High-DPI Cookbook for Visual Studio 2013

This cookbook contains guidance for testing and fixing High-DPI issues in Visual Studio for the Visual Studio 2013 release.

# Quick Introduction

* Windows 8.1 will target 150 and 200% DPI scaling out of the box.
* Windows 8.1 will scale Visual Studio up or down (to an equivalent of 75% and 50% of current UI) when projecting on secondary display when it’s DPI differs from the primary display
* Microsoft’s Surface Pro now ships with a default of DPI scaling 150%, as do several other netbook form factor Windows 8 and 8.1 devices.
* Running Windows on a high pixel dense display needs to work with a DPI scaling of 150 – 200% to be useful.

There is an introductory talk about what this means for Windows users and developers from the [build conference.](http://channel9.msdn.com/Events/Build/2013/4-184)

# Testing your extension with Visual Studio in High-DPI

Throughout the Visual Studio 2013 timeframe, Visual Studio teams completed test passes through their feature areas and user interfaces in both 150% and 200% (using the below DPI and resolution guidelines).

Visual Studio teams also completed performance, responsiveness and stress validations in their areas if applicable based on the issue type. These criteria are more likely to be applicable if the fix applies to displaying a series of icons (i.e. Solution Explorer) than if it only applies to one (I.e. Extension Manager Button).

## Machines and Configuration

Use the following DPI and Resolution guidance to complete high-DPI testing.

Windows 8.1 will require resulting 720 logical px vertical space to allow selecting a DPI zoom factor. That translates to 1080px at 150% zoom and 1440px at 200% zoom.

|  |  |  |
| --- | --- | --- |
| DPI | Minimum Resolution | Ideal Resolution |
| 150% | 1920 x 1080 | 2880 x 1800 or higher |
| 200% | 2560 x 1440 | 2880 x 1800 or higher |

The best way to test this is on the real hardware. This allows the user to see application content at the ideal viewing distance. 16x16 images in Visual Studio will appear acceptable on a 96 DPI device, but users will find them to be unusable at over 200 DPI because the images will be too small at the ideal viewing distance.

While it’s possible to emulate high-DPI scenarios by scrolling the desktop on older devices, we do not recommend using this method for testing because we find it masks the ability to identify images that are too small to use at the ideal viewing distance.

## New Windows 8.1 DPI Scaling Enhancements

In Windows 8.1, Windows now recognizes the separate DPI settings for each monitor. With this feature, Visual Studio users can use Visual Studio with a mix of low and high density screens and always have Visual Studio appear to be the correct size on each display. An example of this was used in our Visual Studio labs to test the use of mixed DPIs with multiple displays:

**Display 1:** Toshiba Kirabook (running Windows 8.1 + latest build of Visual Studio at 150%/200%)

**Display 2:** 17” low res external display (running at 100%)

For additional context on what multi-mon DPI actually means for Windows users, [read this blog post](http://blogs.windows.com/windows/b/extremewindows/archive/2013/07/15/windows-8-1-dpi-scaling-enhancements.aspx).

**Recommended method for multi-mon DPI testing in Visual Studio:**

1. Start on high-DPI display at 150% (Display 1) and proceed with a single/step validation
2. Then drag Visual Studio to second low density display (Display 2)
3. If the validation involves a dialog, tool windows, or piece of moveable UI, drag it to second display
4. Repeat validation at 200%

**Keep an eye out:** make sure all windows, etc. render properly as you drag them over onto the second display. While Visual Studio 2013 is not natively per-display DPI aware, you may have issues that need to be addressed.

# Visual Studio Big Wins & Fixes

Visual Studio teams fixed may important high-DPI issues to create an improved experience for users with high density displays over previous versions. The following are a few examples of areas we’ve addressed in Visual Studio 2013 based on customer feedback and internal test passes.

|  |  |
| --- | --- |
| ***Project Properties in Managed Languages***  We set AutoScaleMode.Font for the Build Events property page to make it scale better on high DPI. | cid:image006.jpg@01CE4827.CF384960 |
| ***Solution Explorer***  Solution explorer icons become difficult to discern at on high-density display. This happened even in WPF due to custom drawing routines that didn’t take into account the scaling factor. | cid:image007.jpg@01CE4827.CF384960 |
| ***Toolbox Icons***  Toolbox icons were too small. The team used the GDI+ classes to scale up this UI built on Win32 and GDI. | cid:image008.jpg@01CE4827.CF384960 |
| **F12 and Diagnostics**  Functional icons did not scale. | cid:image010.jpg@01CE4827.CF384960 |
| **WinForms**  Clicking on these was a pain in testing. The WinForms team scaled up the images to the right size. | HighDPIStateImages |
| **WinForms**  Tool Strip Container could not be designed in WinForms Designer. | cid:image019.jpg@01CE48E3.086A3FC0becomes… cid:image020.jpg@01CE48E3.086A3FC0 |
| **VC++**  The binary editor now supports High-DPI by scaling the font | cid:image043.jpg@01CE4827.CF384960 |

# Fixing High-DPI Bugs

Windows™ already has a way for an application to tell the system it is DPI scaling aware. Visual Studio 2013 opts-in to being “system DPI aware” and therefore is not “virtualized”. [Documentation is available on MSDN.](http://msdn.microsoft.com/en-us/library/windows/desktop/dd464660(v=vs.85).aspx) Visual Studio 2013 is not per-display DPI aware.

## What is already scaled up?

* WPF already measures controls in device independent manner. Units are specified in terms of 96 dpi, and WPF automatically scales to the currently selected system DPI.
* Text sizes are expressed in “points”, not pixels, so they are dpi independent. Texts rendered using GDI or any other technology will be scaled up correctly.
* Win32/WinForms dialogs and windows that don’t have hardcoded sizes will be scaled up, calculated based on enlarged font size. Layouts are best computed relative to the content and laid out automatically.
* Icons loaded from system or resources based on system metrics (SM\_CXICON, SM\_CXSMICON) are already scaled-up . However, if a frame of suitable size is not found in the original icon, the closest frame size is stretched using NearestNeighbour algorithm, potentially causing distortions.

## Win32 (GDI, GDI+) UI

* While WPF is already High-DPI aware and scales up automatically, Win32/GDI based UI needs to be written with DPI awareness in mind in order to scale up correctly in High-DPI modes. While some Win32 UI elements scale automatically for DPI (fonts, icons loaded with default sizes), in other cases DPI settings have to be explicitly retrieved and controls sizes/positions, or images have to be explicitly scaled up. Example of getting system information and scaling up control sizes are provided in [this MSDN article](http://msdn.microsoft.com/en-us/library/windows/desktop/dd464660(v=vs.85).aspx#getting_system_information).

## WinForms UI

* WinForms controls can use the Grid control and others to ensure proper layout at all sizes. Some WinForms controls like ContainerControl also have an AutoScaleMode property that can be set to Font or Dpi to enable automatic scaling. Other controls have AutoSize property that can be set to true to scale the control based on its content (e.g. a Button can auto-scale based on its text, since the font size respects the current DPI). Caution should be used to avoid double-scaling of controls.
* Other WinForms controls like DataGrid, TreeView node images, scale up for High-DPI when the following property is set in application.config file (Visual Studio 2013 already sets this property).

<configuration>

<appSettings>

<add key="EnableWindowsFormsHighDpiAutoResizing" value="true" />

</appSettings>

</configuration>

* When automatic scaling is not possible, controls dimensions, spacing, or images have to be explicitly scaled up for High DPI, like in Win32 case.

## High-Resolution Assets

* Visual Studio 2013 does not have built-in support for selecting different images for different DPI seetings. Images are generally “scaled-up” at runtime to match the DPI scaling of the system.

## Images too small

* Image resources provided to Visual Studio and your own windows will potentially need to be “scaled up”. E.g. a 16x16 pixel icon that is handed off to a TreeViewControl will need to be resized to 32x32 on a system DPI scaling of 200%.
* The VS Platform team has a common way to “scale-up” images to be rendered on GDI or WPF:
  + Managed / Native [DPIHelper classes](#_Using_the_DPIHelper)  - Support scaling Icons, Bitmaps, imagestrips, and ImageLists for current DPI; for native code, support scaling of HICON, HBITMAP, HIMAGELIST, VsUI::GdiplusImage(wrapper over Gdi+ Bitmap)
  + Scaling of a GDI bitmap usually requires only one line change

VsUI::DpiHelper::LogicalToDeviceUnits(&hBitmap);

DpiHelper.LogicalToDeviceUnits(ref image);

* + Scaling of imagelists requires either a call to LogicalToDeviceUnits() with an imagelist, or scale an individual bitmap first before constructing the imagelist it will be inserted into. For example

imagelist.ImageSize = DpiHelper.LogicalToDeviceUnits(imagelist.ImageSize)

In native code, scale the dimensions when creating the imagelist

ImageList\_Create(VsUI::DpiHelper::LogicalToDeviceUnitsX(16), VsUI::DpiHelper::LogicalToDeviceUnitsY(16), ILC\_COLOR32|ILC\_MASK, nCount, 1);

* Functions in the library allow specifying the resizing algorithm. Unless you have some specific requirements for the algorithm, use the default arguments values and let the shell pick up the appropriate resizing algorithm depending on current DPI scale factor.

## Layout Issues

* Layout/text positions will have to be adjusted to account for scaled up images.
* Columns in grids will need to have widths adjusted for the scaled up text. Using the font size instead of pixels will often yield the best results.
* UI element spacing will also need to be scaled up; sizes based on text dimensions will work fine (texts are automatically scaled up).
* Again, developers can use Visual Studio’s [DpiHelper class](#_Using_the_DPIHelper):
  + LogicalToDeviceUnitsX/LogicalToDeviceUnitsY (functions allow scaling on X/Y axis)

int space = DpiHelper.LogicalToDeviceUnitsX (10);

int height = VsUI::DpiHelper::LogicalToDeviceUnitsY(5);

* + LogicalToDeviceUnits
  + (Overloads allow scaling Rect, Point, Size, etc.)

## Image Fuzziness

* Considerations:
  + For logo image & banners artwork, the Bicubic-based resizing mode could be used at any zoom level; this works well for pictures.
  + For menu items & iconography images, the Bicubic algorithm introduces fuzziness. A NearestNeighbor resizing algorithm is preferred instead when it doesn’t cause images distortions (e.g. at 200%, 300% zoom levels).

## In WPF

* Images are resized by WPF by default using a Bicubic algorithm, which introduces fuzziness. Don’t let WPF use default unless you have UI that has zoom levels independent on the shell’s zoom level (e.g. editor)
* The DpiHelper class in Microsoft.VisualStudio.PlatformUI namespace has a member BitmapScalingMode that should be used for binding. It allows the shell to control in a uniform way across the product the bitmap scaling mode (dependent on DPI zoom factor), and allows the user to override via registry settings.

To use it in xaml, add:

xmlns:vsui="clr-namespace:Microsoft.VisualStudio.PlatformUI;assembly=Microsoft.VisualStudio.Shell.12.0"

<Setter Property="RenderOptions.BitmapScalingMode" Value="{x:Static vs:DpiHelper.BitmapScalingMode}" />

The shell sets this property already on top-level windows and dialogs, so chances are your WPF UI already inherits it. If the setting does not propagate to all your UI, set it on root elements in elements on Popups, on elements placed as children in Win32/Winforms parents (HwndHost.RootVisual, ElementHost.Child), etc.

## Win32/WinForms Images Fuzziness

* Win32 functions (StretchBlt, etc) use by default a NearestNeighbor algorithm which causes image distortions at certain zoom levels (125, 150%). This should be avoided.
* Winforms use by default Bilinear which causes fuzziness.
* For consistency, the image resizing functions in DPIHelper class (native and managed) have an argument of type ImageScalingMode that allows the shell to control the resizing algorithm, depending on the DPI zoom level. Functions like LogicalToDeviceUnits() in DpiHelper have an argument of type ImageScalingMode with default value; use the default value of this argument when resizing images unless you have some specific requirements of using an explicit algorithm such as Bicubic.

## DPI Scaling and image theming/inversion

* Sometimes, displayed images need to be themed using IVsUIShell5.ThemeDIBits or CreateThemedImageList().
* For better results, do the DPI Scaling before theming an image

## Icons scaling

* Windows APIs like LoadIcon(), ExtractIconEx(), ShGetFileInfo(), etc. extract icons using sizes given by SM\_CXICON, SM\_CYICON, SM\_CXSMICON, and SM\_CYSMICON system metrics. These are already scaled up for High DPI. However, if an icon frame of exact size is not found in the resource or the registered icon, the frame of closest size is scaled up using NearestNeighbor algorithm, which could introduce artifacts.
* Therefore, it’s best to have the icons with frames of 16x16, 20x20, 24x24, 32x32 sizes
* Loading an icon frame of exact/desired size can be done with LoadImage(IMAGE\_ICON) and specifying the desired size. Such image can be later scaled with DpiHelper for the current HDPI level.

## Using the DPIHelper library/class

* **In native code**
  + You should take the source code included with this cookbook and created your own UI utilities library. You are free to change the source to fit your situation as required.
  + Sample code for scaling images based on the included source code could look like this call of static functions of VsUI::DpiHelper class:

#include "VsUIDpiHelper.h"

int cxScaled = VsUI::DpiHelper::LogicalToDeviceUnitsX(cx);

VsUI::DpiHelper::LogicalToDeviceUnits(&hBitmap);

* **In managed code**
  + The project must reference latest version of Shell MPF

<Reference Include="Microsoft.VisualStudio.Shell.12.0" />

* + Depending on your project, if you get compile errors you may also need to add references to System.Windows.Forms, PresentationCore, PresentationUI
  + In code, use the PlatformUI namespace and call static functions of DpiHelper class. For supported types (points, sizes, rectangles, etc.) there are provided extension functions that return new scaled objects.

using Microsoft.VisualStudio.PlatformUI;

double x = DpiHelper.LogicalToDeviceUnitsX(posX);

Point ptScaled = pntOriginal.LogicalToDeviceUnits();

DpiHelper.LogicalToDeviceUnits(ref bitmap)

Adding HDPI Support to the WebOC

* By default, WebOC controls (such as the WebBrowser control in WPF, or the IWebBrowser2 interface) don’t try to support HDPI.
* To make a WebOC control start rendering content correctly in HDPI, you need to perform the following:
  + - 1. Implement the IDocHostUIHandler interface (See the MSDN documentation)

[ComImport,

InterfaceType(ComInterfaceType.InterfaceIsIUnknown),

Guid("BD3F23C0-D43E-11CF-893B-00AA00BDCE1A")]

public interface IDocHostUIHandler

{

[return: MarshalAs(UnmanagedType.I4)]

[PreserveSig]

int ShowContextMenu(

[In, MarshalAs(UnmanagedType.U4)] int dwID,

[In] POINT pt,

[In, MarshalAs(UnmanagedType.Interface)] object pcmdtReserved,

[In, MarshalAs(UnmanagedType.IDispatch)] object pdispReserved);

[return: MarshalAs(UnmanagedType.I4)]

[PreserveSig]

int GetHostInfo(

[In, Out] DOCHOSTUIINFO info);

[return: MarshalAs(UnmanagedType.I4)]

[PreserveSig]

int ShowUI(

[In, MarshalAs(UnmanagedType.I4)] int dwID,

[In, MarshalAs(UnmanagedType.Interface)] object activeObject,

[In, MarshalAs(UnmanagedType.Interface)] object commandTarget,

[In, MarshalAs(UnmanagedType.Interface)] object frame,

[In, MarshalAs(UnmanagedType.Interface)] object doc);

[return: MarshalAs(UnmanagedType.I4)]

[PreserveSig]

int HideUI();

[return: MarshalAs(UnmanagedType.I4)]

[PreserveSig]

int UpdateUI();

[return: MarshalAs(UnmanagedType.I4)]

[PreserveSig]

int EnableModeless(

[In, MarshalAs(UnmanagedType.Bool)] bool fEnable);

[return: MarshalAs(UnmanagedType.I4)]

[PreserveSig]

int OnDocWindowActivate(

[In, MarshalAs(UnmanagedType.Bool)] bool fActivate);

[return: MarshalAs(UnmanagedType.I4)]

[PreserveSig]

int OnFrameWindowActivate(

[In, MarshalAs(UnmanagedType.Bool)] bool fActivate);

[return: MarshalAs(UnmanagedType.I4)]

[PreserveSig]

int ResizeBorder(

[In] COMRECT rect,

[In, MarshalAs(UnmanagedType.Interface)] object doc,

bool fFrameWindow);

[return: MarshalAs(UnmanagedType.I4)]

[PreserveSig]

int TranslateAccelerator(

[In] ref MSG msg,

[In] ref Guid group,

[In, MarshalAs(UnmanagedType.I4)] int nCmdID);

[return: MarshalAs(UnmanagedType.I4)]

[PreserveSig]

int GetOptionKeyPath(

[Out, MarshalAs(UnmanagedType.LPArray)] string[] pbstrKey,

[In, MarshalAs(UnmanagedType.U4)] int dw);

[return: MarshalAs(UnmanagedType.I4)]

[PreserveSig]

int GetDropTarget(

[In, MarshalAs(UnmanagedType.Interface)] IOleDropTarget pDropTarget,

[MarshalAs(UnmanagedType.Interface)] out IOleDropTarget ppDropTarget);

[return: MarshalAs(UnmanagedType.I4)]

[PreserveSig]

int GetExternal(

[MarshalAs(UnmanagedType.IDispatch)] out object ppDispatch);

[return: MarshalAs(UnmanagedType.I4)]

[PreserveSig]

int TranslateUrl(

[In, MarshalAs(UnmanagedType.U4)] int dwTranslate,

[In, MarshalAs(UnmanagedType.LPWStr)] string strURLIn,

[MarshalAs(UnmanagedType.LPWStr)] out string pstrURLOut);

[return: MarshalAs(UnmanagedType.I4)]

[PreserveSig]

int FilterDataObject(

IDataObject pDO,

out IDataObject ppDORet);

}

* + - 1. (Optional) Implement the ICustomDoc interface (See the MSDN documentation)

[InterfaceType(ComInterfaceType.InterfaceIsIUnknown),

Guid("3050F3F0-98B5-11CF-BB82-00AA00BDCE0B")]

public interface ICustomDoc

{

void SetUIHandler(IDocHostUIHandler pUIHandler);

}

* + - 1. Associate the class that implements IDocHostUIHandler with the WebOC’s document

If you did #2 above, then as soon as the WebOC’s document property is valid, cast it to an ICustomDoc and call the SetUIHandler method, passing the class that implements IDocHostUIHandler.

// “this” references that class that owns the WebOC control and in this case also implements the IDocHostUIHandler interface

ICustomDoc customDoc = (ICustomDoc)webBrowser.Document;

customDoc.SetUIHandler(this);

If you did NOT do #2 above, then as soon as the WebOC’s document property is valid, you’ll need to cast it to an IOleObject, and call the SetClientSite method, passing in the class that implements IDocHostUIHandler.

* + - 1. Set the DOCHOSTUIFLAG\_DPI\_AWARE flag on the DOCHOSTUIINFO passed to the GetHostInfo method call.

public int GetHostInfo(DOCHOSTUIINFO info)

{

// This is what the default site provides.

info.dwFlags = (DOCHOSTUIFLAG)0x5a74012;

// Add the DPI flag to the defaults

info.dwFlags |=.DOCHOSTUIFLAG.DOCHOSTUIFLAG\_DPI\_AWARE;

return S\_OK;

}

This should be all that you need to get your WebOC control to support HPDI.

**NOTES:**

1. If the document property on the WebOC control changes you may need re-associate the document with the IDocHostUIHandler class.
2. If the above does not work, there is a known issue with the WebOC not picking up the change to the DPI flag. The most reliable way of fixing this that we have found is to toggle (meaning 2 calls with 2 different values for the zoom percentage) the optical zoom of the WebOC. Additionally, if this work around is required, it may be necessary to perform it on every navigate call.

// browser2 is a SHDocVw.IWebBrowser2 in this case

// EX: Call the Exec twice with DPI%-1 and then DPI% as the zoomPercent values

IOleCommandTarget cmdTarget = browser2.Document as IOleCommandTarget;

if (cmdTarget != null)

{

    object commandInput = zoomPercent;

    cmdTarget.Exec(IntPtr.Zero,

OLECMDID\_OPTICAL\_ZOOM,

OLECMDEXECOPT\_DONTPROMPTUSER,

ref commandInput,

ref commandOutput);

}

# Need additional help?

If you need additional help with any of the above, please submit a Connect bug on the Visual Studio Industry Partner site and someone from the Visual Studio IDE team will get in touch with you.