A Delphi Developers Guide for 4K Displays

For the last 20 years, Windows desktop applications faced a relatively unchanged display environment. Sure, displays grew in size and the predominant aspect ratio changed from 4:3 to 16:9. A pixel, however, was still a pixel and a logical pixel corresponded to a physical dot on the screen.

This was possible, because the physical display size grew at about the same rate as the resolutions increased. Compare your first 14" CRT monitor with its 1024 x 768 pixel resolution to a typical modern consumer display with a HD resolution of 1920 x 1080 pixels at 24 inch. The physical size of each pixel - or the distance between the dots on the screen - is approximately the same. At least is the difference not huge.

With modern Ultra-HD displays ("4K displays") at sizes of 28" or smaller, this game has changed forever. Simply because no human being is sharp-sighted enough to comfortably read text at the size of a pinhead. Texts, headings, button captions have to be enlarged and with larger text, everything in the user interface needs to be scaled.

When Apple introduced the first mobile devices with retina display, the iOS developer community was prepared for the shift. Besides, iOS and OS/X as well as Android have significantly better operating system support for scaled applications. Windows has not, unfortunately. It leaves it to the individual developer and the development tool to prepare for high resolutions. And the change is as huge as the new display resolutions, which are here to stay, because they are getting more common with every day.

This guide intends to help you getting your Windows applications ready for 4K. It will explain the challenge and tell you how to master it.

Happy 4K coding! :-)

Alexander Halser, November 2014

About the author

Alexander Halser is the founder of and senior developer at *EC Software GmbH*, the makers of **Help+Manual**. You can reach him directly by email to alexander.halser@ec-software.com or a visit to http://www.helpandmanual.com.

Contents

Technical Overview

1

Dots per Inch explained

When you look very closely at your LCD display (use a magnification lens for help), you will see tiny red, green and blue dots on the screen. Three of these dots resemble an RGB-pixel, which we call a *physical pixel*.

An inch is a physical inch (e.g. 24" for the diagonal of a wide-screen display) and how many of these dots fit into an inch, is expressed by the **DPI value**, which means *dots per inch*. To be exact, it means how many triple dots (Red, Green and Blue) fit into an inch. This number is often also referred to as the **PPI value**, which means *pixels per inch* (not *points* per inch). They are all the same and for simplicity, we stick to the DPI term, because it's more commonplace.

A Samsung 23" HD display (1920 x 1080) has 96 DPI, a 24" Dell Ultra-Sharp UP2414Q (3840 x 2160) has 184 DPI. This link helps you to calculate the DPI value of your own display.

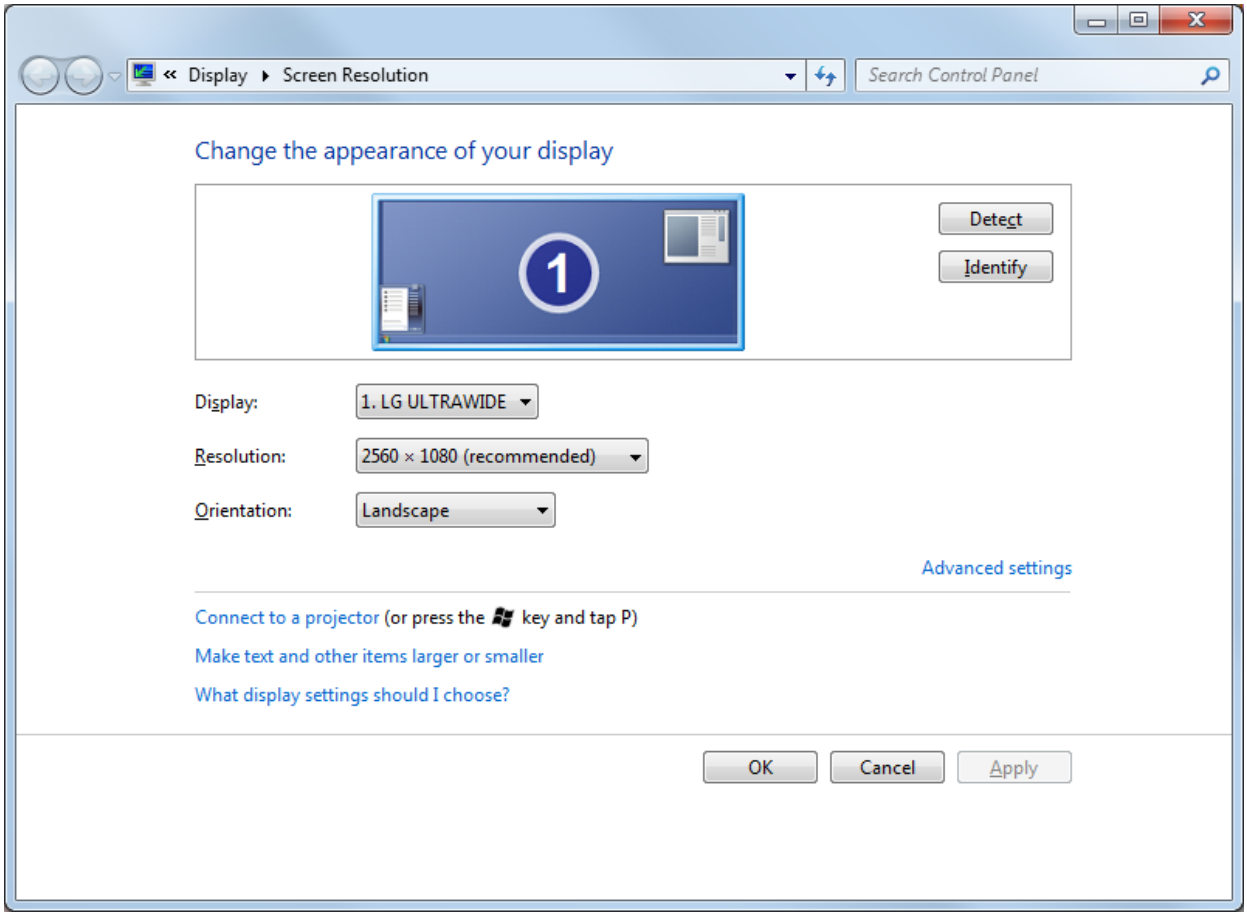
In print, we are used to much higher DPI values. 600 DPI for an ink-jet printer, 1200 DPI for a laser printer are common. The principle is the same: it tells you, how many separate dots or ink drops the printer is able to squeeze into a physical inch on paper.

Quick Summary

The DPI value tells you the resolution of a display relative to its physical size.

Native Resolutions

LCD displays are mostly used at their native resolution. In other words, you tell Windows to show the desktop at the same resolution as your display supports:



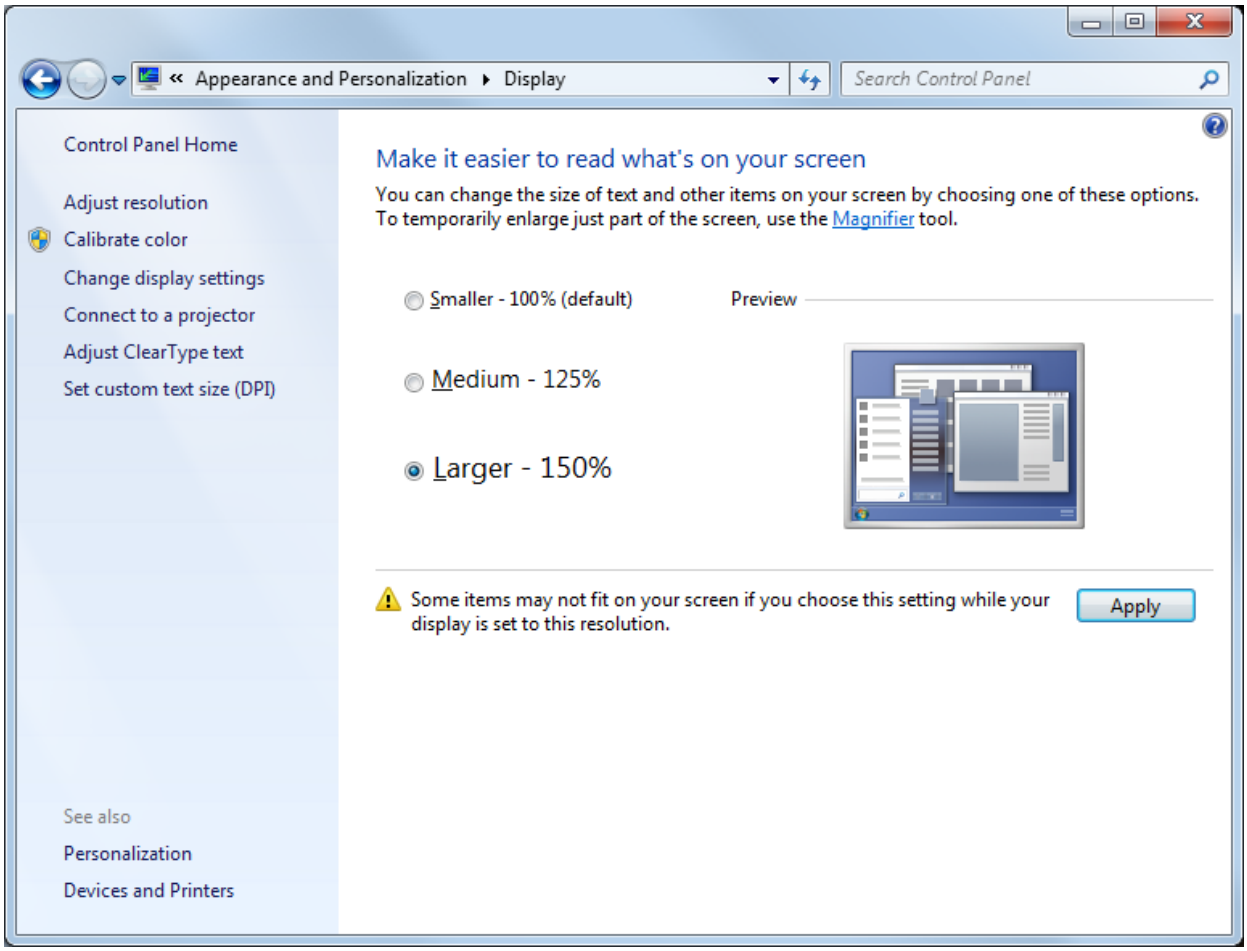
Windows control panel with screen resolution settings

If you change this in Windows - try to make it smaller than the native resolution of your display - the resulting picture is blurred and not pretty to look at. Except for visually impaired users, who prefer a blurred picture for the sake of enlarged text and images, most computer users don't.

They will rather change the *logical DPI* value of Windows.

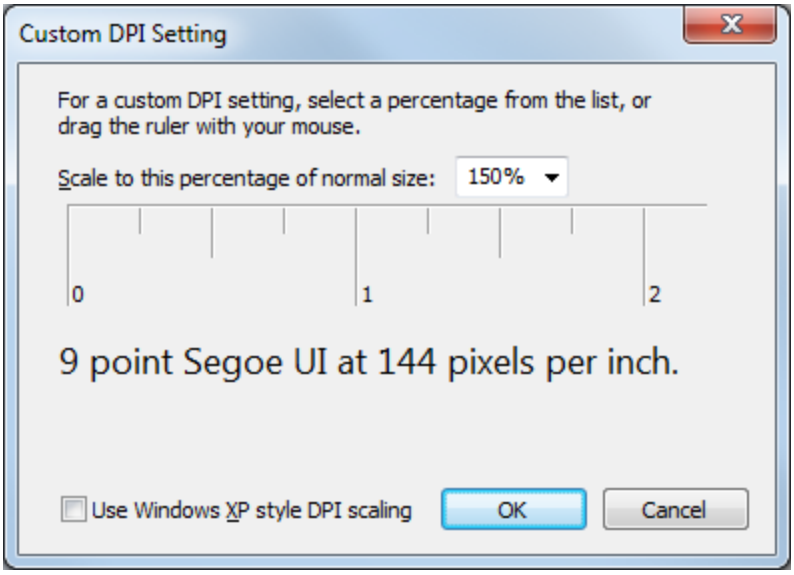
Windows and DPI

While we discussed physical inches and physical DPI above, Windows has its own *logical DPI setting*. The default is 96 DPI (this is 100%) and you can change this in the control panel's Display settings:



Windows control panel with logical DPI settings

This logical versus physical DPI is confusing, isn't it? Sure enough, it is. And for this reason, Microsoft meanwhile uses a different wording in order to not confuse consumers. They call it "*Make text larger or smaller*", which is a pretty accurate description of what the logical DPI value does. Nonetheless, you are still served the "DPI" term when you click "*Set custom text size*" in the dialog above.



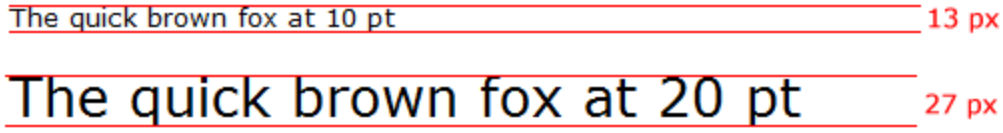
Dialog for a custom logical DPI value

In the picture above, the text size is set to 150%. The default DPI of 96 multiplied by 1.5 results in a new (logical) DPI value of 144. So, does this mean that Windows squeezes more pixels into one physical inch of the display? Of course not. It changes the formula, how text sizes are calculated.

Text is measured differently

Text is measured in *point* - that's also the value that you specify for font sizes in Delphi. A *point* is defined as 1/72 inch. Always, everywhere. Point is a hard-wired entity like 1 inch = 1/12 of a foot.

Take a 10 pt font for example. The height of the font will be 10/72 inch. When an inch has 96 DPI (or pixels), the font will be 10/72*96 pixels high. That's a rounded height of 13 pixels on screen. A 20 pt font is 20/72*96 = 27 pixels rounded.

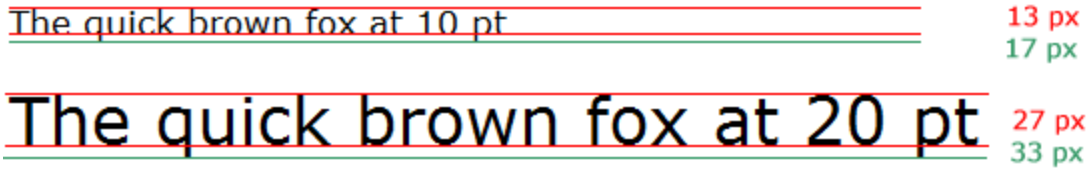


10pt and 20pt fonts on a 96 dpi setting (Windows default)

So, what happens, when we *change the definition of an inch*? If our inch should be composed of 120 pixels instead of the default of 96? In Windows terms, this is a "medium" font size or 125% (compare with the control panel screenshot above).

If we define inch to be 120 pixels, the 10 pt font will result in 17 pixels height (10/72*120 = 17) and the 20 pt font will be 33 pixels (20/72*120 = 33). And because a pixel in Windows

still means a physical pixel on the display, the result is bigger text, when measured in pixels.



10pt and 20pt fonts at 120 dpi (in Windows: 125%)

If a display has a high *physical* DPI value (= the pyhsical pixels are quite small), a regular *logical* DPI of 96 in Windows might result in text that is so small, that it becomes hard to read. By increasing the *logical* DPI in Windows, text becomes larger in pixels, restoring a comfortable reading size on screen *and* a smoother font, because it's made up of more pixels.

If text goes bigger, bigger goes the form

In the two images above, it is clearly visible that the same 20 point-sized text grows in height and width, when the logical DPI value of Windows is increased. However, if the form and the controls on this form remained the same pixel size, the enlarged text would be clipped.

Delphi manages this automatically at runtime, when the **`Scaled`** property of a form is true. The unit *Vcl.Controls.pas* has a function called **`ScaleBy()`**, which changes the position and size of the form and all its controls, when the form is loaded. If **`Scaled`** is *false*, the form will not be scaled and the font will remain at the same pixel size as defined at designtime. We'll examine this automatism in a moment.

Where 96 DPI in Windows comes from

You might ask yourself, why 96 pixels is the default for a logical inch on Windows. Would it not be more intuitive, if a logical inch was defined as 72 pixels, so that 1 pixel is 1/72 inch, similar to the *point* entity? On Mac OS/X this is exactly the case.

This article sheds light on the 96 DPI mystery.

If you designed web sites in the late 1990ies, you certainly remember a puzzling difference in the layout of a web page, that looked pixel-perfect on Windows, but awkward in a Mac browser and vice versa. Because images and boxes and the layout in general were defined in pixels, the fonts in point. On Macs, font sizes in pixels were calculated the same way as in Windows: pixel = point / 72 * DPI. Because DPI was 72 by default, points were effectively pixels on a Mac, but 20% larger on Windows. The point-based font sizes blew up every pixel-perfect web design. That's why designers started to use pixel for font sizes, leading to lengthly discussions over the usability of web sites with fixed-sized fonts.

Technical Overview

1

Today, this is a thing of the past. It doesn't matter anymore, whether you define the font size *for a HTML page* in pixel, point or any other entity. Modern browsers take care of the display differences and scale everything. For Delphi applications, however, the difference between a pixel and a point still matters.

Quick Summary

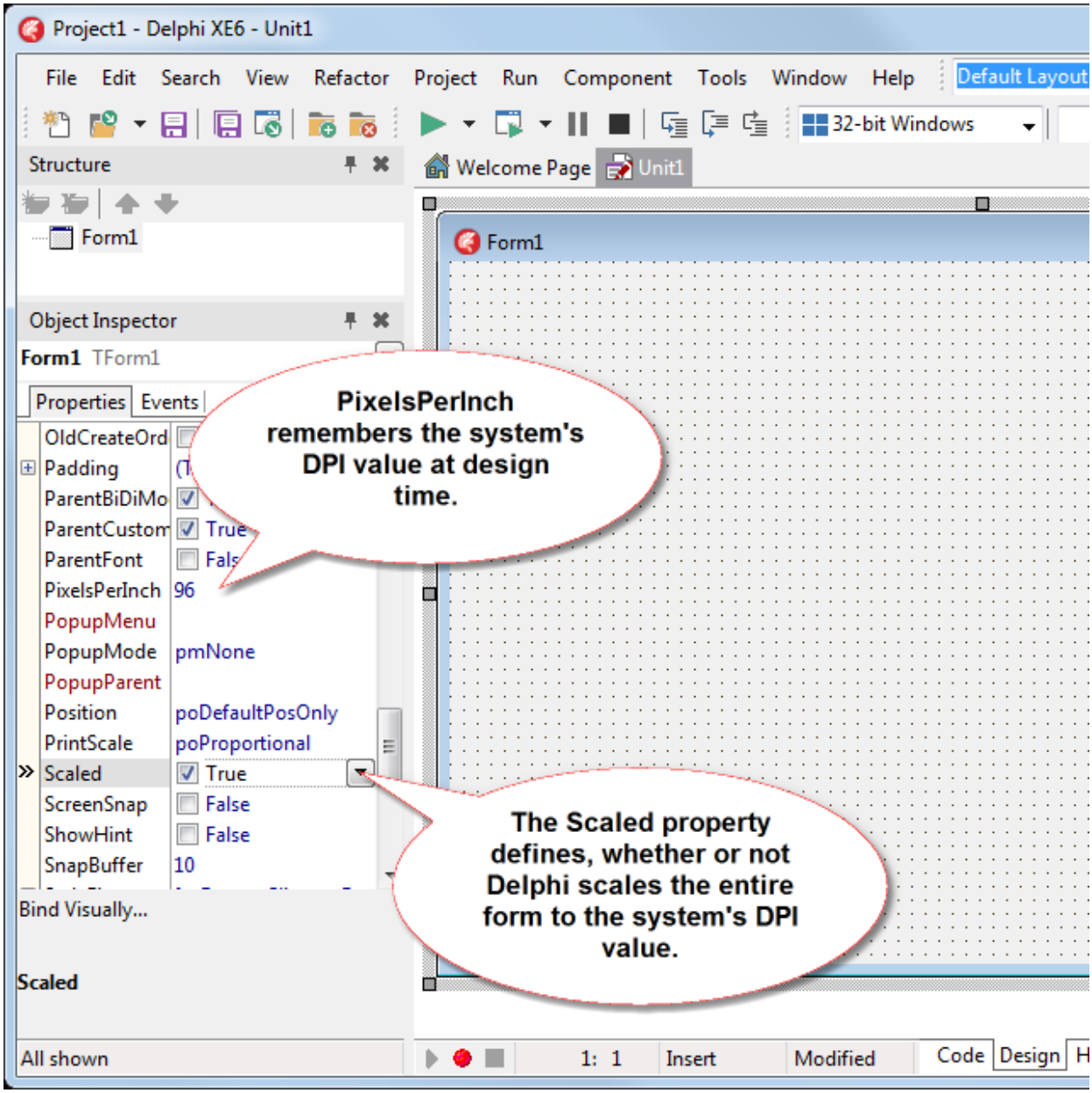
The logical DPI value in Windows has nothing to do with the physical DPI of the display. It defines, how large a text is displayed. More logical DPI = larger text. The value is always compared to the **constant 96**, which is the default on Windows, or "100%". A value of 144 (= 96 x 1.5) means, that items on the screen will appear 50% larger than "normal".

If text appears larger, everything else must be larger as well, to fit the text. The logical DPI is in fact a setting that scales everything, not only text.

Delphi and DPI

The Delphi TForm class has a property called **Scaled**, which is responsible for scaling the form proportionally to the current Windows DPI value at runtime. When **Scaled** is *true*, Delphi scales the form including child controls of the form and nested controls.

All controls' positions and sizes are relative to the DPI value at design time. To remember this design time value and use it as the reference point for the scale, Delphi stores it in the property TForm.PixelsPerInch.



Delphi XE6 form designer with PixelsPerInch and Scaled property

What the scale function does

The scale function changes the position and size of the form and all its child controls including nested controls. In particular, it scales:

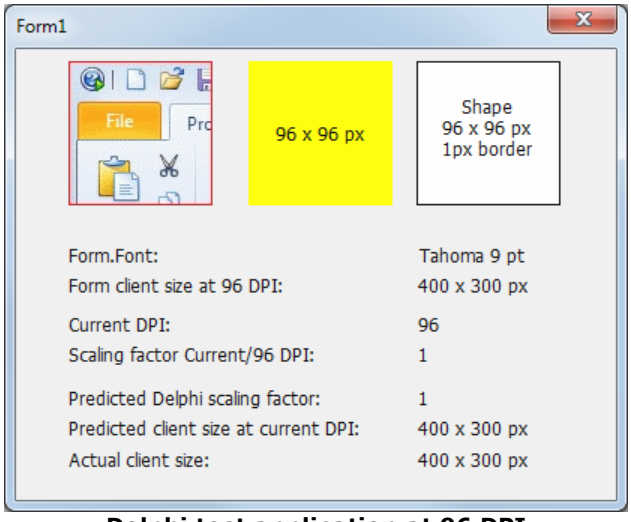
- Width, Height and constraints
- Margins and Padding
- Scrollbars
- Font sizes
- Third party controls might implement the `TControl.ChangeScale()` method to scale additional properties.

Roundup

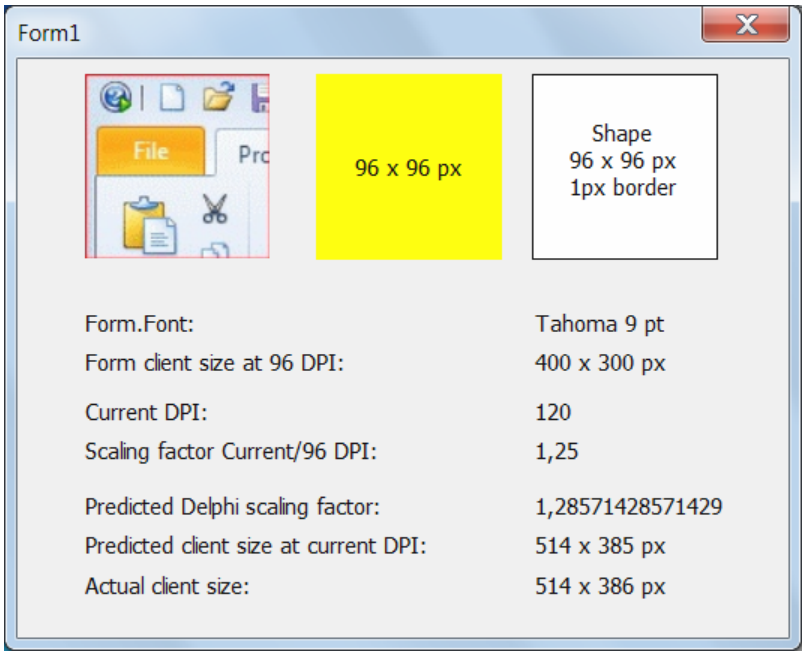
To have your Delphi form scaled automatically on Windows systems with a logical DPI > 96, leave the Scaled property true.

Computing the scaling factor

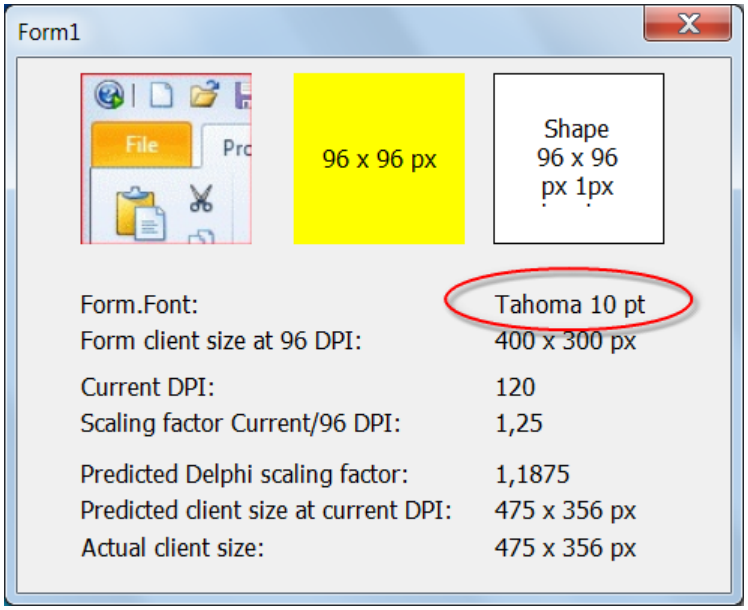
Though Delphi saves the design-time DPI value as a reference point for the scaling factor, it does not compare design-DPI to runtime-DPI directly. Instead, Delphi computes the height of the form's font in pixels at both, design-DPI and runtime-DPI and scales the form by the difference. This can lead to interesting results, due to rounding differences in the pixel-height of the font.



Delphi test application at 96 DPI
Form.Font = Tahoma 9 pt



Delphi test application at 120 DPI
Form.Font = Tahoma 9 pt



Delphi test application at 120 DPI again but with a 10 pt font instead of 9 pt

The screenshots above show the same test application at 96 DPI (100%) and 120 DPI (125%) scaled. The difference between the last two images is the font of the form, assigned in the Object Inspector. The form with a *Tahoma 9 pt* font is scaled to about 128.6% at 120 DPI. The same form with a *Tahoma 10 pt* is scaled to only 118.75%.

How can the same form become *smaller* when the default font of the form is *larger*?

Well, Tahoma 10 pt is larger than 9 pt, that's true. But it's not 9 pt versus 10 pt, that is compared here. Delphi tests the 10 pt font height in pixels at 96 and 120 DPI and because of rounding differences in the pixel result - in particular with small fonts - the scaling factor can vary. Let's have a look:

	Tahoma 9 pt	Tahoma 10 pt
Point size of font	9 pt	10 pt
Height in pixel on 96 DPI*	14 px	16 px
Height in pixel on 120 DPI*	18 px	19 px
Scaling factor for 120 DPI	18:14 = **1.2857**	19:16 = **1.1875**

* The height in pixel is computed including the internal leading of the font

To compute the scaling factor of the form exactly like Delphi does, use the following function:

```
function FontHeightAtDpi(aDPI, aFontSize: integer): integer;
var
  tmpCanvas: TCanvas;
begin
  tmpCanvas := TCanvas.Create;
  try
    tmpCanvas.Handle := GetDC(0);
    tmpCanvas.Font.Assign(self.Font);
    tmpCanvas.Font.PixelsPerInch := aDPI;
    tmpCanvas.Font.Size := aFontSize;
    result := tmpCanvas.TextHeight('0');
  finally
    tmpCanvas.free;
  end;
end;

intFontHeightAt96Dpi := FontHeightAtDpi(96, ThisFont.Size);
intFontHeightCurrent := FontHeightAtDpi(Screen.PixelsPerInch,
                                        ThisFont.Size);
```

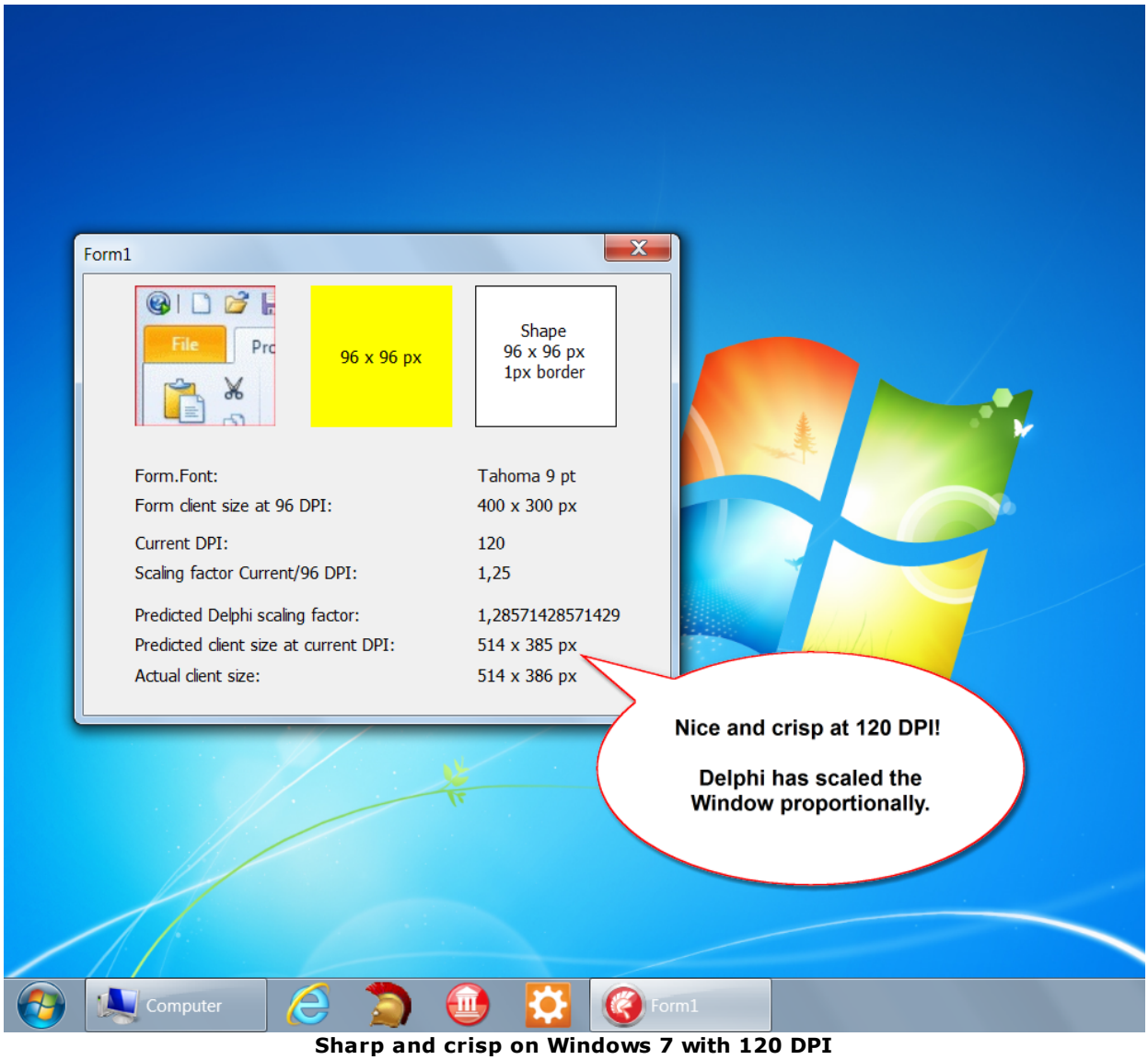
Quick Summary

The scale is not linear to the DPI value, but depends on the form's font, set at design time. Assigning a completely different font to a form at runtime might lead to unpredictable results.

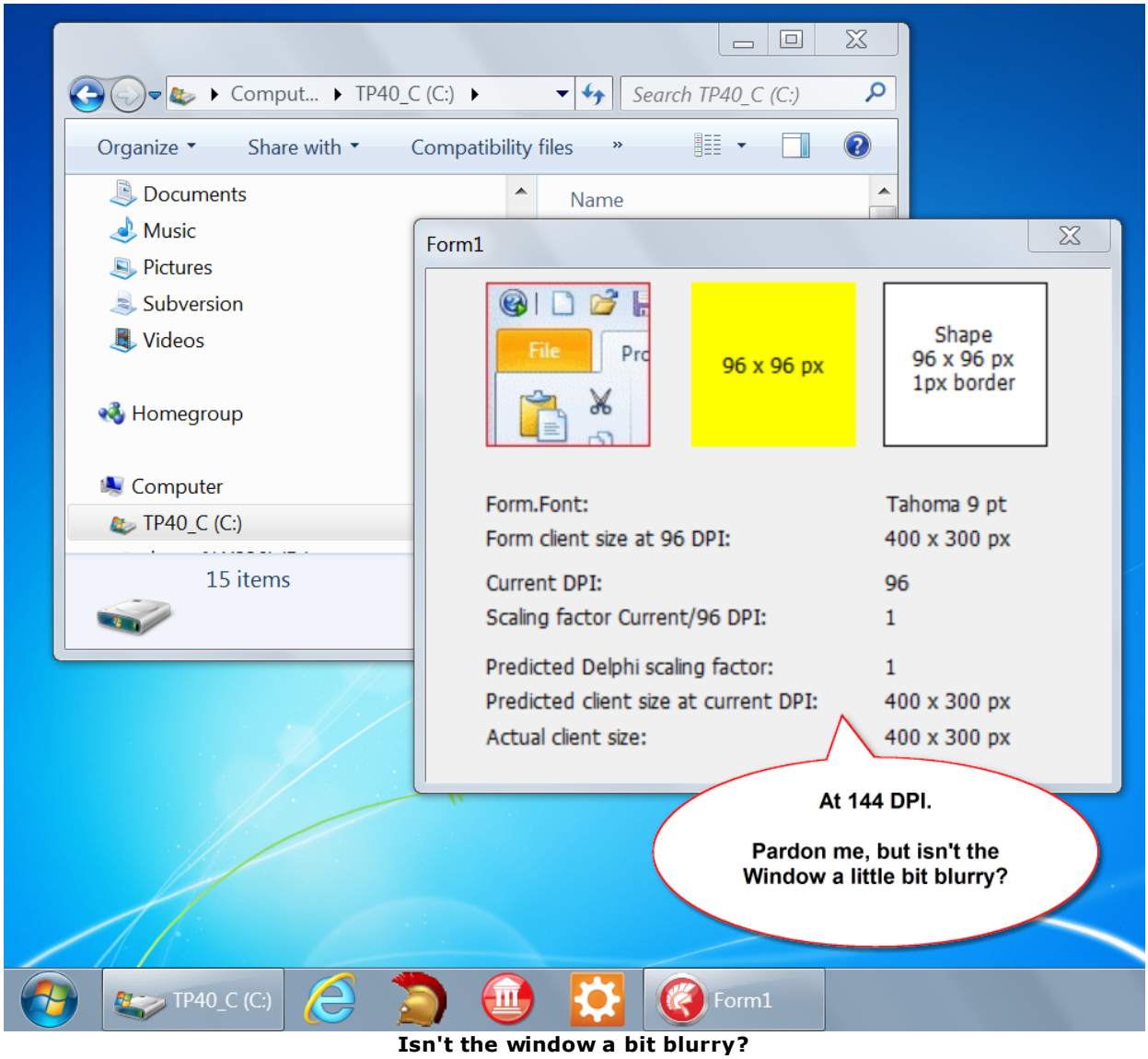
Testing high resolutions

You don't need to have a 4K monitor yourself, to test your application with high resolutions. Set up a virtual machine and increase the Windows DPI setting. Large Ultra-HD (4K) desktop displays typically have a DPI setting of 144 pixels per inch (150%), tablets with Ultra-HD resolution run with 192 DPI (200%).

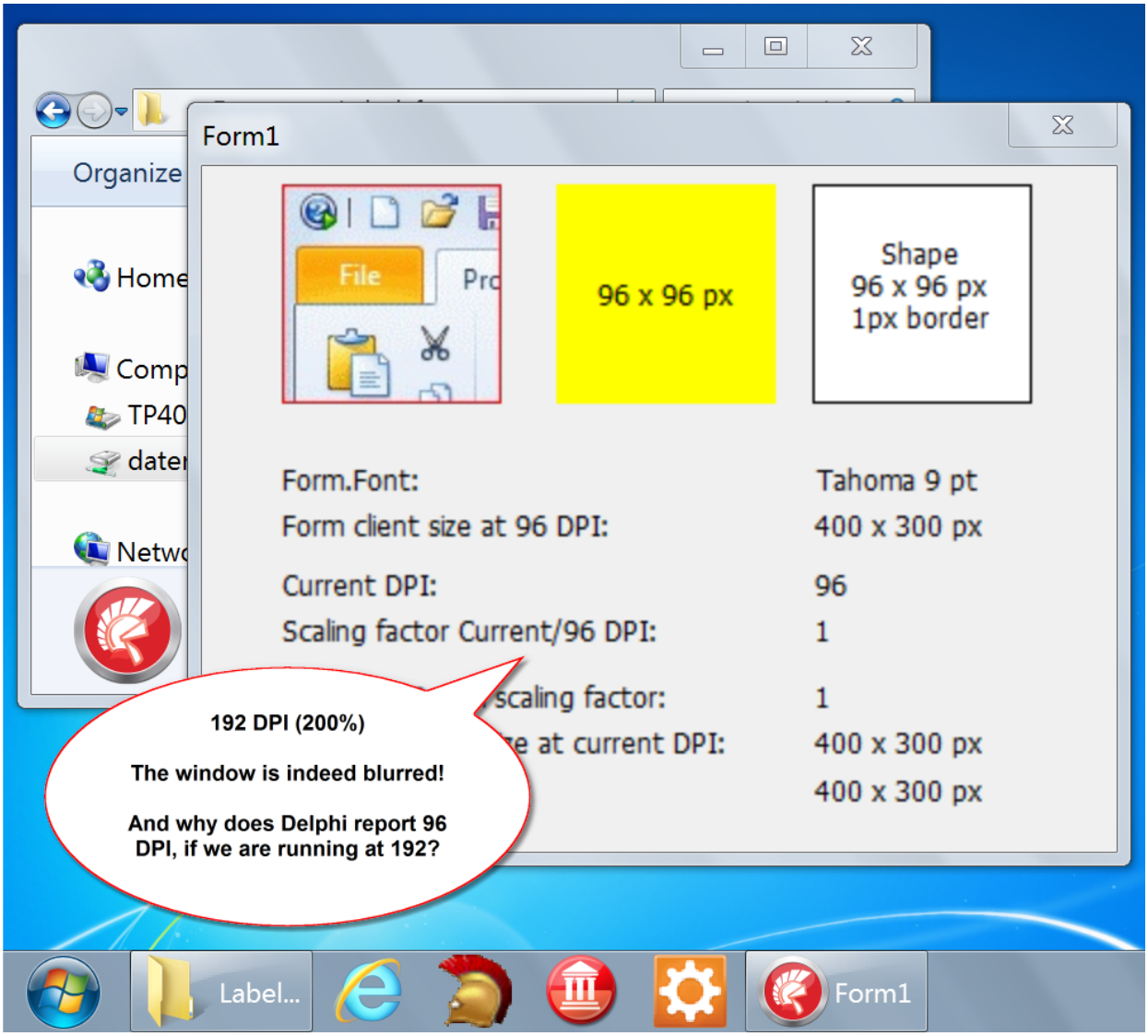
When your application scales properly at 125% and 150%, you can almost safely assume that it also runs on 200% without glitches.



Sharp and crisp on Windows 7 with 120 DPI



Isn't the window a bit blurry?



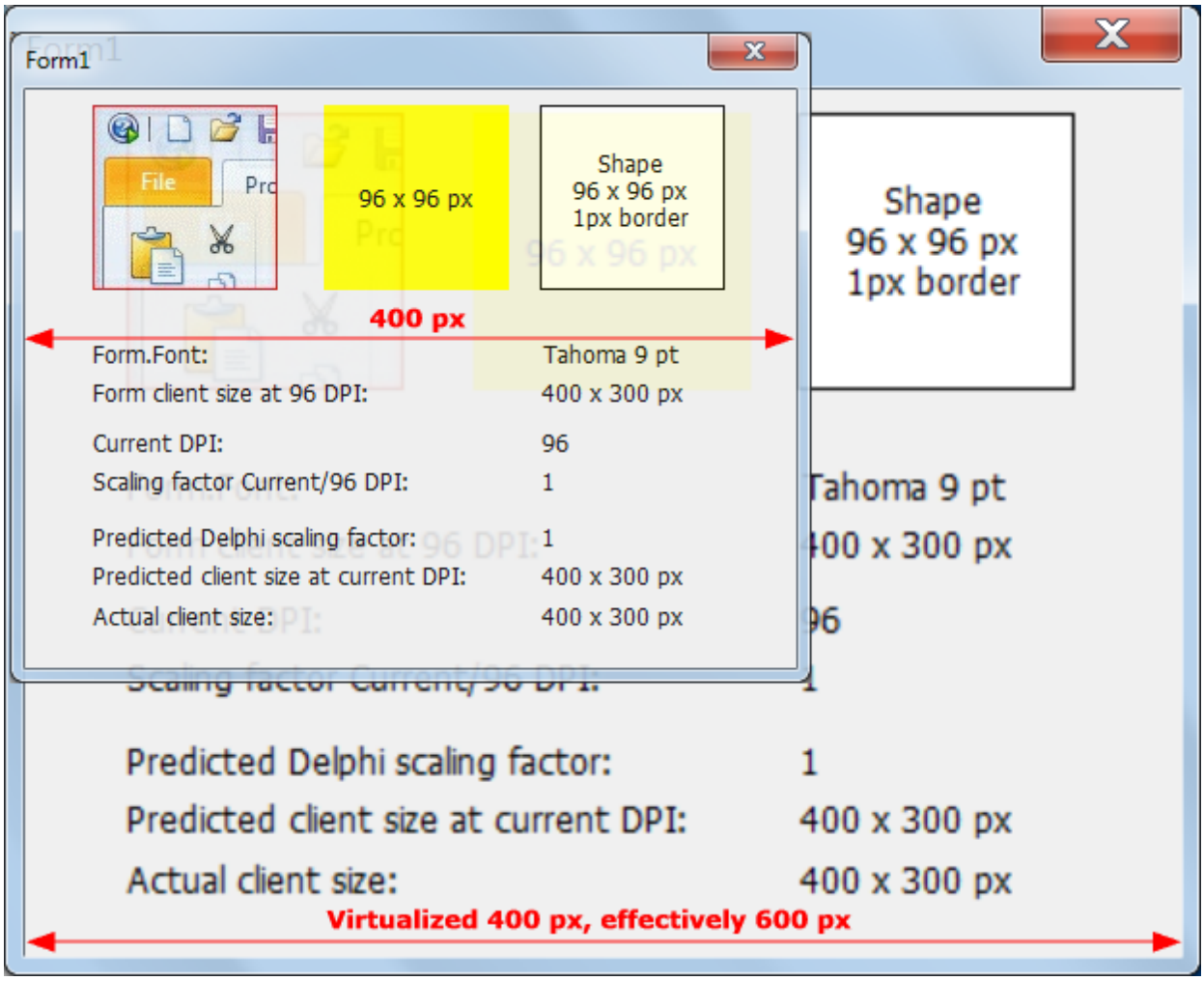
Yes, it's blurry. And at 200% it gets even worse...

Something's going on here... while the test application looks good, even scaled on 120 DPI (125%), it appears blurred on DPI settings larger than 120. Look at the last two screenshots: the application *thinks* that it is **running on 96 DPI** (normal fonts, 100%)!

Is this a Windows 7 bug? No, it's intentional and called **DPI virtualization**.

DPI virtualization in Windows

DPI virtualization was introduced with Windows Vista. Windows runs an application in a virtual 96 DPI environment and upscales it in realtime to the system DPI value. This results in slight blur. Below is a screenshot showing of the same application running in 96 DPI (native) and on 144 DPI (virtualized). Note that the **application "thinks" that it's running on 96 DPI**:



A virtualized application window at 144 DPI (background)
compared to the original window at 96 DPI (foreground)
The application "thinks" it's running on 96 DPI

Scaling in Windows XP and before

Windows (Win95 and later) always had an option to let the user change the text scale on the display. The options were "Small Fonts" (100% or 96 DPI) and "Large Fonts" (125% or 120 DPI).

Windows XP introduced custom settings beyond 120 DPI, but nobody really used it. Because Windows applications were notorious for not scaling well at "Large Fonts" in many cases. Going beyond "Large Fonts" usually blew up the entire desktop and many Windows applications became completely unusable.

So, users kept working with 120 DPI at the most and if the programmer had the providence to leave enough space between controls on a form to give them space to expand, it usually worked. 100% and 125% are not that much different. The Windows world was entrapped between 96 and 120 DPI.

Windows Vista

This changed with Windows Vista, which introduced DPI virtualization. On 120 DPI and larger, applications were run by the operating system in a virtual environment with 96 DPI. The resulting window bitmap was upscaled to the target size and everything appeared blurred. Vista also introduced an option to use the old Windows XP style DPI scaling (see screen shot below).

Windows 7 and Windows 8.0

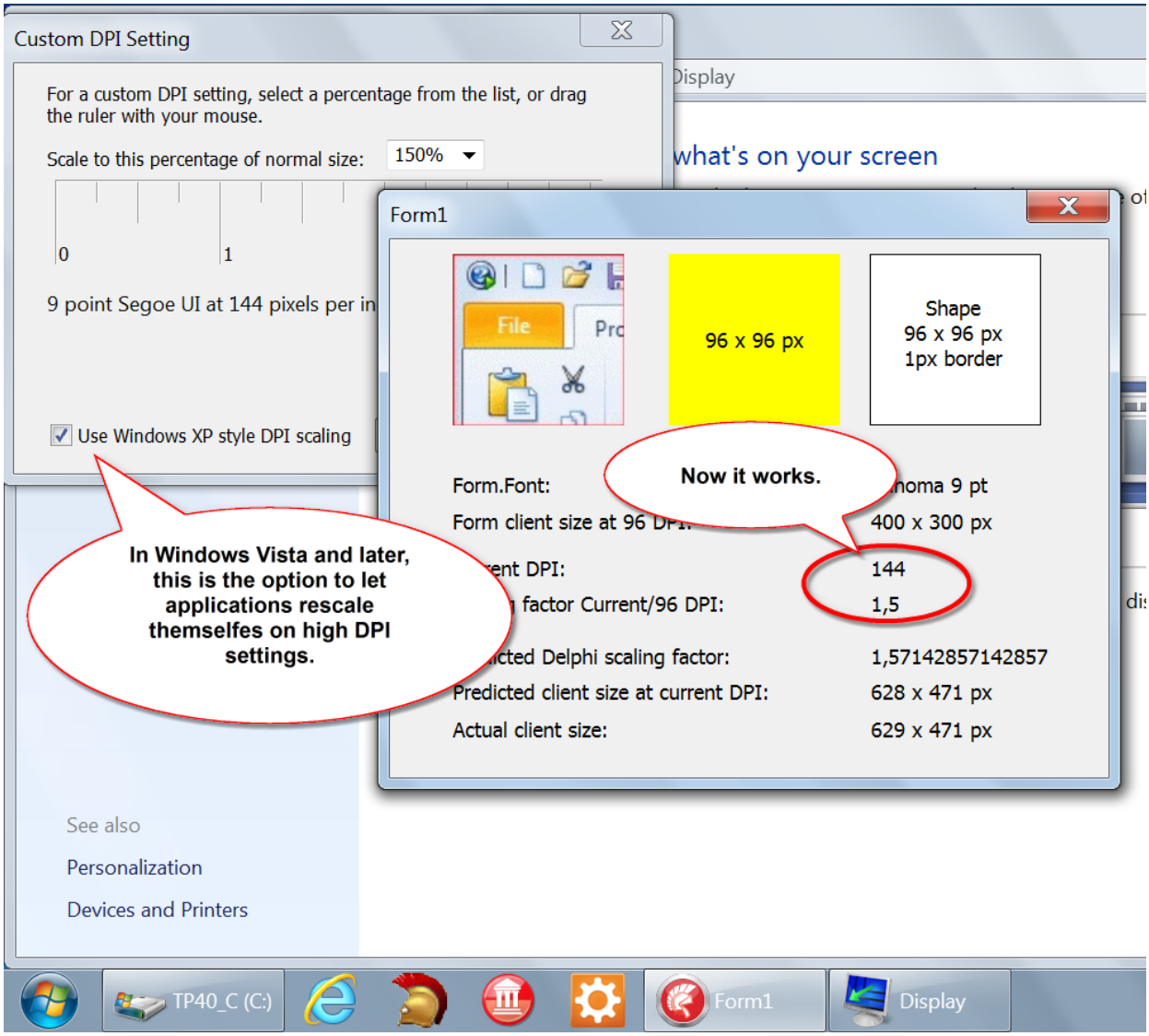
Vista clearly overshot the mark. It unnecessarily broke, what worked before. In Windows 7, Microsoft went one step back for this reason.

The new precept was: **between 96 and 120 DPI**, let applications scale themselves. Beyond 120 DPI, run the application in a virtual environment with 96 DPI and upscale the window bitmap for display, like Vista did for all DPI settings > 96.

This is true for Windows 7 and Windows 8.0 only, Win 8.1 introduced a *per-monitor* DPI scaling, which will affect us a great deal. More about that in the next chapter.

Technical Overview

1



This a "DPI-aware" application running on Windows 7 with 144 DPI

Quick Summary

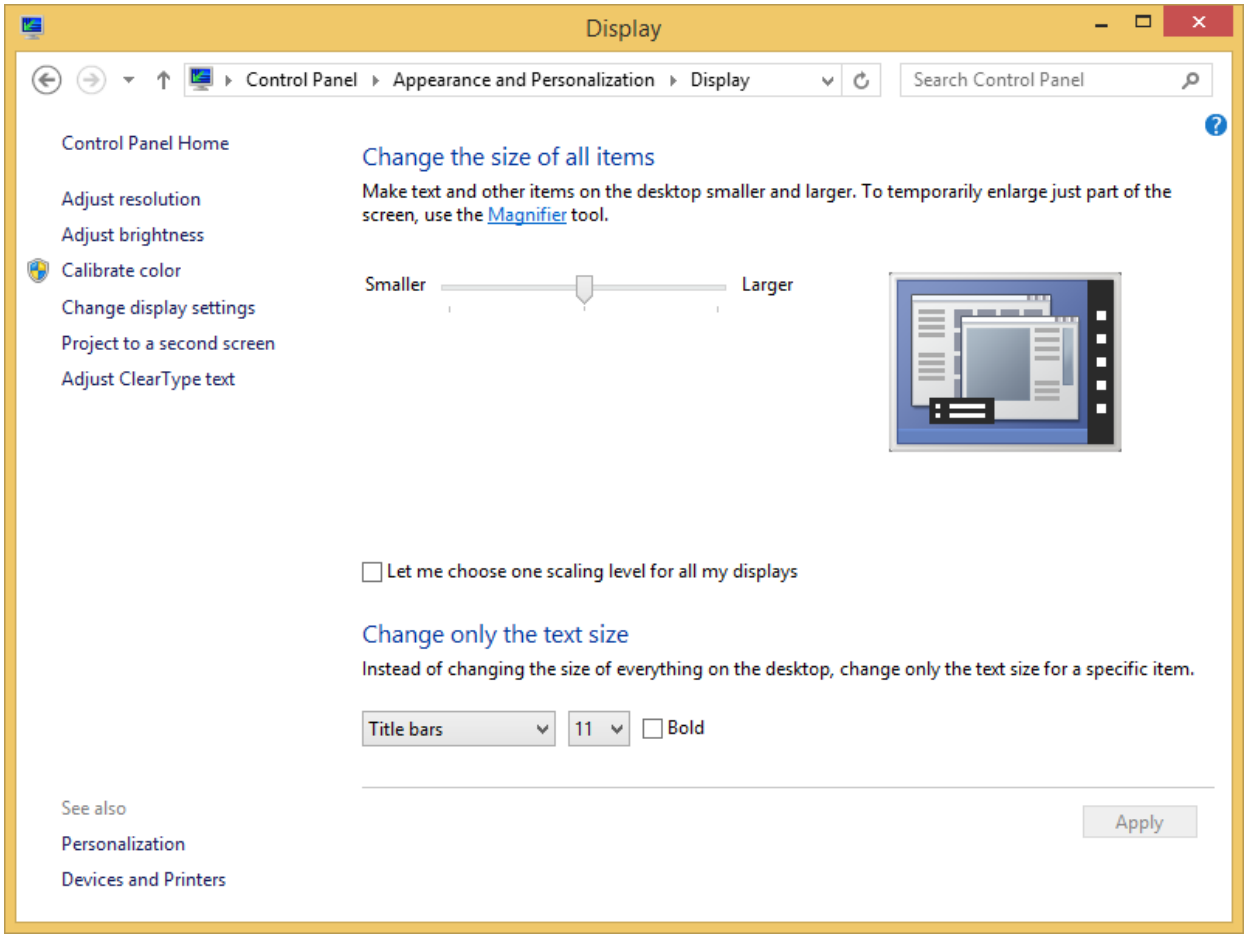
DPI virtualization helps to run legacy applications on ultra-high resolution displays without changes to the application. This tutorial, however, is about taking advantage of high resolutions and we therefore want to avoid DPI virtualization. **An application must declare itself DPI-aware**, to avoid DPI virtualization.

Without this declaration, Windows 7 and 8 gracefully allow Delphi to scale the application between 96 (100%) and 120 DPI (125%). Beyond that, DPI-awareness declaration is a requirement.

Windows 8.1 multi-montor DPI scaling

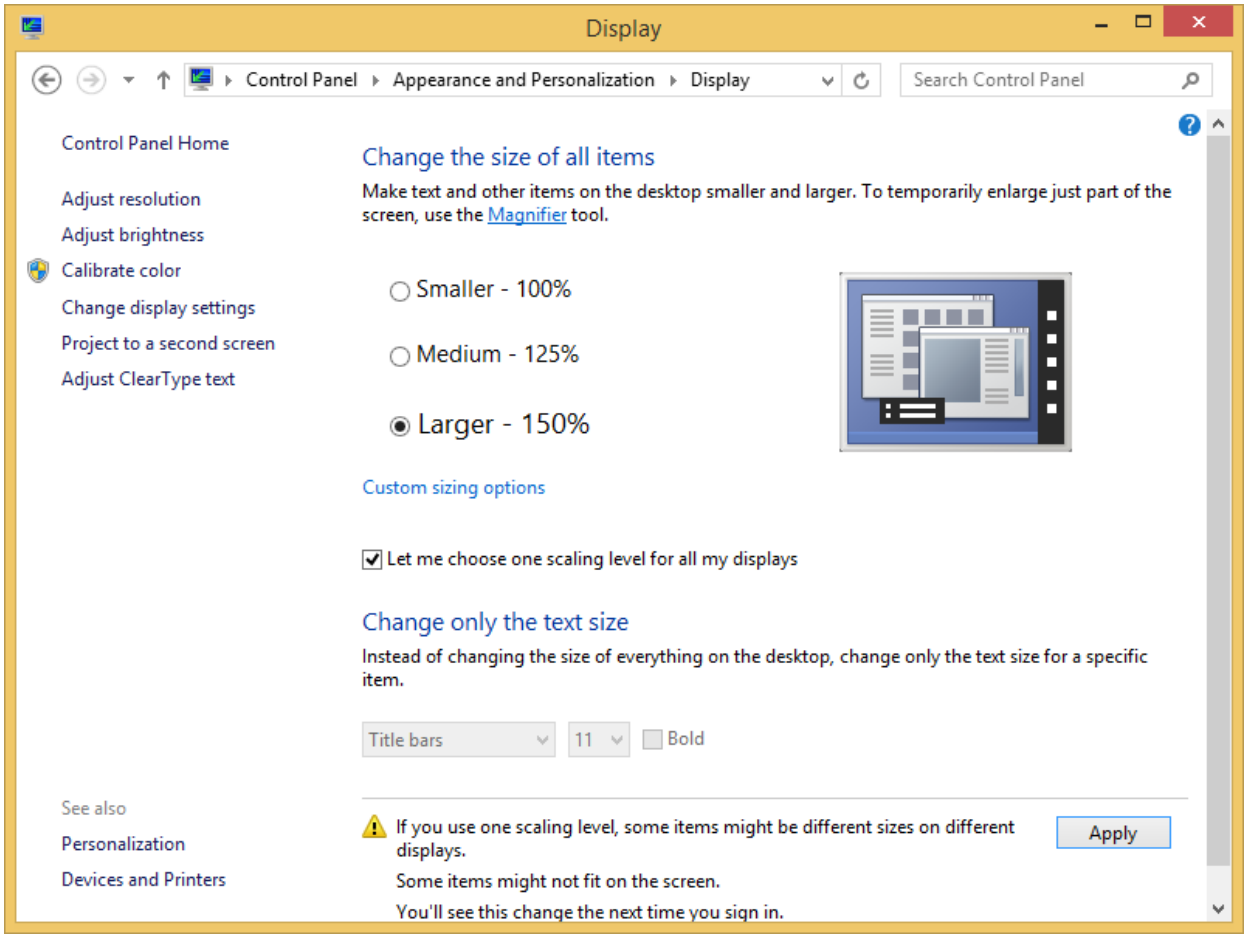
When working with multiple monitors of vastly different sizes and resolutions, it's a pain to have one uniform system DPI value. Think of a tablet PC with an external display. The same DPI setting will rarely fit all displays in your setup. Windows 8.1 attempts to improve this and introduces a new multi-monitor DPI scaling feature.

In Windows 8.1, the user can just choose between making everything "smaller" or "larger" by moving a slider. This roughly represents the percentage steps that one could choose before. The default is "smaller", which represents 96 DPI. If you move the slider to the right, Windows will increase the DPI on those displays of your multi-monitor setup, that it thinks need a higher DPI value. Unfortunately, it doesn't let you set a dedicated DPI value for every monitor.



DPI settings in Windows 8.1, default dialog

To switch off different DPI scales on multiple monitors, you need to tick the box "**Let me choose one scaling level for all my displays**". This brings the old dialog back and will work on multiple monitors as earlier Windows versions did:



Windows 8.1 with unified DPI settings for multiple monitors

Exploring per-monitor DPI scaling

Well, let's see how the new per-monitor scaling works and how it affects our applications.

My setup is a **12" HP Elite slate** with a relatively low native screen resolution of 1280 x 800. It connects to an **external 27" HD display with 1920 x 1080**. When both are next to each other on my desk, the screen of the HP tablet is too far and hard to read at 96 DPI. I prefer to set it to 120 DPI (125%) and leave the huge 27" display at 96 DPI, because that one's really big enough to read everything comfortably. Anything bigger than 96 DPI would be overkill and eat into my available desktop space.

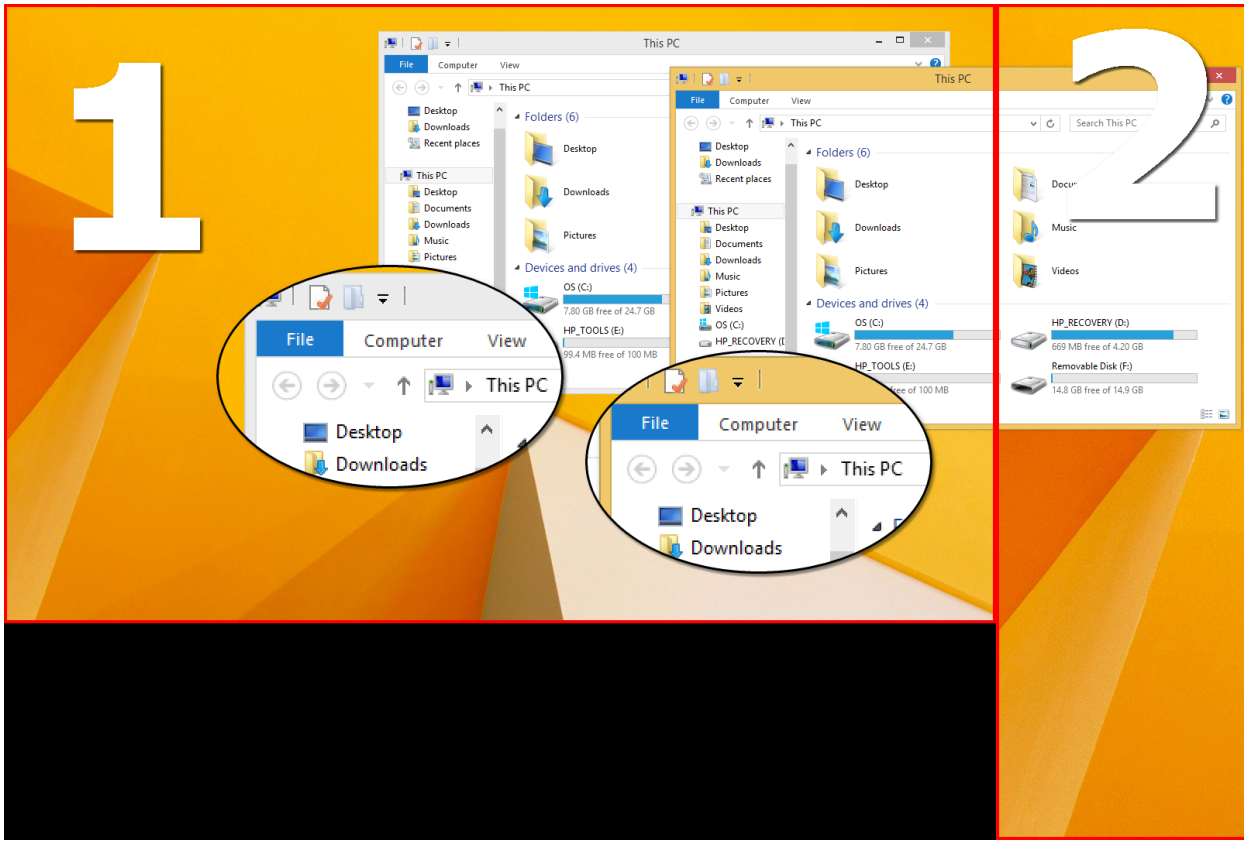
Windows 8.1 decided to run the two display with the following DPI settings (there is no option do deliberately change it per display):

	Windows 8.1 uses	How I would prefer it
HP Elite tablet 12", 1280 x 800	96 DPI (100%)	120 DPI
External screen 27", 1920 x 1080	120 DPI (125%)	96 DPI

Hm... this is pretty much the opposite of what I expected it to do and makes the feature rather useless for this hardware configuration.

Okay, illogical software decisions put aside, what happens on the screen and, more importantly, how does Windows accomplish the display on multiple monitors with different DPI settings?

It does it - again - by *DPI virtualization*. Unless the application declares itself **per-monitor DPI-aware**.



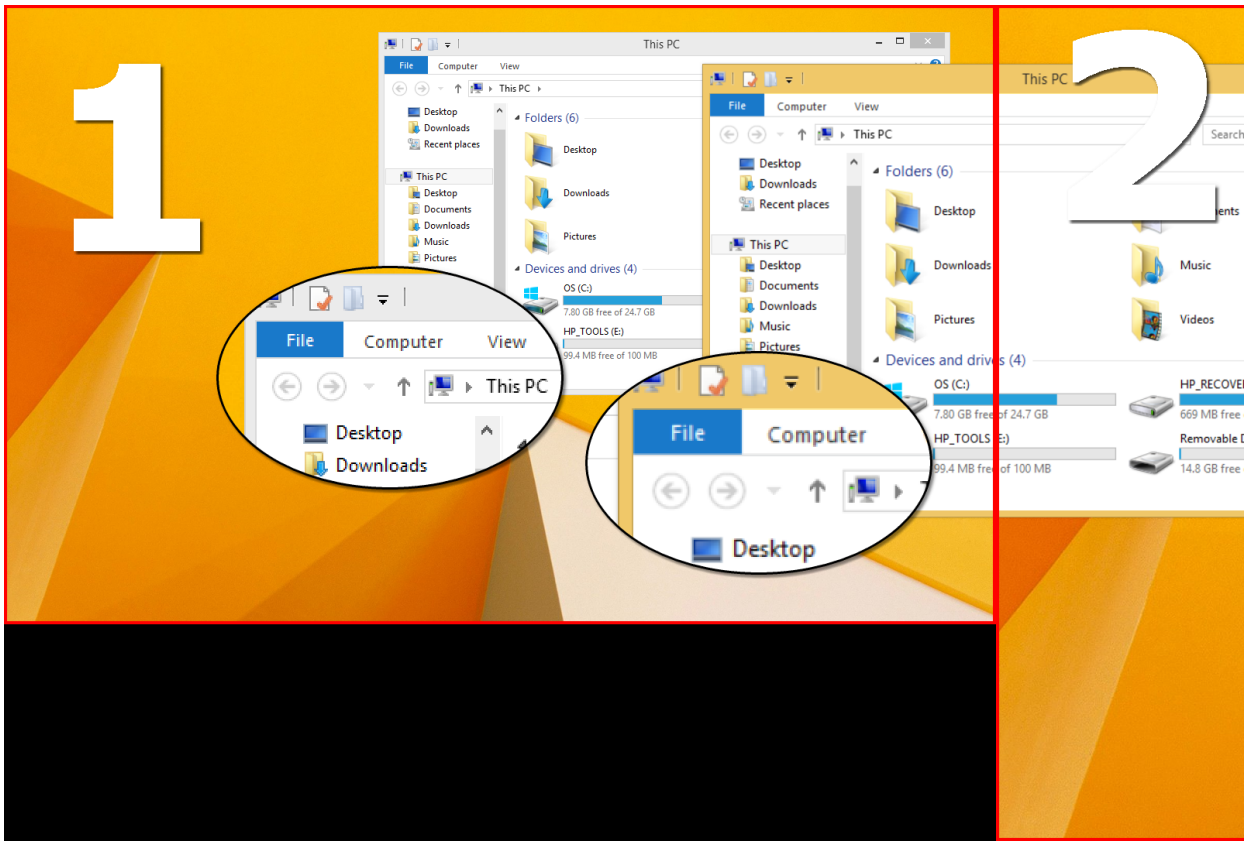
Here are 2 Explorer windows side by side, both at 96 DPI

The screenshot above shows two Explorer windows side by side, the second window is still on the left screen (left: 96 DPI, right: 120 DPI).

When this window is moved to the right display, it keeps its resolution until it appears mostly

on the other screen. When you keep moving it, it changes at a certain point and scales to 120 DPI, the resolution of the right screen. This is shown in the screenshot below.

Take a close look at the enlarged details in the screenshot below: the left Explorer window is still native at 96 DPI, it's sharp and crisp. The right Explorer window has been virtualized to 120 DPI and appears slightly blurred. The Windows 8.1 core components are yet not per-monitor DPI-aware.



Here are 2 Explorer windows side by side, one at 96 DPI, the other already at 120 DPI upscaled

Summary Table - High DPI Windows Features					
Feature	**Windows XP**	**Windows Vista**	**Windows 7**	**Windows 8**	**Windows 8.1**
DPI Virtualization of DPI-unaware apps	No	Yes	Yes	Yes	Yes
DPI Virtualization of System DPI-aware	No	No	No	No	Yes

Summary Table - High DPI Windows Features					
Feature	**Windows XP**	**Windows Vista**	**Windows 7**	**Windows 8**	**Windows 8.1**
apps					
API to declare DPI awareness level	No	Yes	Yes	Yes	Yes
API to declare per-monitor DPI awareness	No	No	No	No	Yes
APIs to retrieve system metrics and DPI	Yes	Yes	Yes	Yes	Yes
Window notification of DPI change	No	No	No	No	Yes
APIs to retrieve monitor DPI	No	No	No	No	Yes
Requires reboot for monitor DPI change	N/A	N/A	N/A	N/A	No
Requires a reboot/log off for system DPI change	Reboot	Reboot	Log off	Log off	Log off
Per user DPI setting	No	No	Yes	Yes	Yes
Auto configuration of DPI at first logon	No	No	Yes	Yes	Yes
Viewing distance incorporated in default DPI calculation	No	No	No	No	Yes

Technical Overview

1

Quick Summary

Windows Vista, 7 and 8 use DPI-virtualization for applications that do not declare "DPI-awareness". Windows 8.1 extends the DPI awareness and introduces two levels:

- System DPI-awareness - applies if just one monitor is used or all monitors have the same DPI setting.
- Per-monitor DPI-awareness - applies if more than one monitor with a different DPI setting is used.

Getting your application ready for 4K

2

Declaring DPI-awareness

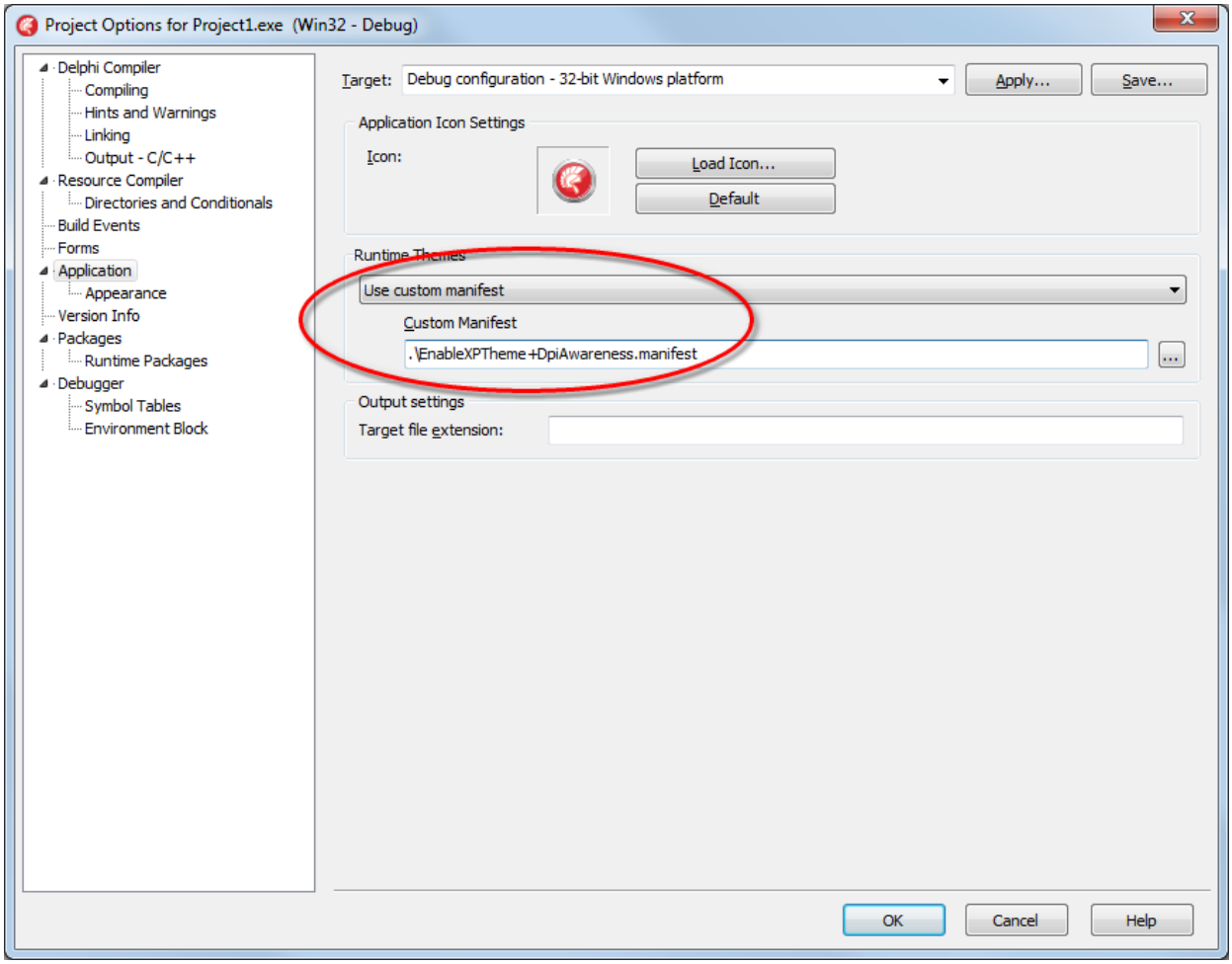
The easiest way to declare DPI-awareness is to **include an application manifest**. As explained in the previous chapter, Windows 8.1 introduced multiple DPI-awareness *levels*.

Windows DPI awareness levels			
Method of declaration	**Windows XP**	**Vista, 7 and 8.0**	**Windows 8.1**
Application manifest	ignored	Yes	Yes
API call to declare system DPI-awareness	not supported	Yes	Yes
API call to declare per-monitor DPI-awareness	not supported	not supported	Yes

DPI-awareness with application manifest

Recent Delphi editions have a setting in the Project Options dialog to include custom application manifests. If you are using an older Delphi version, you might need to compile the manifest into a resource first and include the resource by code.

Getting your application ready for 4K



Custom manifest in Delphi

The manifest part that declares **system DPI-awareness**, is:

```
<asmv3:application xmlns:asmv3="urn:schemas-microsoft-com:asm.v3">
  <asmv3:windowsSettings
       xmlns="http://schemas.microsoft.com/SMI/2005/WindowsSettings">
    <dpiAware>true</dpiAware>
  </asmv3:windowsSettings>
</asmv3:application>
```

If you want to declare **per-monitor DPI-awareness on Windows 8.1** (plus system DPI-awareness on Vista, 7 and 8.0), use the extended value "**True/PM**" for <dpiAware>:

```
<asmv3:application xmlns:asmv3="urn:schemas-microsoft-com:asm.v3">
  <asmv3:windowsSettings
       xmlns="http://schemas.microsoft.com/SMI/2005/WindowsSettings">
    <dpiAware>true/PM</dpiAware>
```

```
    </asmv3:windowsSettings>
  </asmv3:application>
```

But you usually want to enable Windows XP styles as well, so it makes sense to include a little bit more into the manifest. Here is the complete manifest - this version declares *system DPI-awareness* on all Windows editions before 8.1 and *per-monitor DPI-awareness* on Windows 8.1:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <dependency>
    <dependentAssembly>
      <assemblyIdentity
        type="win32"
        name="Microsoft.Windows.Common-Controls"
        version="6.0.0.0"
        publicKeyToken="6595b64144ccf1df"
        language="*"
        processorArchitecture="*"/>
    </dependentAssembly>
  </dependency>
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">
    <security>
      <requestedPrivileges>
        <requestedExecutionLevel
          level="asInvoker"
          uiAccess="false"/>
        </requestedPrivileges>
    </security>
  </trustInfo>
  <asmv3:application xmlns:asmv3="urn:schemas-microsoft-com:asm.v3">
    <asmv3:windowsSettings
        xmlns="http://schemas.microsoft.com/SMI/2005/WindowsSettings">
      <dpiAware>True/PM</dpiAware>
    </asmv3:windowsSettings>
  </asmv3:application>
</assembly>
```

Getting your application ready for 4K

2

Quick Summary

By including an application manifest, you can either declare *system DPI-awareness* or (Windows 8.1 only) *per-monitor DPI-awareness*. The extended version for Win 8.1 also declares system DPI-awareness on older Windows versions that did not support per-monitor DPI-awareness.

Though you can also declare DPI-awareness at runtime by calling the Windows API, there is no good reason to use API calls. The manifest is way simplier.

It does, however, matter whether you declare *system* or *per-monitor* DPI-awareness. For the extended version, your application should be able to respond to the runtime notification from Windows and switch its scale dynamically.

Getting your application ready for 4K

Flipping DPI at runtime

Delphi, even the most recent Delphi XE7 (at the time I write this), is not prepared for *per-monitor* DPI-awareness. To make this happen, we need to implement several things ourselfs. The manifest explained in the last chapter declares our application *per-monitor DPI-aware*, so Windows 8.1 is happy and lets us have our way, even on multiple monitors with different DPI settings.

How is this going to work?

1. On start, we declare DPI-awareness, which in turn will make Windows tell Delphi the true DPI setting and not a virtualized one. Without that, Delphi thinks we're running on 96 DPI, no matter what the setting really is.

2. Delphi will rescale every form, *when the form is created*. It will not do anything after that.

3. When the user moves the application form to another monitor with a different DPI setting, nothing will happen. The window will stay at exactly the same size. Windows doesn't feel responsible for virtualizing our form, because we declared DPI-awareness. Delphi isn't responsible, either. It did its job when the form was created.

The WM_DPICHANGE message handler

In order to automatically respond to DPI changes on different monitors, we have to implement a message handler and respond to WM_DPICHANGED (follow link to MSDN for details). This message is received when the user moves the form to another monitor (with a different DPI value).

The WPARAM of WM_DPICHANGED contains the new DPI value, separate for the X and Y axis. I don't know if there is a monitor with different DPI for the horizontal and vertical axis, but Delphi only uses vertical differences for scaling, so we will ignore the horizontal X value.

Our naked per-monitor DPI-aware form looks like this:

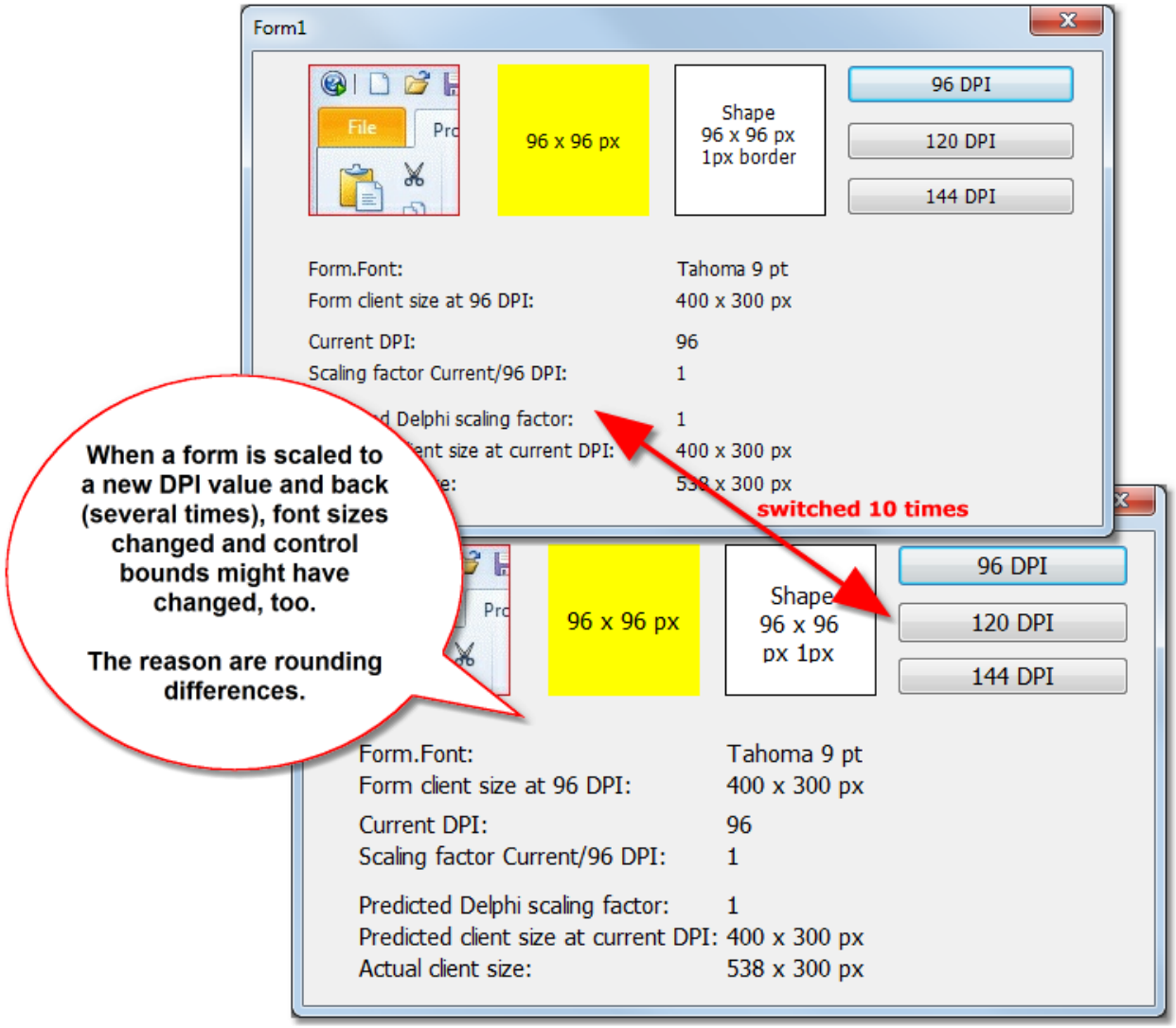
```
unit Unit1;

interface

uses
  Winapi.Windows, Winapi.Messages, Vcl.Controls, Vcl.Forms;

const
  WM_DPICHANGED = 736; // 0x02E0

type
  TForm1 = class(TForm)
  private
    procedure WMDpiChanged(var Message: TMessage); message WM_DPICHANGED;
  public
```

```
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

{-$DEFINE DELPHI_STYLE_SCALING}
procedure TForm1.WMDpiChanged(var Message: TMessage);

  {$IFDEF DELPHI_STYLE_SCALING}
  function FontHeightAtDpi(aDPI, aFontSize: integer): integer;
  var
    tmpCanvas: TCanvas;
  begin
    tmpCanvas := TCanvas.Create;
    try
      tmpCanvas.Handle := GetDC(0);
      tmpCanvas.Font.Assign(self.Font);
      tmpCanvas.Font.PixelsPerInch := aDPI; //must be set BEFORE size
      tmpCanvas.Font.size := aFontSize;
      result := tmpCanvas.TextHeight('0');
    finally
      tmpCanvas.free;
    end;
  end;
  {$ENDIF}

begin
  inherited;
  {$IFDEF DELPHI_STYLE_SCALING}
  ChangeScale(FontHeightAtDpi(LOWORD(Message.wParam), self.Font.Size),
              FontHeightAtDpi(self.PixelsPerInch, self.Font.Size));
  {$ELSE}
  ChangeScale(LOWORD(Message.wParam), self.PixelsPerInch);
  {$ENDIF}
  self.PixelsPerInch := LOWORD(Message.wParam);
end;

end.
```

I have implemented 2 formulas for scaling the form at runtime. My default method is proportional to the difference of the DPI value. When switched from 96 to 120 DPI, the form will grow by exactly 25% in width and height. Delphi scales a form by the difference of the form's font height.

However, you will experience rounding differences. Delphi's `ChangeScale()` function uses `MulDiv()` to calculate the new size and *truncates the decimals*. This is not an issue when scaling the form once, but it becomes an issue when scaling the same form multiple times, up and down. That's what happens when you move a form from one monitor to another with different DPI settings and back.

The resulting form after several switches (despite we're back to 96 DPI where we started from) looks different than the original form:



Differences after rescaling the form several times

Getting your application ready for 4K

2

Quick Summary

Responding to the WM_DPICHANGED notification is a snap. The difference between Delphi's way of scaling a form (by measuring the height of the font at the target DPI) and simply scaling by the DPI difference is not a problem in real life. It actually works better in most cases, because of less rounding differences. Keep in mind that we can only scale *relative to the current size*. Once a form as been rescaled, we do not know the original absolute reference points anymore.

It makes sense to use the code above as a prototype form and inherit all application forms from that prototype.

Remember to throughly test 3rd party controls, that might still rely on `Screen.PixelsPerInch`, which is not valid anymore.

Getting your application ready for 4K

Inventory of your Application

Well, our application is DPI-aware and it even switches DPI values at runtime when moved from one monitor to the next on Windows 8.1. The `ChangeScale()` procedure nicely changes the form and the controls but it does not change, for instance, icons. There are still a few things we need to check. Let's do an inventory of the changes still required...

Things you can consider safe

Delphi's own controls are mostly safe. Buttons, fonts, edit controls, combo controls, list boxes, Win32 controls such as TListView are basically fine. In my own superficial tests I did not come across any problems with the standard controls. Some minor glitches such as database grids that still separate cells with a 1 px line (even on 200% magnification) are not a problem in practise.

Make sure that any instance of **TImage** used in your application uses the `Streched := true` property. AutoSize must be false, otherwise it will not scale but revert to the pixel x pixel size of the image it holds. If you include a TImage for displaying logos and splash screen backgrounds, you might want to include a higher resolution of that picture and change it at runtime.

Things you need to test

You need to throughly test 3rd-party controls and components, even if they are not visible controls. The main reason is that the global variable `Screen.PixelsPerInch` is set by Delphi at the start of your application. If the application is moved to a different monitor, the setting is not valid anymore. If a component/control relies on that, it is highly likely to fail.

- Treeviews, grids that come with separate header fonts and custom header design. These component should overwrite the ChangeScale() procedure internally to respond to DPI changes.
- Report generators and print previews: test them on high DPI settings and, more importantly, if they are still working on Win 8.1 multi-monitor setups, when the application has been moved to a different monitor. Remember: the global `Screen.PixelsPerInch` is not guaranteed to be identical with `TForm.PixelsPerInch`.
- Controls on the non-client area of a form: 3rd-party buttons that integrate themselves into the caption bar of a form or paint on the NC area in any way. These could fail on high-resolution displays.
- Embedded Active-X and OCX controls need to be aware of the high DPI as well.

Getting your application ready for 4K

Things that don't work out of the box

Image lists

A typical application toolbar or ribbon control uses "small" and "large" images, which are by default 16 x 16 pixels and 32 x 32 pixels, respectively. Stretching icons at runtime is not an option. So you need additional icons in bigger sizes and change them at runtime. Those icons you have in 16 px squared, will also be required in 24 px and 32 px. The 32 px icons require high-resolutions with 48 px and 64 px.

Windows recommended icon sizes		
Resolution	**Small Icon**	**Large Icon**
96 DPI ("normal", 100%)	16 x 16 px	32 x 32 px
>= 144 DPI (150%)	24 x 24 px	48 x 48 px
>= 192 DPI (200%)	32 x 32 px	64 x 64 px

Custom paint handlers for TListbox and TCombobox

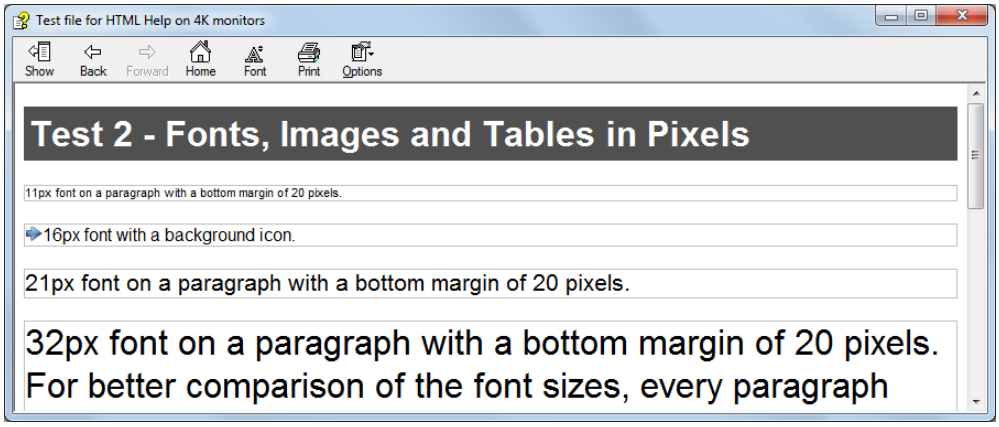
If you have designed your own paint handlers for list boxes or combos, keep in mind to re-calculate item sizes and adjust icons.

Making HTML Help DPI-aware

Most Windows desktop applications include documentation in HTML Help (.chm) format. This is still the standard documentation format on Windows. The Microsoft HTML Help viewer, part of the operating system, is typcially involved by Windows API calls to the HTML Help API. That makes the help window a child process of the application. It inherits the DPI-awareness of the host application.

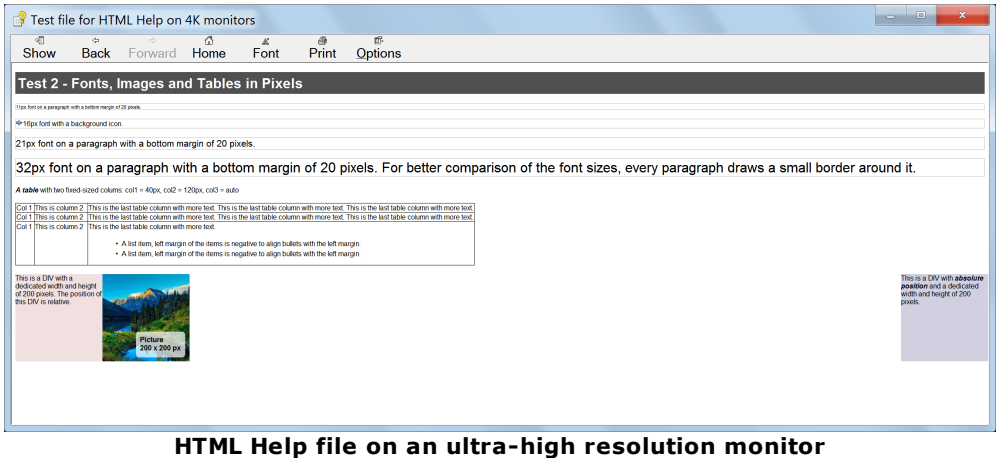
The HTML Help viewer is *somewhat DPI-aware* of high resolution displays. In particular, it scales the UI fonts relative to the screen DPI. Icons are not scaled, which looks a bit odd. But the actual problem comes from the fact that the embedded HTML Help viewer treats HTML pixel-values as *hardware* pixels and *not as virtual* pixels like a stand-alone browser does.

This is what you see when you view a CHM help file on your standard display:



HTML Help file on a regular monitor

And this is what the customer sees on a hi-res display (both help windows are approximately the same size):



HTML Help file on an ultra-high resolution monitor

Getting your application ready for 4K

2

Quick Summary

Getting HTML Help (.chm) files ready for 4K monitors is a different story. I have covered this in a separate article:

Zooming in on HTML Help: How to Make CHM Files Ready for 4K Displays

Download link:
http://download.ec-software.com/getting-html-help-ready-for-4k-monitors.zip

Summary

Though for many years there was not much progress on the Windows desktop world regarding high resolutions (presumably because Windows is so weak at scaling properly?), recent hardware developments have boosted this technology. We are quickly moving towards a *retina world*. It's high on time to adjust your application and make it aware of the new capabilities.

Sure, you can lean back and rely on Windows DPI virtualization. Many users won't even notice, because they don't know better. But it's like new smartphones and iOS apps not prepared for taking advantage of high-resolution display: they look blurred and old, when viewed next to an app that is prepared. Sure as fate, the same will happen to Windows desktop applications over the next few (fewer) years. If you don't want your application to look like a 16 bit program on a 32 bit OS, take action now!

System DPI-awareness

System DPI-awareness is the minimum requirement for all applications. That's what you need to shine on a 4K display. It doesn't require Windows 8.1+ but is already available with Windows 7. Making your application system DPI-aware is the first step.

Per-monitor DPI-awareness

Is a new mode introduced with Windows 8.1: the DPI value changes when a form is moved to a different monitor. Implementing per-monitor DPI-awareness is a more difficult task for real world application, because it affects each and every 3rd party control and component that you use in your application. If you don't have the source (think OCX controls hosted in a Delphi form), you might even be blocked from upgrading your application until the 3rd party vendor has fixed it.

At the moment, even Windows 8.1 core components do not take advantage of per-monitor DPI-awareness. In other words: when you are running Win 8.1 with multiple monitors at default settings, you will be looking at a blurry world and don't know better. This is an argument for not declaring per-monitor DPI-awareness! If you just declare system DPI-awareness, your application will look as bad as Windows 8.1 itself, but not worse.

However, don't rest assured, the API is here, it is a step into the right direction and Windows 10 is just around the corner. I bet it will deliver what Windows 8.1 does not.

Links

4

Note

If you received this eBook in PDF format only and it did not include the Delphi example application source code, please download the **complete package from this link**:

http://download.ec-software.com/delphi-developers-guide-4k.zip

Furthermore, have a look at the second article in this series:

Zooming in on HTML Help: How to Make CHM Files Ready for 4K Displays

http://download.ec-software.com/getting-html-help-ready-for-4k-monitors.zip

The following links were helpful when writing this guide (in no particular order):

http://blogs.windows.com/bloggingwindows/2013/07/15/windows-8-1-dpi-scaling-enhancements/
Windows blog - Windows 8.1 DPI Scaling Enhancements

http://www.anandtech.com/show/7939/scaling-windows-the-dpi-arms-race/5
AnandTech - Scaling Windows - The DPI Arms Race

http://www.virtualdub.org/blog/pivot/entry.php?id=384
virtualdub.org - Implementing per-monitor DPI awareness

http://msdn.microsoft.com/de-at/magazine/dn574798.aspx
MSDN blog - Write High-DPI Apps for Windows 8.1

http://msdn.microsoft.com/en-us/library/windows/desktop/hh447398(v=vs.85).aspx
MSDN - High DPI Reference