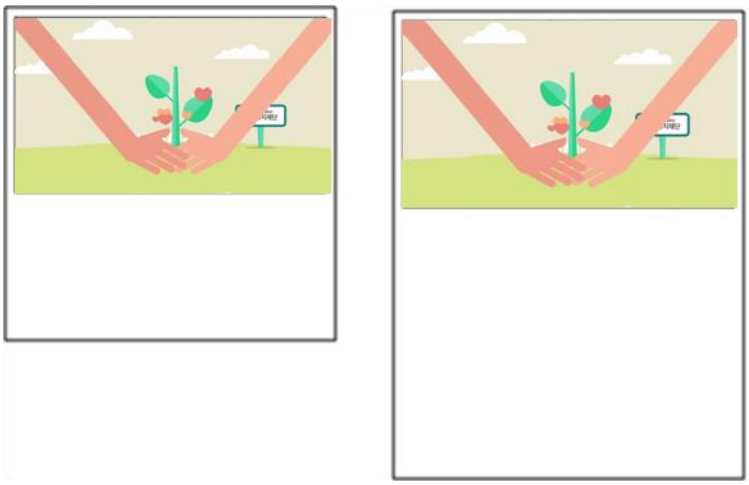

Figure 8. Flexible media on 320x320 and 360x480

If you set the video element size using a fixed size, as shown in Figure 9, you get no benefit from the increased resolution, and the layout on the whole screen becomes inharmonious, decreasing usability.

```
.video iframe { width: 320px; height: 240px; }
```

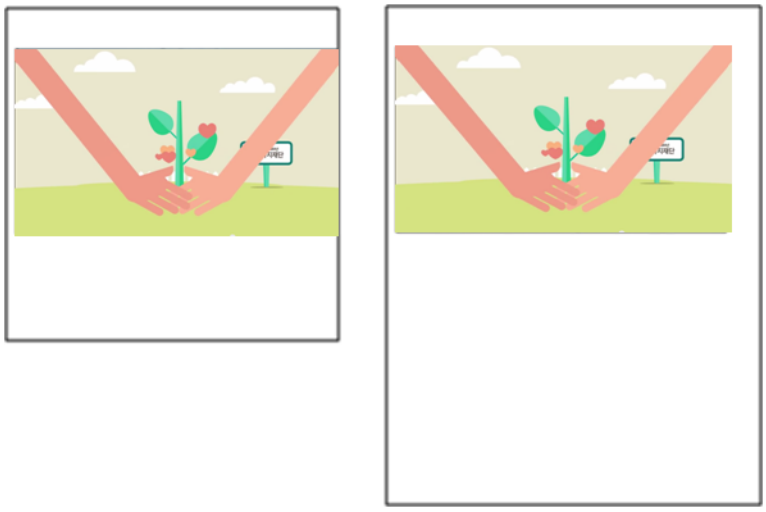

Figure 9. Fixed size media element

Exception in Product Support

Tizen Web Runtime provides a mechanism that helps you to migrate from a previously released product to new product. This is a product-specific feature, and its behavior is not guaranteed on a normal Tizen-based device. Currently, migration is supported for Gear2 applications. The migration allows you to run a Gear2 application on a Gear S device, while keeping the 1:1 aspect ratio and performing a scale-up without any modification of the source code (the aspect ratio of Gear2 is 1:1, while the aspect ratio of Gear S is 1:1.3).

This mechanism is an exception to the previous policy, and was introduced to support clock faces on a watch device. On the clock face application UI, just scaling does not prevent the usability drop if the aspect ratio is not kept, as shown in Figure 10.

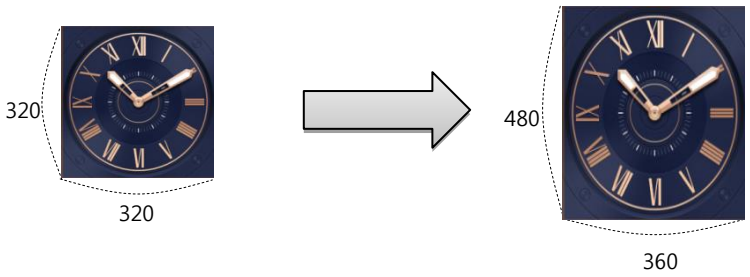

Figure 10. Clock face applications with the basic policy

If the aspect ratio and view position changes are critical for the usability of the application, you must add the following preference value line to the config.xml file in the wgt package:

```
<preference name="view-compat" value="square"/>
```

When Tizen Web Runtime reads this preference from the config.xml file, it assumes that the application is written for a square-resolution-based device and keeping the aspect ratio and center-positioning is essential for the right UX. In this case, the Web Runtime scales up the application view size and puts the view on the center of the screen with the 1:1 aspect ratio.

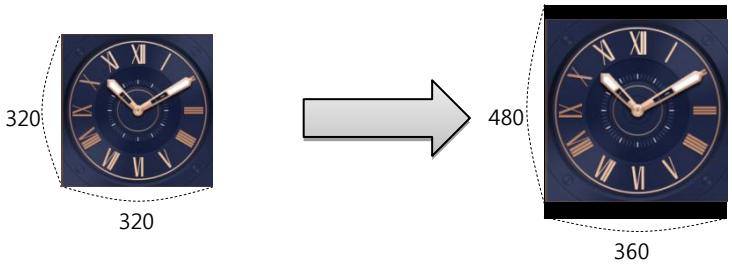

Figure 11. Exception handling with scale-up and center-positioning

This feature is not the recommended way to support any future device. However, it can be useful if you want to quickly migrate your previous Gear2 application to a Gear S device.

Single Layout Conclusion

This section illustrates a complete application layout that takes advantage of the issues discussed in previous sections. In particular, it shows how to set the viewport properly to help the Web engine scaling mechanism and how to create a CSS-based layout.

First of all, set the targeted size of the content area as 320 px in the viewport meta tag, and disable the user-scalable property, as shown in the following code snippet.

```
<html>
<head>
```

```
    <link href="css/style.css" rel="stylesheet" type="text/css">
    <meta name="viewport" content="width=320px,user-scalable=no" />
</head>
<body>
    <main>
        <div class="tile left">
            <div id="box1" class="box"></div>
        </div>
        <div class="tile right">
            <div id="box2" class="box"></div>
        </div>
        <div class="tile left">
            <div id="box3" class="box"></div>
        </div>
        <div class="tile right">
            <div id="box4" class="box"></div>
        </div>
    </main>
</body>
</html>
```

In the above HTML code, the content area consists of 4 box elements, and their UI layout is described in the `style.css` file. The following CSS code programs the layout and the specific box element styles that were defined in the HTML file.

The code snippet shows that the content size is set to be same as the viewport size. Each box is filled with a different color and placed so that it fills a quarter of the content area. Because each box size is defined with the width and height of 100%, the relative size of each box element remains the same even if the viewport size changes.

```
/* Default, used for all*/
* { box-sizing: border-box; }
body, div, h1 { margin: 0px;    padding: 0px; }
html, body { width: 100%; height: 100%; overflow-x: hidden; }

#box1 { background-color: rgb(255, 255, 141); }
#box2 { background-color: rgb(165, 241, 238); }
#box3 { background-color: rgb(248, 179, 179); }
#box4 { background-color: rgb(192, 161, 246); }
.tile
{
    width: 50%;
    height: 50%;
    float: left;
}
.left { padding: 10px 5px 5px 10px; }
.right { padding: 10px 10px 5px 5px; }

.box
{
    width: 100%;
    height: 100%;
}
```

Figure 12 shows a final view in a 320px device set as a target device in the HTML file. As defined in the CSS file, the content area fills the whole screen, 320x320 pixels, and the 4 box elements equally divide the area.

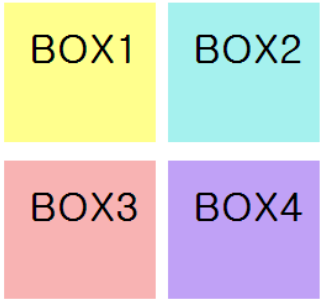

Figure 12. Original layout on a target device

Figure 13 shows that the relative layout and scaling are correctly applied even when the same application is run on a mobile device with the aspect ratio of 16:9.

Figure 13. Scaled layout on a 16:9 mobile device

The following code snippet shows an example where the background color is replaced with an image while the layout is exactly same as in the 4 box example. The Figure 14 and Figure 15 show how the layout including media elements (such as images) functions exactly like the basic box layout.

```
/* Default, used for all*/
* { box-sizing: border-box; }
body, div, h1 { margin: 0px;    padding: 0px; }
html, body { width: 100%; height: 100%; overflow-x: hidden; }

#box1 { background-color: rgb(255, 255, 141); }
#box2 { background-color: rgb(165, 241, 238); }
```

```
#box3 { background-color: rgb(248, 179, 179); }
#box4 { background-color: rgb(192, 161, 246); }
.tile
{
    width: 50%;
    height: 50%;
    float: left;
}
.left { padding: 10px 5px 5px 10px; }
.right { padding: 10px 10px 5px 5px; }

.box
{
    width: 100%;
    height: 100%;
    padding: 20px;
    background: no-repeat center;
    background-size: 100% 100%;
}

#box1 { background-image: url('../a.png'); }
#box2 { background-image: url('../b.png'); }
#box3 { background-image: url('../c.png'); }
#box4 { background-image: url('../d.png'); }
```

Figure 14. Image-based layout on a target device

Figure 15. Image-based scaled layout on a 16:9 mobile device

3. Multiple Layout for Multiple Devices

The first part of the document explained how to develop an application usable in various device categories by taking advantage of the viewport fitting feature provided by the platform for a single layout. In many cases, the desired end result can be achieved using scaling and a relative layout. However, this approach does not always provide the best quality for the end user.

This section describes how you can support the best possible layout and usability with some additional development efforts. For instance, you can create an application which, on a tablet, shows the main information (clock face) while also delivering more meaningful information (calendar), while on a watch device only the main information (clock face) is displayed. This kind of end result, shown in figure 16, cannot be achieved by only scaling up the whole layout of the watch device.

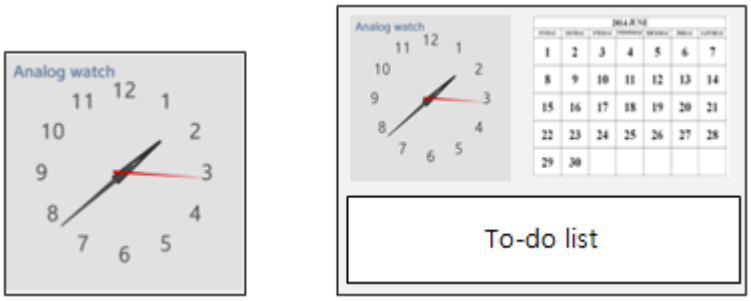

Figure 16. Application for a watch and tablet

W3C CSS3 Media Queries already support the majority of techniques needed to provide category-based layouts. This section you learn in detail how to support the optimal layout using CSS techniques.

Viewport Setting for Multiple Layouts

The difference from the single layout case example is that the content width of the viewport is set to “device-width”. This sets the layout viewport width to the ideal viewport width. Setting the viewport width to “device-width” tells the Web engine that the application adapts to the device width. In short, the “device-width” setting is needed to create an adaptive and responsive Web application.

To use the tag in an HTML file, set its name and content:

```
<meta name="viewport" content="width=device-width, user-scalable=no" />
```

Category Classification

When creating multiple layouts, you must first configure the layout categories. In other words, a classifying category is needed for layout fitting on the current executable environment. Media queries are supported in CSS3 to give Web application information to the executable environment.

W3C Media Queries

Media queries consist of a media type and expressions of media features.

The media types indicate the media on which the current Web application is running. They are defined in HTML4. The complete list of media types is: ‘aural’, ‘braille’, ‘handheld’, ‘print’, ‘projection’, ‘screen’, ‘tty’, and ‘tv’. You can declare that sections apply to certain media types inside a CSS style sheet. As in the following example, you can declare “screen” as a media type and describe its layout inside {…}. That code applies the layout inside {…} when the Web application is run on the executable context on the screen type.

```
@media screen { ... }
```

Several media queries can be combined in a media query list to configure the executable environment. The following example shows a case in which the Web application runs on the executable environment having a “screen” media type and a 500 pixel minimum width. Its final view shows the layout inside {..} to the end user.

```
@media screen and (min-width:500px) { ... }
```

Table 1 lists the CSS media features, which you can use to specify the layout utilizing the media queries.

Table 1. Media features

Feature	Value	Min/Max	Description
color	integer	yes	Number of bits per a color component
color-index	integer	yes	Number of entries in the color lookup table
device-aspect-ratio	integer/integer	yes	Aspect ratio
device-height	length	yes	Output device height
device-width	length	yes	Output device width
grid	integer	no	Set to true for a grid-based device
height	length	yes	Rendering surface height
monochrome	integer	yes	Number of bits per pixel in a monochrome frame buffer
resolution	resolution ("dpi" or "dpcm")	yes	Resolution
scan	"progressive" or "interlaced"	no	Scanning process of the "tv" media types
width	length	yes	Rendering surface width

As shown above, media features can be combined when a detailed layout configuration is needed.

Classification for Display Mode (Landscape and Portrait)

You can configure the device landscape and portrait mode using the “device-aspect-ratio” in W3C Media Queries. This approach is already commonly used in creating mobile Web applications. The following example shows how the max-device-aspect-ratio feature is used.

If you use the orientation:portrait/landscape feature, the layout can change unexpectedly when the virtual keypad is displayed. To avoid the problem, use the device-aspect-ratio to configure the portrait and landscape mode.

```
@media screen and (max-width: 320px) and (max-device-aspect-ratio: 1/1)
{
   // For portrait mode
}
@media screen and (max-width: 640px) and (min-device-aspect-ratio: 11/10)
{
```

```
    // For landscape mode
}
```

4. Summary

This document offered some important tips for developers to compose a Web application UI layout that performs on multiple devices.

At the beginning, it described how you can create 1 application layout that works in all devices. In the second half, it showed how to create multiple layouts for multiple devices using CSS media queries.

The document also provided full HTML/CSS samples that allow you to “copy and paste” the code and test how it works on Tizen.