Blink's Text Stack

This README serves as an documentation entry point of Blink's text stack.

It can be viewed in formatted form here.

Overview

Blink's font and text stack covers those functional parts of the layout engine that provide measurement, geometric operations and drawing for runs of CSS-styled HTML text.

The API methods in font.h describe the interface between the layout code and the font code. font.h provides API for mainly three kinds of requests coming from the layout and paint code:

- Measuring text
- Geometric operations used for text selection:
    - Computing bounding boxes
    - Mapping from coordinates to character indices
    - Mapping from character index to coordinates
- Painting text

From source HTML to visual output this roughly comprises the following stages:

- From CSS Styling to Font Objects
- Using this font definition for matching against available web and system fonts
- An excursion into prerequisites before text shaping can be performed
- Segmenting text into portions suitable for shaping
- Looking up elements from the previously shaped entries in the word cache
- Using the matched font for shaping and mapping from characters to glyphs
- Font fallback

From CSS Styling to `Font` Objects

During the style resolution stage of layout `ComputedStyle` objects are calculated for each element of the DOM tree. Each `ComputedStyle` will own a `Font` and a `FontDescription` object. For this to work, after CSS is parsed into the various specialized `CSSValue`-derived types, the `ConvertFont…` methods in `style_builder_converter.cc` convert from `CSSValue`s to `FontDescription` data structures. In the opposite direction, the `ValueForFont…` methods in `computed_style_css_value_mapping.cc` convert back from `FontDescription` back to CSS values.

Then, during style resolution, `FontBuilder::CreateFont` is called to update the `Font` object based on the properties stored in `FontDescription` and thus link those two objects together. The key to linking them together is `CSSFontSelector`. `Font` is assigned a `CSSFontSelector`, which serves as a looking

glass to know about which fonts are available in the document's scope. This includes fonts from the realm of web fonts as well as system fonts.

The `Font` object does not yet represent a particular typeface. Similarly, `FontDescription` does not point to an individual typeface yet. This is because `FontDescription` contains a `FontFamily` list of family names originating from the CSS `font-family` property.

`Font` objects represent the API for style and layout operations to retrieve geometric information about a font used in the DOM: retrieve metrics such as line height, x-height etc. and measure text to retrieve its bounding box. The paint stage uses `Font` objects to draw text.

Only when such geometric or painting operations are requested from a `Font` object, internally `Font` will perform font matching.

Font Matching

As soon as a `Font` object is requested to perform any operation, it needs to find an individual typeface to work with. The information stored in its `FontDescription` member needs to be resolved against available web fonts or system fonts. This process is called font matching. The detailed algorithm for this process is described in the CSS Fonts Module, section Matching Font Styles.

For this purpose, a `Font` object has an `Update(CSSFontSelector*)` method to inform it about an updated set of available fonts. `Font` internally delegates the resolution of CSS family names from the `font-family` against available fonts to its `FontFallbackList` member, and thus hands the `CSSFontSelector` down to `FontFallbackList`.

`FontFallbackList` calls `CSSFontSelector::GetFontData` with a `FontDescription` to perform a lookup. `CSSFontSelector` will in turn ask `FontFaceCache` to find a FontData object among the available web fonts. If there is already a match for `FontDescription` in the cache, this `FontData` object is returned. If not, the `FontSelectionAlgorithm::IsBetterMatchForRequest` comparison function is used to find the best match among the available web fonts. This comparison function implements the CSS font matching algorithm.

If `CSSFontSelector` can't find a font using this approach, it will try to find a font from the system. To this end, it will query `FontCache`. `FontCache` is for system fonts what `FontFaceCache` is for web fonts. `FontCache` has OS/system specific implementations in font_cache_skia.cc, font_cache_mac.mm, font_cache_linux.cc, font_cache_android.cc in order to perform system font lookups using the respective OS/system API.

Excursion: Setting up Shaping

Before looking at the next stages, we need to understand what text shaping means.

Shaping text is the process of mapping a unicode string to a sequence of glyph IDs from a font plus their exact geometrical positions through performing OpenType layout operations defined in the font. For Latin script, the output is mostly glyph IDs and horizontal advances, but for complex scripts, the output also describes positioning in the vertical direction, reordered glyphs, and association into grapheme clusters.

In more detail, a unicode string is not the only required input for this mapping. Instead, a number of variables are required before shaping can be performed. For a single run of text ready for shaping, the following input variables need to be isolated and constant:

- Font
- Font Size
- Text Direction (LTR, RTL)
- Text Orientation (Horizontal, Vertical)
- Requested OpenType Features
- Unicode Script
- Unicode Language
- Text (the actual text of this run)
- Context (the text surrounding this run of text)

This means that before shaping can be performed, incoming text and `FontDescription` information coming from the layout code needs to be segmented into sub-runs where the above inputs stay the same. For example, if the incoming text contains Unicode sequences of different scripts, the incoming text run needs to be broken up into sub-runs of only one script. Similarly, if the incoming text contains differing orientations, for example Latin text embedded in Japanese vertical layout, then these sub-runs need to be isolated and shaped separately.

Emoji

Emoji place additional requirements in isolating sub-runs for shaping. Emoji Unicode code points and code point sequences have different default presentation styles, text-default, or emoji-default. This is defined in the section Presentation Style of Unicode Technical Report #51. So in order to select the correct font for emoji presentation — either a color font, or a regular contour font — the incoming text needs to be segmented and isolated by its emoji properties as well.

Word Cache

Because text shaping and font fallback are costly operations and geometric operations on text runs are performed over and over again during layout, a word cache is used to speed up these operations.

Cacheable units

The basic unit for storing shaping results in a cache is a word, separated by spaces. Since CJK text often does not use spaces to separate words, for CJK text, each individual CJK character is treated as a word.

Cache Keying

As the excursion on text shaping explains, multiple variables go into the shaping equation, such as a fixed font, a fixed font size, script, et cetera. This in turn means that the word cache becomes invalid if the set of available font changes changes because `FontFallbackList` would change in what it returns when available fonts change. We cannot reduce the cache key computation for the word cache to the string/word itself plus the requested font as represented by `FontDescription`, but we also need to capture the set of available fonts at the time shaping for this word and its `FontDescription` was performed. This state is

captured by computing a composite key off of the `FontFallbackList` in `FontFallbackList::CompositeKey`.

Accessing the Cache

`cachin_gword_shaper.h` is the entry point for retrieving shaping results through the word cache. It defers to `caching_word_shape_iterator.h` for word/space or CJK segmentation and responds to requests for a `TextRun`'s `Width()` or returns a `ShapeResultBuffer` containing a list of `ShapeResult` objects. If not found in the cache `ShapeResult` objects are produced by the text shaping stage. So `CachingWordShaper` serves as the accelerating caching layer between `Font` operations and `HarfBuzzShaper`.

Run Segmentation

The section Setting up Shaping described the requirements for constant input requirements before shaping can be performed.

RunSegmenter is the top level API for segmenting incoming text runs using sub-segmenters. It splits text by their Unicode script property (via `ScriptRunIterator`), orientation and direction — horizontal LTR/RTL, vertical LTR/RTL (via `OrientationIterator`), and emoji presentation attributes (via `SymbolsIterator`).

`RunSegmenter` is constructed from a text run in UTF-16 `UChar` format, then functions as an iterator returning sub-runs of constant script, emoji presentation, and orientation. These sub-runs are then suitable as units for shaping.

Text Shaping

The text shaping implementation is in shaping/harfbuzz_shaper.h and shaping/harfbuzz_shaper.cc

Shaping text runs is split into several stages: Run segmentation, shaping the initial segment starting with the primary font, identifying shaped and non-shaped sequences of the shaping result, and processing unshaped sub-runs by trying to shape using the remaining list of fonts, then trying fallback fonts until the last resort font is reached.

If small/petite caps formatting is requested, an additional lowercase/uppercase segmentation pass is required. In this stage, OpenType features in the font are matched against the requested formatting. If the respective caps feature is found, the feature is used in shaping and activated from the font. Otherwise small-caps glyphs are synthesized as required by the CSS Level 3 Fonts Module.

Below we will go through one example — for simplicity without caps formatting — to illustrate the process: The following is a run of vertical text to be shaped. After run segmentation in `RunSegmenter` it is split into 4 segments. The segments indicated by the segementation results showing the script, orientation information and font fallback preference (text, emoji presentation) of the individual segment. The Japanese text at the beginning has script “Hiragana”, does not need rotation when laid out vertically and does not require an emoji font, as indicated by `FontFallbackPriority::kText`.

```
0 い
1 ろ
2 は USCRIPT_HIRAGANA,
    OrientationIterator::OrientationKeep,
    FontFallbackPriority::kText

3 a
4 ̄(Combining Macron)
5 a
6 A USCRIPT_LATIN,
    OrientationIterator::OrientationRotateSideways,
    FontFallbackPriority::kText

7 い
8 ろ
9 は USCRIPT_HIRAGANA,
     OrientationIterator::OrientationKeep,
     FontFallbackPriority::kText
```

Let's assume the CSS for this text run is as follows: `font-family: "Heiti SC", Tinos, sans-serif;` where *Tinos* is a web font, defined as a composite font (i.e. two separate CSS `@font-face` instances with different `unicode-range` subsetting ranges), one for Latin `U+00-U+FF` and one unrestricted `unicode-range`.

`FontFallbackIterator` provides these fonts to the shaper. It will start with *Heiti SC*, then the first part of *Tinos* for the restricted unicode-range, then the unrestricted full unicode-range part of *Tinos*, then a system *sans-serif*.

The initial segment 0-2 is sent to the shaper, together with the segmentation properties and the initial Heiti SC font. Characters 0-2 are shaped successfully with Heiti SC.

```
Glyphpos: 0  1  2
Cluster:  0  1  2
Glyph:    い ろ は
```

The next segment, 3-6 is passed to the shaper. The shaper attempts to shape it with Heiti SC, which fails for the Combining Macron. So the shaping result for this segment would look similar to this.

```
Glyphpos: 3 4 5 6
Cluster:  3 3 5 6
Glyph:    a □ a A (where □ is .notdef)
```

Now in the `extractShapeResults()` step we notice that there is more work to do, since *Heiti SC* does not have a glyph for the Combining Macron combined with an a. So, cluster 3 consisting of the characters a plus ̄(Combining Macron) is placed in the `HolesQueue` for clusters that need to be processed after switching to the next fallback font.

After shaping the first segment as whole, the font is cycled to the next font coming from `FontFallbackIterator` and the remaining items in the `HolesQueue` are processed, picking them from the head of the queue.

In this case, the next font is *Tinos* (for the range `U+00-U+FF`). Shaping using this font, assuming it is subsetted, fails again since there is no combining mark available in the unicode range `U+00-U+FF`. This triggers requesting yet another font. This time, the Tinos font for the full range. With this, shaping succeeds with the following HarfBuzz result:

```
Glyphpos: 3 4 5 6
Cluster:  3 3 5 6
Glyph:    a ◌̄a A (with glyph coordinates placing the ◌̄above the first a)
```

Now this sub run is successfully processed and can be appended to `ShapeResult`. A new `ShapeResult::RunInfo` is created. The logic in `ShapeResult::insertRun` then takes care of merging the shape result into the right position the vector of `RunInfo`s in `ShapeResult`.

Shaping then continues analogously for the remaining Hiragana Japanese sub-run, and the result is inserted into `ShapeResult` as well.

Font Fallback

The section Text Shaping illustrates that font selection during shaping is part of an iterative process, which first tries to use as many glyphs as possible from the primary font, then in subsequent iterations proceeds to fill gaps from the secondary font and so on until there are no more so called `.notdef` glyphs, i.e. no more boxes of text for which no glyph was found.

`FontFallbackIterator` meets the needs of HarfBuzzShaper to deliver new fonts to fill such gaps. `FontFallbackIterator` is an iterator style API, which on calling `next()` will deliver the first suitable font to try. A `FontFallbackList` is the internal representation of fonts resolved from the CSS `font-family:` property. `FontFallbackList` attempts to resolve font family names from the CSS `font-family:` property in descending order. It tries to find them among the list of available web fonts which were declared by `@font-face` rules or added to the document using the `FontFace` JavaScript API. If a requested font family is not found among web fonts, system fonts are searched next. This behavior matches the requirements of the font style matching algorithm of the CSS Fonts specification, which mandates to prioritize web fonts over system fonts.

`FontFallbackIterator` is intialized with a `FontFallbackList` and starts retrieving fonts from this list as its first source for fonts. If during shaping a run of text `HarfBuzzShaper` keeps requesting additional fonts after `FontFallbackList` is exhausted, this means that `HarfBuzzShaper` still tries to fill gaps in the run. In other words, the fonts specified in `font-family` did not have sufficient glyph coverage to draw the whole run of text on the screen. In this situation, system font fallback is invoked, which means attempting to find a surrogate font which contains those glyphs that were missing so far. To this end `FontFallbackIterator` calls `FontCache::FallbackFontForCharacter()` in order to retrieve a font that has a glyph for the requested Unicode character. This means, beyond what is listed in `font-family` there are additional system fonts pulled in to the shaping process.

In summary, `FontFallbackIterator` feeds fonts from the CSS `font-family` list as well as system fallback fonts to `HarfBuzzShaper` for use in the shaping iterations until ideally all gaps are filled and the full text run can be drawn with the correct glyphs.