

High-DPI, Subpixel Text Positioning, Hinting

*What happens when an unstoppable bullet hits an impenetrable wall?*¹

<http://goo.gl/yf3M7>

Behdad Esfahbod

behdad@google.com

August 30, 2012

[Introduction](#)

[Linearity and Aliasing](#)

[Constraints](#)

[Non-linear Layout](#)

[Linear Layout](#)

[Problem at Hand](#)

[Solution](#)

[Remove Non-linearity](#)

[Remove Linearity Requirement](#)

[Metrics Hinting Meets Linear Layout](#)

[Other Systems](#)

[Conclusions and Recommendations](#)

[Other Resources](#)

[Acknowledgements](#)

Introduction

When getting text onto the screen one has many choices, and decisions to make:

- Do outline hinting? If yes, what kind of hinting (bytecode or autohinting), in which directions (X and Y, or Y-only), and how aggressively (in each direction!),
- Do metrics hinting? Ie. snap each glyph's width to whole pixels,
- Do antialiasing, which may be grayscale antialiasing or subpixel antialiasing (aka subpixel rendering). If antialiasing, using what gamma for blending? If subpixel, what kind of filter to use,
- Do subpixel text positioning (ie. be able to rasterize glyphs differently based on their sub-pixel origin position).

What you decide to do largely depends on the type of application, layout requirements, display device,

¹ The wall will move with the bullet.

and how much control you have on those. It's best to learn to think of these different knobs as, well, different knobs. You can turn each on / off or otherwise adjust, independently of the others. Some combinations make more sense than others, but in general, you still *can* change them independently.

Note that when most people talk about subpixel text positioning, what they really mean is subpixel text positioning *and* no metrics hinting. Technically, nothing stops you from having subpixel text positioning and still hint metrics. Though that's not what most people mean or do.

It also helps learning to think of text rendering as two separate processes: layout and rendering. Layout is the process of deciding where to show each glyph. Rendering is actually showing glyphs at those positions given all the constraints of the display medium.

Linearity and Aliasing

When it comes to layout, there are two opposite directions you can go: linear, and non-linear. Glyph positions produced by a linear layout function can be transformed by an affine (or even projective, if you are careful) transformation, and they would result in exactly what would have had resulted if the font scale matrix was transformed by such transformation before layout. I.e. linearly laying out a paragraph at 12pt to a width of 4in will result in the exact same look and line breaks that results from setting it at 24pt to a width of 8in. That's a very nice property, because it means that you can zoom, rotate, translate, shear, even project the layout results freely.

Non-linear layout would be different. For example, you may decide (for many legitimate reasons), that at 10px size, the glyph for letter 'i' should take 2 pixels of space (one column of black stem, one column of white space after). But the same glyph, at size 20px may take only 3 pixels (one full black stem in the middle column, and two very light gray columns on the sides). That's clearly non-linear, because although the font size was increased 100%, the glyph width only increased 50%. Note that this is not a matter of local error. If you consider a string of i's ("iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii"), the whole string is now only 50% wider than the one "half the size". Line breaks will be calculated differently, page breaks will be calculated differently, and the document may end up consuming a different number of pages. In short, with non-linear layout all bets are off.

The world would have been a rosy place if we could only care about linear layout. Alas, not in this world. Damn you [Nyquist](#)! Trying frequency limited sampling on linear layout (ie. rendering to pixelated screens) results in [aliasing](#). Almost all of the knobs mentioned earlier have the same mandate: to fight aliasing. Some do that at the expense of non-linearity, some don't. Here's how:

- Hinting modifies the outlines to better match the sampling grid. I.e. transform high frequencies to lower frequencies. Clearly, this is modifying the signal (A.K.A. shape)
- Metrics hinting acts as a low-pass filter. Results in non-linear layout.

- Anti-aliasing acts like a blur, again, a low-pass filter.
- Subpixel rendering increases spatial sampling frequency, pushing up the Nyquist frequency.
- Subpixel text positioning, again, increases the spatial resolution. By resolution I don't mean the sampling frequency here, but how finely you can *resolve* distances. Without it, you cannot resolve (A.K.A. display) non-integral distances (A.K.A. positions). With it, you can.

Constraints

Note that out of all the knobs mentioned above, only one introduces non-linearity into layout: metrics hinting. As we said before, some combinations of the knobs make more sense than others. Here are some of the ways that the various knobs interact with each other:

- If you don't have subpixel text positioning (ie. your graphics system knows only how to rasterize one image for a glyph and blit that onto the screen), then you really want to enable metrics hinting. Otherwise the rounding error from non-integer metrics will result in highly-visible low-frequency patterns like this:



By enabling metrics hinting, you get the much more elegant rendering:



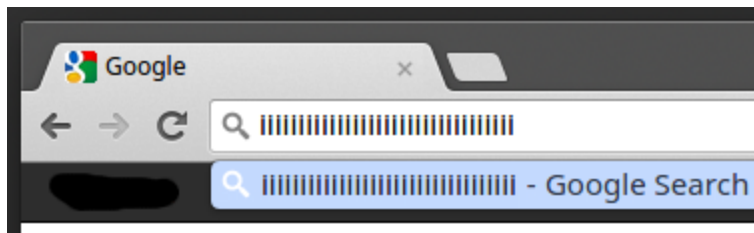
Make sure you view the images at 100% zoom, or they may show other artifacts. A corollary of this is that if you want linear layout and hence no metrics hinting, you better have subpixel text positioning supported or things will look bad.

- Hinting metrics and hinting shapes typically go together. Which makes sense: if you modify the shape to snap white and black spaces to whole pixels, you should also adjust the glyph width to match. Otherwise, things will look bad: Neighboring glyphs will collide, or the spacing between them will look odd. Same happens if you hint metrics but don't hint the shapes to match.
- If you don't have antialiasing, you *really* have to do hinting, otherwise entire stems may completely disappear. Since it's 2012, let's not go there.
- Subpixel text rendering can be *extremely* tricky to get right if you don't control the display device. It will *only* look good if either the screen is high-density (in which case subpixel text rendering does not matter much), or your display panel is high-quality and well-calibrated in respect to its gamma behavior from all viewing angles. Apple has been the only company championing both categories. Consider the following screenshot from iPhone 3. It looks ok on the phone, but the 'iiiiiiiiiiiiii' sequence probably looks really bad on your screen, because your screen is not well-calibrated, and quite possibly your browser does not do fully gamma-corrected antialiasing / compositing

either (which is another hard problem in itself).



Or this one, if you happen to have subpixel antialiasing going on also:



- There are also memory considerations when deciding which rasterization features you can afford. For example, non-antialiased non-subpixel-positioned glyphs take 1 bit per pixel of texture cache memory. Add grayscale antialiasing and that jumps to 8 bits per pixel. Add subpixel antialiasing and the number would be more like 32 bits per pixel. Add subpixel positioning (and no metrics hinting), and you may have to cache multiple (2? 3? 4? 16? up to you.) different rasterizations of the same glyph. Add different sizes of the font, different fonts, etc, and these all adds up pretty quickly.

Non-linear Layout

These are some of the conditions under which you may want to decide that non-linear layout is a good match for you:

- Your display device is very low-density. 96dpi certainly fits the bill. 320dpi certainly doesn't. There's a spectrum of opinions about the cut-off in between.

- You (and your users) come from a Windows background. Historically Windows shipped with highly hinted, non-linear, text rendering and Windows users are used to that.
- You need to render text at small sizes.
- You value readability over aesthetics.
- Your users or the rest of your codebase assume that text segments have an integral width.

If you are happy with non-linear layout, good for you. Just implement as many of the rasterization techniques as you can afford (grayscale anti-aliasing, subpixel anti-aliasing, subpixel text positioning), and provide as many different hinting options as you care to do, and choose the combination that looks best on your target devices given your criteria for “best”.

Linear Layout

There are situations that you have no choice but to do linear layout. In particular:

- You need to preserve document formatting across systems. Eg. your MS Word cannot re-layout a book just because the user changed zoom. At small sizes (which require hinting), you are left to choose between enabling shape hinting and look awful, or disabling shape hinting and look awful. Really, there’s no way you can win that one. That’s exactly why MS Word documents look so bad on older Windows.
- Similarly, the layout may be forced upon you. That is the case for PDF viewers. The PDF file tells you exactly where you need to put the glyphs. You have no say whatsoever.
- You **MUST** be linear. I.e. you have to support pinch-to-zoom or other zoom interaction that must behave like a magnifier. That by definition requires linear layout.
- You have high enough spatial density (as a result of high physical density, subpixel positioning, subpixel rendering, etc), that non-linear layout does not offer much increased legibility.

Problem at Hand

To handle emerging high-dpi displays, Chrome added a variable called the device-scale-factor. For example, if the display is 240dpi, Chrome will layout the page as if it was 120dpi, and then scale everything by a factor of 2. There are two issues with this approach:

- This, clearly, makes a strong linearity assumption!
- The webkit layer does not even know about this!

Not surprisingly, if your layout algorithm is non-linear (hinting-metrics enabled), this will break badly, as

can be seen in the image below:

[Metasearch Search Engine - Search.com](#)

www.search.com/

Search the Web by searching the best engines from one place.

[Images](#) - [Most Popular Searches](#) - [Services](#) - [Downloads](#)

Note how glyph spacing is completely off. Check these sequences: “Sea”, “sea”, and “rvi”. This is using hinted metrics suitable for 120dpi, multiplying by two, and using those with rasterization done for 240dpi.

Needless to say, something needs to be done about this. We can't ship *that*! To confirm that we have hinting enabled, lets look at the width of an 'i' at different sizes:

i i i i i i i i i i i i i i i i

```
span[0]: rect.width=0
span[1]: rect.width=1
span[2]: rect.width=1
span[3]: rect.width=1
span[4]: rect.width=1
span[5]: rect.width=2
span[6]: rect.width=2
span[7]: rect.width=2
span[8]: rect.width=3
span[9]: rect.width=3
```

Yep. Very well-hinted indeed. The values reported are in *nominal* pixels. Each nominal pixel is two physical pixels wide in the device-scale-factor=2 case.

(test case)

Solution

To solve this problem we explore three different possible solutions in the following sections.

Remove Non-linearity

The simplest way to fix the issue is to remove non-linearity during layout. At 240dpi, it's much easier to get away with no-hinting than it is at 96dpi or even 120dpi. So, we can disable hinting, enable subpixel text positioning, and get beautiful text again:

[Metasearch Search Engine - Search.com](http://www.search.com/)

www.search.com/

Search the Web by searching the best engines from one place.

[Images](#) - [Most Popular Searches](#) - [Services](#) - [Downloads](#)

Now lets see what happened to "i" measurements:

```
.....iiiiiiiiiiiiiiii
span[0]: rect.width=0.27783203125
span[1]: rect.width=0.5556640625
span[2]: rect.width=0.83349609375
span[3]: rect.width=1.111328125
span[4]: rect.width=1.38916015625
span[5]: rect.width=1.6669921875
span[6]: rect.width=1.94482421875
span[7]: rect.width=2.22265625
span[8]: rect.width=2.50048828125
span[9]: rect.width=2.7783203125
```

Look just how linear those widths are. Harmony. Problem solved.

Now, as we discussed before, there may be valid reasons to not want to do that. Web app / CSS backward compatibility is one such reason. A long-tail of hidden bugs rising from assuming integer segments in the codebase is another.

In the meantime, lets see how else we can fix this problem...

Remove Linearity Requirement

As we demonstrated the problem is rooted in the Chrome compositor doing a 2x scaling of the layout and expecting it to look right at twice the resolution. There is no inherent reason for handling high-density displays this way. Ie. instead of laying out at 120dpi and scaling the results, we can simply layout at 240dpi. That's afterall what the non-high-dpi-phobic would do. This possibility was explored in [this](#) webkit bug. Here's the mandatory screenshot:

[Metasearch Search Engine - Search.com](#)

[www.search.com/](#)

Search the Web by searching the best engines from one place.

[Images](#) - [Most Popular Searches](#) - [Services](#) - [Downloads](#)

Looks respectable. Lets check the 'i's:

.....iiiiiii

```
span[0]: rect.width=0.5
span[1]: rect.width=0.5
span[2]: rect.width=1
span[3]: rect.width=1
span[4]: rect.width=1.5
span[5]: rect.width=1.5
span[6]: rect.width=1.5
span[7]: rect.width=1.5
span[8]: rect.width=2
span[9]: rect.width=3
```

Note how the glyphs are taking non-integer widths in the nominal pixels now. This reflects the fact that they are hinted to whole physical pixels, each of which is 0.5 nominal pixel under this high-dpi mode.

Now, some would argue that this has the same problems the previous solution had. And we agree to some extent. Plus, now layout is dependent on the device-scale-factor, which some may find unacceptable!

Metrics Hinting Meets Linear Layout

A.K.A., have your cake and eat it too. Ok, looks like we're at a stalemate. There is one more thing we can try though: Hint glyph shapes and metrics for the low-resolution layout, then scale the results linearly. The resulting rendering is not optimal for the physical pixels of the screen, but is a legitimate approach. It just skews glyph shapes a bit unnecessarily. We explored that in [this](#) change. Screenshot:

[Metasearch Search Engine - Search.com](http://www.search.com/)

www.search.com/

Search the Web by searching the best engines from one place.

[Images](#) - [Most Popular Searches](#) - [Services](#) - [Downloads](#)

This doesn't look perfect, but is very respectable. Note the 'rvi' sequence, it's cramped, but that's what you get when you rasterize shapes hinted for one size at another! Lets check the 'i's:



```
span[0]: rect.width=0
span[1]: rect.width=1
span[2]: rect.width=1
span[3]: rect.width=1
span[4]: rect.width=1
span[5]: rect.width=2
span[6]: rect.width=2
span[7]: rect.width=2
span[8]: rect.width=3
span[9]: rect.width=3
```

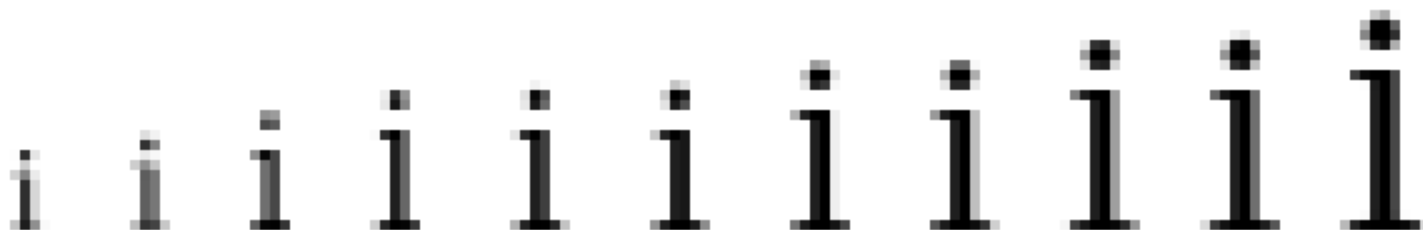
Note how the nominal width is exactly the same as the problematic case.

This solution looks better than it sounds. If you have asked me whether someone can do this I would have had said you get garbage. However, we're not. The primary reason seems to be that we are using FreeType's autohinter, and in the slight hinting mode. In the slight hinting mode, the strokes are not all necessarily snapped to full pixels (but the total glyph width is). This is good, otherwise we would have had got rendering that essentially is bulky two-pixel-wide all over. We are not. Here are two zoomed-in pictures of the 'i's to show that:

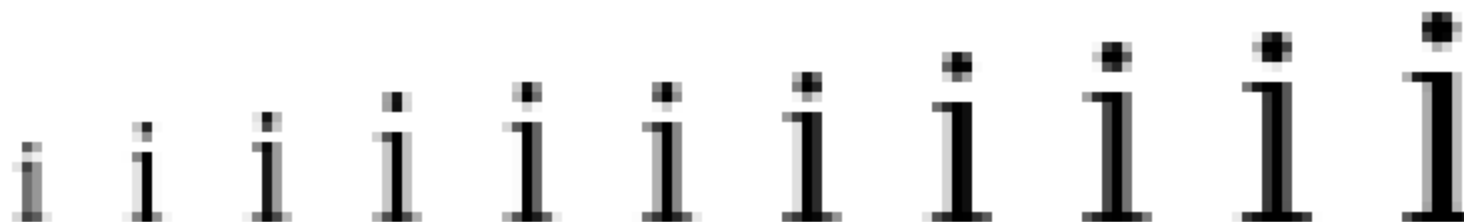
Properly hinted rasterization:



Our hybrid weirdo:



And no-hinting subpixel-positioned for comparison:



The most noticeable difference between the properly hinted version and our hybrid version seems to be in the hinting of the Y direction. Which makes sense. In the hybrid, you note that the autohinter has been trying hard to snap horizontal edges to whole nominal-pixel boundaries, which translates to two physical pixels in the rendering. Ie, the distance between the body of 'i' and the dot, the height of the body, etc, are mostly even numbers. The same is true about the full glyph width, but not about the individual features (the serifs, the stem width).

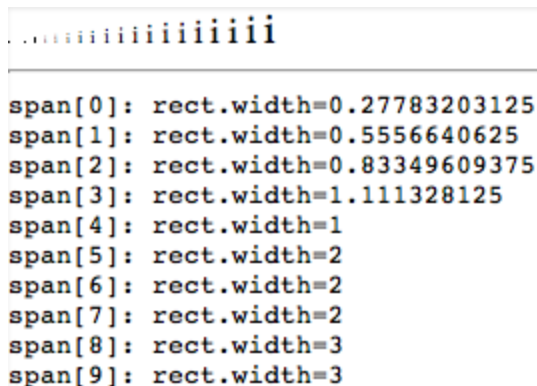
These nice properties will not hold for aggressively hinted fonts though. For example, bytecode-hinted fonts like Times New Roman, Tahoma, and Arial may simply render bulky using this method. We have not tested that.

Other Systems

What have we been smoking? Is the idea of hinting and then scaling so broken that one should not even explore it? Lets see what other systems do when it comes to their approach to text rendering:

- Historically Windows and Linux never did subpixel text positioning, and as a result always used hinted fonts. Windows hinted fonts more aggressively than Linux, so much as to be perceived pixelated and aliased.
- Internet Explorer on Windows 8, however, supports subpixel text positioning with unhinted metrics. The text looks blurrier than the rest of Windows. This is true in both Desktop and Metro modes of IE. This is expected since IE has pinch-to-zoom. Note that it only does this if [not in compatibility mode](#).
- iOS (iPhone and iPad) have always done subpixel text positioning, including in the browser. This is not surprising since iOS devices support pinch-to-zoom.
- Historically MacOS X has been doing subpixel text positioning everywhere and only hint font metrics in the Y direction. But surprisingly NOT in Safari! In Safari, metrics are hinted.

That last case is interesting and worth looking into. I'm assuming that Apple made this decision for backwards compatibility with web apps when CSS was being written in exact numbers of pixels. (pre-historic by today's standards). Anyway, here's the numbers for Safari in the low-dpi mode:



```
span[0]: rect.width=0.27783203125
span[1]: rect.width=0.5556640625
span[2]: rect.width=0.83349609375
span[3]: rect.width=1.111328125
span[4]: rect.width=1
span[5]: rect.width=2
span[6]: rect.width=2
span[7]: rect.width=2
span[8]: rect.width=3
span[9]: rect.width=3
```

Fair enough: they do hinting, except for very small sizes. That's actually a good idea that we may want to implement in FreeType.

Now, lets see what the numbers look like when we switch to the high-dpi mode, as shipped with the retina Macbook Pros:

.....iiiiiiiiiiiiiii

```
span[0]: rect.width=0.27783203125
span[1]: rect.width=0.5556640625
span[2]: rect.width=0.83349609375
span[3]: rect.width=1.111328125
span[4]: rect.width=1
span[5]: rect.width=2
span[6]: rect.width=2
span[7]: rect.width=2
span[8]: rect.width=3
span[9]: rect.width=3
```

Snap! Twice the size, but exactly same numbers! Close inspection proved to me that Safari is indeed hinting metrics at the low-dpi mode and scaling by a factor of 2. In other words, no matter what size you try, you cannot get an 'i' glyph to consume an odd number of physical pixels on the retina display.

This is interesting. And very close to our hack! Now, if I was to explain the why, I think it's because Apple does not like to break anyone's layout, and to minimize the amount of testing needed by app developers. That is why they always wait for the technology to get there to switch to exactly twice the resolution / density, such that they can multiply everything by two without having to deal with rounding issues and seams. That makes some kind of sense given that they wanted to stay with hinted metrics for Safari even though the rest of OS X was not hinting metrics. And while it doesn't look perfect, it does not look bad either:

[Metasearch Search Engine - Search.com](#)

www.search.com/

Search the Web by searching the best engines from one place.

[Images - Most Popular Searches - Services - Downloads](#)

As to the how, their approach seems to be different from, and superior to, ours. Lets look at the close-up

of the ‘i’s:



Note how even in the Y direction the shapes seem to be very linearly transformed and not snapped. This is in fact a result of different approaches to hinting between Apple and FreeType’s autohinter more than anything else. We can, if we want to, add a mode to FreeType to force hinting metrics for one resolution, but hint the actual stems for another resolution given the desired glyph width as a constraint. We don’t have any free software solution for performing that operation. It is not impossible to develop but it is not a trivial task either.

Conclusions and Recommendations

We presented three solutions to solve the same problem. Each has its own merits and shortcomings. Personally I would recommend that for high-dpi devices we stick to full subpixel positioned text (no metrics hinting), and only if that proved to be problematic consider switching to the hybrid solution.

For low-dpi devices we should stick with what we currently have, ie. hinted text and no subpixel text positioning. That is, unless, we can sort out our gamma and subpixel filtering issues, have great displays on devices we ship, and full subpixel text looks good on them. That’s not a very likely situation in the short term.

An interesting situation arises when you attach two displays to a device, one high-dpi and one low-dpi. Do we care about the layout on both displays being the same? If yes (and only if there’s a good reason for that answer), then if the main display (ie. laptop display) is low-dpi, and external display is high-dpi, then we will end up using the hybrid approach on the external display. That’s one possible place that this may be useful even if we end up not using it for our main high-dpi display.

Other Resources

There are a few good reads on the net that explore some of these ideas in more detail:

- “The Raster Tragedy”, rather long but detailed exploration of all things hinting: <http://www.rastertragedy.com/>
- Subpixel antialiasing (rendering) in a nutshell: <http://www.grc.com/ct/ctwhat.htm>
- “Rendering good looking text with resolution-independent layout”, short and good read with self-explanatory title: <http://people.redhat.com/otaylor/grid-fitting/>
- “A Treatise on Font Rasterisation With an Emphasis on Free Software”, putting it all together (sans the high-dpi part): <https://freddie.witherden.org/pages/font-rasterisation/>
- “Text Rasterization Exposures”, putting it all together (again, sans high-dpi), though misguided at times: http://www.antigrain.com/research/font_rasterization/index.html

Acknowledgements

- Daniel Erat (derat@) did the plumbing to turn subpixel text positioning on,
- Terry Anderson (tdanderson@) developed patches for the two other approaches and helped with producing screenshots of Chrome,
- Rick Byers (rbyers@) led the effort to explore the solution space and produced screenshots of Safari,
- Rob Kroeger (rjkroege@) and Alex Nicolaou (anicolao@) drove me to explore the problem,
- Many other individuals reviewed the patches and shared their insights.