Highlight overlay painting

Authors: dazabani@igalia.com, rego@igalia.com
February 2022

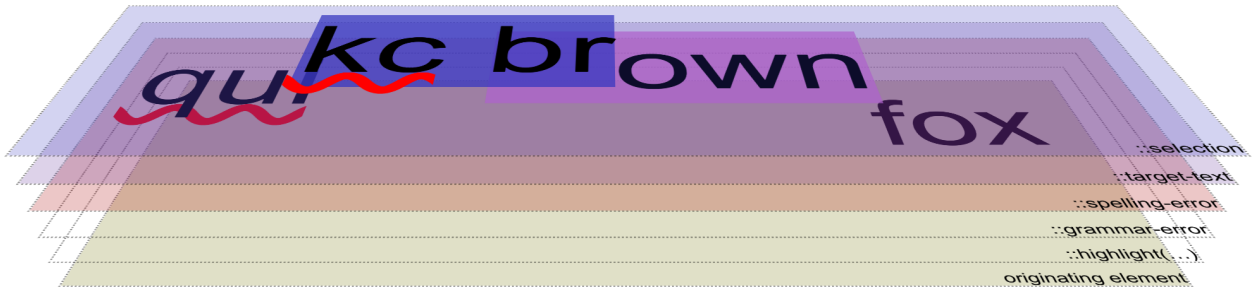
Highlight overlays are the Web-exposed generalisation of text selections, and painting them correctly requires new and more complex algorithms in PaintNG.

- Blink feature: HighlightOverlayPainting
- Code affected: PaintNG, CSS, style
- Bug: 1147859

Background

::selection, ::target-text, ::spelling-error, ::grammar-error, and each ::highlight(…) are the *highlight pseudo-elements*. There is one *highlight overlay* for each highlight pseudo, and these overlays are painted over the originating content. Highlight pseudos can introduce their own backgrounds, text shadows, and text decorations.

When multiple highlights are active for some part of the text, we paint all of their backgrounds and shadows, but only the topmost active highlight paints the “text proper” (non-shadow text), combining the decorations from originating content and all active highlights. The text proper is “suppressed” in the originating layer and all other overlays.



Text and decorations are painted by the topmost active highlight (demo).

For more details, see css-pseudo #highlight-painting.

Old behaviour

Painting behaviour is currently split between ::selection and the other highlights.

::selection more or less complies with the spec already, and even has a bunch of dedicated logic for edge cases, like painting pixel-snapped backgrounds in vertical writing modes, or painting text twice to handle split ligatures and ink overflow.

The other highlights use a naïve painting algorithm that makes no attempt to avoid painting text proper twice, so it generally only made sense for non-CSS markers, which were limited to painting a background under everything (e.g. find-in-page) and/or a squiggly line over everything (e.g. composition, suggestion).

Initial algorithm (CL:3426940, 3529115, 3538562)

Given a text fragment with a range of offsets, plus ranges for each active highlight that touches the fragment in some way, how can we determine which highlight overlays need to be painted, which parts of the text need to be painted by a given overlay, and which decorations to use for each part?

An initial algorithm would transform those inputs into *layers*, *edges*, and *parts*.

In each time complexity below, *n* is how many highlighted ranges are associated with the text node (edges are asymptotically equivalent at 2*n*+2), and *m* is how many highlight overlays those ranges are associated with (layers are equivalent at *m*+1).

m could theoretically be large if there are ranges for many different custom highlight names in a single text node, but for all currently known use cases, the number of different custom highlights will be page-design-dependent rather than content-dependent, plus up to 4 overlays for the non-custom highlights.

The exact number of parts depends on how the offsets of the highlighted ranges overlap, but it can never exceed the number of edges, so we also say this is *n*.

Layers

We start by *computing the layers*, which yields a list consisting of the originating content (always) plus the highlight overlays that actually apply to the text fragment, in the spec's highlight painting order. For example, if the fragment only has spelling errors, then the result is just [originating, ::spelling-error].

This is O(*n*) because of the push loop, plus O(*m* log *m*) because of the sort.

type Layer = tuple of (*pseudo*, *argument*)
let *layers*: list of Layer = []
layers.push((null, null)) // originating
for each highlighted range (*pseudo*, *argument*, *from*, *to*)
    layers.push((*pseudo*, *argument*))

sort *layers* by highlight painting order
return *layers*

Edges

To help us determine which pieces of text belong to each overlay and which decorations to apply, we first need to *compute the edges*, yielding a list of highlight boundaries in offset order, each indicating that some layer starts or ends there.

This is O(*n* log *n*) because of the sort.

type Edge = tuple of (*offset*, *layer*, start | end)
let *edges*: list of Edge = []
for the originating fragment (*from*, *to*)
    edges.push((*from*, (null, null), start))
    edges.push((*to*, (null, null), end))
for each highlighted range (*pseudo*, *argument*, *from*, *to*)
    if necessary, map offsets (*from*, *to*) to fragment offset space
        * see note 1
    if *from* < *to*
        edges.push((*from*, (*pseudo*, *argument*), start))
        edges.push((*to*, (*pseudo*, *argument*), end))
sort *edges* by (*offset*, *layer*, end < start)
return *edges*

1. Be careful to ensure that all of the edge offsets are in the same space, be it DOM space or some other space. For example, in Blink, text fragments and ::selection ranges are in a "canonical text" space that accounts for CSS 'white-space', while other highlight ranges are in DOM space and need to be mapped.

Parts

We then need to *break the text into parts*. Each edge indicates that the text continues in a different layer or may need a different set of decorations, so each sliding window of two increasing offsets represents a piece of the text that can be painted in the same layer with the same decorations, and these are the parts. The pair of offsets may involve more than two edges (hence the *prev_offset* < *edge*.offset check), because multiple highlights can start or stop at the same offset.

This is O(*nm*) because we loop over the edges, each time doing a linear search over the active layers, which in the worst case is all of the layers.

type Part = tuple of (*layer*, *from*: offset, *to*: offset, *decorations*: list of Layer)
let *parts*: list of Part = []
let *active* = [false for each *layer* in *layers*]

```
let prev_offset: optional of offset = none
for each edge in edges
    if (prev_offset is not none) and (prev_offset < edge.offset)
        let topmost_active_index = last index of true in active
            * see note 1
        if topmost_active_index is not none
            let part = (layers[topmost_active_index], prev_offset, edge.offset, [])
            for each (layer_index, layer) in layers
                if active[layer_index]
                    part.decorations.push(layers[layer_index])
            parts.push(part)
    set active[index of edge.layer in layers] to
        true if edge is start
        false if edge is end
        * see note 2
    set prev_offset to edge.offset
return parts
```

1. If perf testing indicates this is a problem, we might want another data structure for *active* that allows for efficient peek-max, insertion, and arbitrary removal, e.g. a heap (providing the first two) plus a hash table (providing the last two).

2. This step will malfunction if there are overlapping ranges, but highlight overlays have no concept of overlapping ranges anyway — they are either active or not active. If overlapping ranges can appear in the input, they need to be merged before computing edges. CL:3529115 does this for custom highlights.

Painting the text fragment

To paint the text correctly, we need to do it in multiple passes: originating text shadows, originating text proper, each highlight's backgrounds and text shadows (alternating), then each highlight's text proper. The passes for text proper use the list of parts to ensure that the text is only painted by the topmost active layer, while the passes for backgrounds and text shadows are unaffected by whether or not any other highlights are active.

This is O(*mn*) because we loop over ranges and parts in loops over the layers.

```
paint text shadows for (from, to) in originating style
for each Part (part_layer, from, to, ...) in parts
    * see note 1
    if part_layer ≠ the originating layer (null, null), skip this Part
    clamp part offsets (from, to) to fragment offsets (from′, to′)
    paint decorations for the originating layer only
    paint text proper for (from′, to′) in originating style
```

for each *layer* in *layers*, except the originating layer (null, null)
    for each highlighted range (..., *from*, *to*) of type (*layer*.pseudo, *layer*.argument)
        * see note 1
        clamp range offsets (*from*, *to*) to fragment offsets (*from′*, *to′*)
        paint background for (*from′*, *to′*) in *layer* style
        paint text shadows for (*from′*, *to′*) in *layer* style
for each *layer* in *layers*, except the originating layer (null, null)
    for each Part (*part_layer*, *from*, *to*, *decorations*) in *parts*
        * see note 1
        if *part_layer* ≠ *layer*, skip this Part
        clamp part offsets (*from*, *to*) to fragment offsets (*from′*, *to′*)
        paint decorations for each Layer in *decorations*
        paint text proper for (*from′*, *to′*) in *layer* style

1. If perf testing indicates this is a problem, we can maintain separate lists of ranges and/or parts for each layer (or at least each layer pseudo). CL:3426940 already does this for ranges when painting each text fragment.

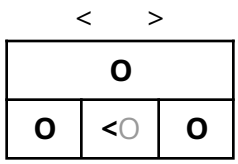
Problems

1. **Impedance mismatch between fragments and ranges.** DOM ranges are defined relative to a pair of text nodes, so querying all of the highlight ranges that touch a given node is easy, but querying only those that touch a fragment is hard.
2. **The outputs of the preprocessing algorithms are not cached.** Every time the text fragment painter is rerun, even if there were no highlight changes involving the associated node, we compute the layers/edges/parts from scratch.
3. **The preprocessing algorithms are not incremental.** If there are any highlight changes, we compute the layers/edges/parts from scratch. Perhaps it's possible to extend these algorithms to update their outputs over time?
4. **The painting algorithm is not incremental.** If there are any highlight changes, we repaint all of the text fragments associated with the node in full. Perhaps we can avoid painting some fragments or parts of fragments (but see problem 1)?

Incremental preprocessing [WIP]

To explain how incremental preprocessing would work, let's use tables where the *edges* are written along the top, and the rows represent the *parts* after each step. For example, if we start with nothing highlighted then add a range for <::selection>, the table might look like this, where "O" means the originating layer, the topmost layer is in **bold**, and the other

decoration layers are in grey. Note that in reality, we actually list the topmost layer again in Part.decorations, but let's omit it here for readability.



Say there are ranges for <::selection> and some (::highlight), and we want to add a range for [::spelling-error]. The highlight painting order requires <::selection> to be on top, then [::spelling-error], then that (::highlight), then the originating layer "O".

When the [::spelling-error] range is added, we need to insert two *edges*, and if necessary, a new Layer into *layers*. To update *parts*, we need to add the layer to all the Part.decorations between "[" and "]", updating the topmost Part.layer if necessary (step 2). If no other *edges* are at "[" or "]", we also need to split the *parts* straddling "[" or "]" respectively (step 1).

	(	<	[	)	>	]
initial parts (**topmost**, other decorations)	**O**	**(**O	**<**(O		**<**O	**O**
1. split last parts that start before [and/or]	**O**	**(**O	**<**(O	**<**(O	**<**O	**O**	**O**
2. update all parts between [and]	**O**	**(**O	**<**(O	**<[**(O	**<[**O	**[**O	**O**

If the same range for [::spelling-error] is later removed, we need to remove two *edges*, and if there are no other ranges, remove the Layer from *layers*. To update *parts*, we need to remove the layer from all the Part.decorations between "[" and "]", updating the topmost Part.layer if necessary (step 3). If no other *edges* are at "[" or "]", we also need to delete the *parts* inside "[" or "]" respectively (step 4).

	(	<	[	)	>	]
3. update all parts between [and]	**O**	**(**O	**<**(O	**<**(O	**<**O	**O**	**O**
4. delete parts that start at [and/or end at]	**O**	**(**O	**<**(O →		**<**O	← **O**	