Understanding fonts and fontconfig

Nicolas Mailhot
Fedora Fonts Special Interest Group
http://fonts.fedoraproject.org

Vocabulary

- A “*typeface*” is a writing design
- A “*typeface*” is composed of variants, or “*faces*”
- A “*font*” is a digital file containing “*typeface*” data
- A “*face*” name is composed of:
  - A “*font family*” name : “*DejaVu Sans*”
  - A “*font style*” name : “*Oblique*”, “*Condensed ExtraBold*”
- The symbols contained in a “*font*” are “*glyphs*”
- “*Glyphs*” are identified by “*code-points*”
- A letter is drawn using one or several “*glyphs*”

Some history

In ancient times

- You used one small 8-bit local encoding
- All your fonts had 100% coverage of this encoding
  - It was small and trivial to draw
- Fonts didn't declare styles such as *“Bold”*
  - Users had to guess looking at the font name
- Rendering text consisted of stacking letter after letter sequentially, from left to right
  - This is still the model used in terminal emulators
  - 1 letter = 1 symbol = 1 glyph = 1 byte

I18n limits

- Could not communicate with people that used another 8-bit encoding
- Could not write multi-lingual documents
- Could not write languages that do not use simple left-to-right stacking
- Needed to simplify or even transliterate many languages so they fit in the encoding

Bland text

- The computer was a glorified typewriter
- Every document used the same boring fonts
- No typesetting for the masses
  - Professional typesetters used more varied typefaces
  - Professional typesetters did not limit themselves to one glyph per letter (*swashes*, *ligatures*...)
  - Professional typesetters knew that sometimes is is a good idea to replace a symbol with an alternative

We are not in ancient times any more

- The web unified written communication
- Unicode replaced local encodings
- Text libraries are now smart enough to go beyond the linear glyph stacking model
- Typefaces multiplied, with numerous faces/styles
  - Including *ligatures*, *swashes*, *alternate glyphs*, and other typographic features

At a cost

- Partial coverage!
  - No typeface covers 100% Unicode in all common styles
- Variable availability!
  - No typeface is guaranteed to be available everywhere
- Choice!
  - Can not expect users to know font names by heart
  - Can not expect local fonts to match only the local language, need complete coverage to browse the web

Modern fonts: coverage

Unicode coverage

- The Unicode consortium has divided the encoding plane in discrete blocks
- Each of those blocks has a specific target
  - For example *“Cyrillic”*
- Most of those blocks are incomplete
  - Space is reserved for new glyphs
- Every time a new Unicode standard revision is published, those blocks may have grown
  - New Unicode standard revisions happen regularly

Language coverage ≠ Unicode coverage

- A language may need only part of a Unicode block to be written
- A language may need parts of several Unicode blocks to be written
- Several languages may use the same parts of an Unicode block
  - For example, French uses the same glyphs as English, plus a few others (*œ, ç, ù ê...*)
- Some Unicode blocks are not used by any human language (*Mathematic symbols*, *Arrows*...)

Language coverage ≠ code-point coverage

And different languages do not draw the same code-points the same way

- Russian and Balkanic Cyrillic have different drawing conventions for a few glyphs
- Arabic, Farsi and Urdu have different drawing conventions for many glyphs
- Chinese and Japanese have different drawing conventions for a lot of glyphs

Junk coverage

- Problem:
  - Many applications do not switch fonts automatically, when switching languages
  - You need to write English everywhere
  - Switching fonts manually is a chore
- Solution?
  - most fonts include a basic latin block, even when they are not primarily latin fonts
  - This block is often of very poor quality
  - The same problem exists for other blocks

Other coverage

- Modern *"smart"* fonts can also provide several variants of the same glyph, purely for typographic effects (not i18n)
  - This is better than shipping a separate file with hundreds of glyphs every time you want to decline a couple of them
  - It keeps user font lists short
- It is not expected by application writers

Conclusion

- Coverage is no longer simple
- It can not be assumed by applications
- The list of code-points included in a font is insufficient to decide what languages it is appropriate for

Modern fonts: naming

Many layers of naming

- Ancient fonts had no *"style"* concept
- Since then many different conflicting ways to specify a *"style"* name have been specified
  - Especially during last millenium's desktop and font format wars
- They all have different limitations
  - One of the earliest ones only allows: *"Normal"*, *"Bold"*, *"Italic"* or *"Bold Italic"*

Many overlapping layers of naming

- Applications have different naming expectations, depending on when they were coded and for what target desktop
- To accommodate them all, most fonts expose their naming using all the possible naming layers
- The font author is supposed to make sure their content is consistent, and one layer does not say *“left”* when the other says *“right”*
- However, font authors are human

First font naming unification

- OpenType ended the font format wars, and defined a new naming layer
  - It was supposed to supersede all the previous ones
  - In this naming layer there is no limitation whatsoever on the style name
  - Font authors were expected to exert common sense
- Since most applications had no support for this naming layer yet, font authors were not exposed to the effects of their naming decisions
- Ooops

First font naming unification: failure

- Font authors started declaring creative style names such as *"by Quantized and Calibrated"*
- Font authors did not pay attention to their other metadata; applications could only rely on style names to guess font attributes
- CSS styling became pervasive. CSS has strict naming expectations. Faces must be selectable:
  - By name
  - By name + an operator such as *"bold-er"*
- Creative style names are useless for CSS

Second font naming unification: WWS

- To be able to use CSS-like styling in its new Vista presentation layer, Microsoft convinced other OpenType specification editors to introduce another font naming layer: WWS
- This layer only point is to enforce simple naming conventions so applications can actually use fonts
- The WWS authors tried very hard to minimize the changes needed in existing fonts
- However, many fonts still need fixing

The WWS naming model

- Font style names are composed solely of *“Width”*, *“Weight”*, *“Slant”* attributes
  - space separated, and in that exact order
- Each attribute is only specified once:
  - no *“Bold Medium”* allowed
- Attributes are declared using standard qualifiers
- Everything else belongs to the family name
- The family name does not include any *“Width”*, *“Weight”*, *“Slant”* attribute qualifiers

WWS naming: normal is implied

- A font with no special “*Width*”, “*Weight*” or “*Slant*” characteristic does not declare any corresponding qualifier in its style name.
- When “*Width*”, “*Weight*” and “*Slant*” are all normal, the style name is “*Regular*”

WWS naming: *"Width"* qualifiers

WWS name	OS/2 usWidthClass
UltraCondensed	1
ExtraCondensed	2
Condensed	3
SemiCondensed	4
Normal (implied)	5
SemiExpanded	6
Expanded	7
ExtraExpanded	8
UltraExpanded	9

WWS naming: *"Weight"* qualifiers

WWS name	OpenType weight
Thin	100
ExtraLight	200
Light	300
Normal (implied)	400
Medium	500
DemiBold	600
Bold	700
ExtraBold	800
Black	900
ExtraBlack	950

WWS naming: *"Slant"* qualifiers

WWS name
Oblique
Normal (implied)
Italic

WWS naming: last words

- Microsoft also defined an heuristic to:
  - compute WWS names from old-style names,
  - display old names while actually processing the computed WWS ones
    - i.e. do not have to change "*Arial, Narrow*"
- This heuristic will fail horribly on some fonts
- When it works, fonts will behave differently from their displayed name, confusing users
- It is therefore worthwhile to change fonts so that they declare strict WWS names that need no fixing

Fontconfig overview

Fontconfig's mission

- Ancient font systems expected the user to always select the appropriate font
  - Otherwise things failed and no text was displayed
- This burden is too high given modern constraints
  - Partial encoding coverage, variable font availability
- Fontconfig is here to automate font fall-back so text is displayed in all cases
  - Using the more appropriate font available
  - Displaying text is more valuable than failing completely

Font stacks

- Fontconfig analyses available fonts, and constructs stacks of fonts and font subsets
- When a fontconfig application requests a particular font context (family name, style name, needed glyphs, current language, etc), fontconfig will:
  - try each element in the corresponding stack till it finds elements corresponding to the application needs,
  - and return them
- Sometimes its answer will be composed of elements taken from different fonts

Human intervention

- In an ideal world fontconfig would be fully automatic
- However actual fonts:
  - do not declare all the information fontconfig needs
  - sometimes export wrong information
  - Are not of uniform quality (see *"junk coverage"*)
- Humans need to provide fontconfig this missing information

Basic fontconfig rule structure

A fontconfig rule is composed of:

- The optional specification of what it is supposed to match
  - if absent, match all the times ie create a fake font
- an ordered list of font elements to try when trying this matching (the font stack)
- The optional specification of font characteristics to fake in the result
  - if they can not be deduced from the font stack elements or,
  - if the values provided by font stack elements need fixing

Fontconfig rules are modular

- Old Fontconfig used a single master file. However:
  - Every font packager needed to edit this file (conflicts!)
  - End-users could not safely select the rules they wanted
- Nowadays fontconfig configuration is modular:
  - Fontconfig reads all the files in its configuration directory, in filename sequential order
  - It merges their rules: if file 50-x defines element X in stack A, and file 51-y defines element Y in the same stack, fontconfig will consider this stack is composed of X followed by Y

Generics

Because CSS and fontconfig address the same font selection problem at different levels, fontconfig provides synthetic fonts that match CSS generics

CSS/fontconfig generic font name
serif
sans-serif
cursive
fantasy
monospace

Fontconfig use-cases

Foreword

- The following examples use a fontconfig-ish simplified syntax
- Actual fontconfig syntax is less simple and consistent
- Hopefully, fontconfig will evolve in this direction

Declaring a new font in the right generic

Fontconfig will auto-discover new fonts, but it can't guess their desirability or to what generic they belong

```
<rule>
  <prefer>
    <family>my font family</family>
  </prefer>
  <fake>
    <family>generic name</family>
  </fake>
</rule>
```

Generic choice decision tree

1. A font that can not be used to write long *“professional”* texts is a *“fantasy”* font.
2. Otherwise:
    1. if it's monospaced, it's a *“monospace”* font,
    2. if it's variable-width and:
        1. it simulates hand-written text, it's a *“cursive”* font,
        2. it uses serifs, it is a *“serif”* font,
        3. it does not use serifs, it is a *“sans-serif”* font,
3. Otherwise, are you really sure it is not a *“fantasy”* font?

In case of conflicting coverage

```
<rule>
  <match>
    <font>
      <lang>a language</lang>
    </font>
  </match>
  <prefer>
    <font>
      <family>my font family</family>
      <coverage type=”lang”>a language</coverage>
    </font>
  </prefer>
  <fake>
    <family>generic name</family>
  </fake>
</rule>
```

Restrict your generic registration to cases where matching is being done for a specific language, and only for the glyphs associated with this language (people quote words in other languages all the time, you do not want to override the glyph defaults for those other languages even if the application declares your language)

Because the language being matched is not always known, the same block without the `<lang>` restriction can be added to a file with lower priority (higher prefix number), to make your font appear even in this case.

Declaring where to complete a font from

```
<rule>
  <match>
    <family>my font family</family>
  </match>
  <first>
    <family>my font family</family>
  </first>
  <prefer>
    <family>some other matching font that may be available</family>
    <family>yet another matching font that may be available</family>
  </prefer>
  <last>
    <family>generic name</family>
  </last>
</rule>
```

It is unsafe to use `<first/>` unless you also control the matched name, which is the case here.

In case of junk coverage

```
<rule>
  <match>
    <family>my font family</family>
  </match>
  <first>
    <font>
      <family>my font family</family>
      <not>
        <coverage type=”unicode”>
          <min>start of the junk unicode region</min>
          <max>end of the junk unicode region</max>
        </coverage>
      </not>
    </font>
  </first>
  <last>
    <family>generic name</family>
  </last>
</rule>
```

<not> can be used to specify some coverage to reject, and not select. Coverage can be specified by language <coverage type=”lang”/> or by unicode region <coverage type=”unicode”/>. Multiple coverage elements can be specified.

Declaring a font is a valid substitute to another

```
<rule>
  <match>
    <family>other font family</family>
  </match>
  <prefer>
    <family>my font family</family>
  </prefer>
</rule>
```

It is safe because the packager of the other font family used `<first/>` which will override this rule if the other font is present

Font rescaling

The font author didn't use the same scale as other authors, so text using his fonts appears too big or too small compared to other fonts at the same size.

```
<rule>
  <match>
    <family>my font family</family>
  </match>
  <first>
    <family>my font family</family>
  </first>
  <fake>
    <font>
      <matrix>
        <double>factor</double>
        <double>0</double>
        <double>0</double>
        <double>factor</double>
      </matrix>
    </font>
  </fake>
</rule>
```

Font remapping

The font author made a mistake in its font naming and it is desirable to expose a different naming.

```
<rule>
  <first>
    <font>
      <family>my font family</family>
      <style>broken non-standard style name</style>
    </font>
  </first>
  <fake>
    <font>
      <style>correct standard style name</style>
    </font>
  </fake>
</rule>
```

Hide the broken name from applications

```
<hide>
  <font>
    <family>my font family</family>
    <style>broken non-standard style name</style>
  </font>
</hide>
```

Font merging 1

The font author released what should have been a single font as multiple fonts.

```
<rule>
  <first>
    <family>one of the font families to merge</family>
  </first>
  <fake>
    <family>merged font family</family>
  </fake>
</rule>
<rule>
  <match>
    <font>
      <lang>a language</lang>
    </font>
  </match>
  <prefer>
    <font>
      <family>the font family to merge for this language</family>
      <coverage type=”lang”>a language</coverage>
    </font>
  </prefer>
  <fake>
    <family>merged font family</family>
  </fake>
</rule>
<rule>
  <prefer>
    <family>fallback font family to merge</family>
  </prefer>
  <fake>
    <family>merged font family</family>
  </fake>
</rule>
```

Font merging 2

And then of course hide the merged elements from applications

```
<hide>
  <family>one of the font families to merge</family>
  <family>the font family to merge for this language</family>
  <family>fallback font family to merge</family>
</hide>
```