Representation and implementation of Paint Lists in Blink

chrishtr@, pdr@, leviw@

September 2014

This is a design document for how to implement Paint Lists in Blink, and efficiently transfer them to the Chromium Compositor (cc). It does not cover in detail how cc should translate that display list into content on the screen; that is covered by other documents, such as this and this.

Status

There is an ongoing effort to prototype this design in Blink under the runtime flag SlimmingPaint.

The data structure described below is optimized for clearly describing the algorithms involved. We fully expect to optimize the actual representation for maximum efficiency, for example by factoring out the transform and clip trees. However, the data structures in

this document are represented in a way such that optimizations will gain only O(1) improvements in algorithmic performance.

Overview

Web pages in Blink are drawn to the screen in various phases. The major phases are Style, Layout and Paint. Paint is responsible for following the Web Painting Algorithm in order to draw the given Render tree on the screen. In practice, “drawing on the screen” means issuing a sequence of low-level commands, such as “draw rectangle in green” or “put a run of text in this font at this size and location”. With impl-side painting, the output format of painting is a single SkPicture which contains a list of these commands.

The Blink render tree has code to implement the Web Painting Algorithm. The same render tree objects are used to implement layout. There is currently an effort to move the painting code into its own directory, to make the distinction clearer.

The rules for painting web pages involve a somewhat complicated multiple recursive walks over the elements in the DOM, painting different pieces of them at different times. For example, the background of an element and its block level & floating descendants are painted before the text of the element, and the block-level background are painted before the floating ones. There are a number of rules, all of which are important.

The Web Painting Algorithm can be thought of as painting in a number of phases, for example background before border before text content. These phases are represented explicitly in Blink (see here and here), and are used in the natural way to communicate state during the recursions mandated by the painting algorithm. This means that there is a **natural breakdown of the existing paint code** into display items, one for each paint phase of each Render object. These atomic display items are the building blocks of the proposed format.

Details

Representation

For the purposes of this document (see “Status” at the top of this document for notes about more efficient representations), we propose to represent the display list (called here `PaintList` for lack of a better name) according to the following pseudocode:

```
typedef Vector<DisplayItem> PaintList
class DisplayItem {
public:
     Id id; // Tracks the renderObject and phase.
}
```

There will be one `DisplayItem` for each paint phase of each Render object. The vector is in paint order. DisplayItem, and other subclasses of it, are defined in the Layerization based on display lists design doc.

Algorithm to construct

The data structure is initially filled in a straightforward way: as we traverse the Render tree and need to paint a particular phase of a particular Render object, we write the code as follows:

```
void Painter::paint(PaintInfo& paintInfo, ...) {
    DrawingRecorder recorder(..., &m_renderer, paintInfo.phase, ...);
    paintInfo.context.drawRect(); // for example.
}
```

Here `DrawingRecorder` is an RAII object which automatically creates and stores a `DisplayList` object representing just the piece of recording desired. The process of building and appending display items is encapsulated by `DrawingRecorder`, an RAII object that automatically stores a `DisplayList` for all drawing commands that are in its scope.

Algorithm to mutate

See also this doc for some examples of the algorithm in action.

When paint invalidations happen, they are represented as bits on RenderObjects indicating that they need to be painted. (These paint invalidations are now available due to the substantial completion of the Paint Invalidation after Layout & Compositing project.)

The algorithm takes advantage of two invariants of the paint invalidation system:

1. RenderObjects that are **not** invalidated for paint do not need to re-generate their DisplayItems, and can reuse them from the previous frame.
2. RenderObjects that are **not** invalidated for paint do not change the relative paint order of their DisplayItems to other non-invalidated RenderObjects.

The algorithm, in pseudocode, is the following:

```
1 void ViewDisplayList::updatePaintList() {
2   PaintList updatedList;
3   currentDisplayIt = currentPaintList.begin();
4
5   for (newDisplayItem in iteratorForDisplayItemsOfRenderTree()) {
6     if (!wasInvalidated(newDisplayItem)
7         && repaintIt = findExistingDisplayItem(newDisplayItem))) {
8       // Copy over display items until we hit the repaint.
```

```
9         do {
10          updatedList.appendIfNotInvalidated(currentDisplayIt);
11        } while (++currentDisplayIt != repaintIt);
12        ++currentDisplayIt;
13      }
14      // Copy over the newitem.
15      updatedList.append(newDisplayItem);
16    }
17    // Copy over any remaining items that were not invalidated.
18    do {
19      updatedList.appendIfNotInvalidated(paintListIt);
20    } while (++paintListIt);
21  }
```

This algorithm has running time equal to O(n) + O(m), where n is the number of nodes in the render tree, and m is the number of invalidated RenderObjects. Invariant 1 (above) allows us to avoid recomputing a display item for a non-invalidated RenderObject.

Hints, 3D transforms, clips, filters and other special cases

The above algorithm works for “simple” display items. what about more complex scenarios involving transforms, clips etc?

Filters, masks and opacity

If a filter is present on a RenderObject, that means it must be a stacking context. Opacity is a special case of filters. Masks also imply stacking contexts.

Therefore we can easily bracket painting of the contents of that RenderObject and its children with a begin and end `FilterDisplayItem` object. However, to maintain correct paint order, `iteratorForDisplayItemsOfRenderTree()` must always output the start and end `FilterDisplayItem` objects so long as the render object with the filter or any of its children are invalidated.

The `FilterDisplayItem` start & end objects also serve to hint to the compositor that the display items between them should be considered for a composited layer.

3D transforms

If a 3D transform is present on a RenderObject, that means it must be a stacking context. Therefore all display items of children are nicely nested inside of it. The situation is basically the same as for a filter (see above), except using a `TransformDisplayItem` start/end pair. Perspective and preserves-3d can be represented in the `TransformDisplayItem` objects to which they apply.

Non-scrolling clips

Non-scrolling clips such as due to overflow:hidden CSS, clip non-self-painting descendants but do not induce a stacking context. Therefore we need to represent the clip as an independent start/end display items that brackets clipped content from children. The children which are not clipped (absolute- or fixed-postion children, unless the parent is absolute- or fixed-position itself) have their display items issued outside of the clip start/end boundaries.

Scrolling clips

Scrolling clips, such as due to overflow: scroll CSS, have clipping behavior identical to non-scrolling clips. The additional complications are that they need to represent the information necessary for composited scrolling. Each group of descendants that is clipped by the scroll needs to be individually clipped and marked with a transform hint. Furthermore, we need to paint all content, even content that is clipped, in anticipation of the compositor performing a composited scroll to make it visible.

These will then be grouped together by the compositor code that processes the paint list in order to transform them together. (Note: factoring out a transform tree will likely make this simpler to implement in the compositor.)

Note: clipped content can be unboundedly large. We will likely need to implement a limit on it, i.e. only compute a valid painted output for a certain region of the scrollable content, and communicate to the compositor a “valid rectangle” equal to the bounds of this region. A similar optimization needs to be made at the frame level.

Backface visibility

Backface visibility does not induce a stacking context. Therefore it faces similar challenges to scrolling clips and multiple hint regions.

Reflection

All reflections, including ones involving 3D transforms, will be painted via Blink paint paths. It should be painted as if the reflection was represented explicitly in the Render tree with the transform. Invalidation of the reflection source invalidates the reflection.

Compositor-driven animations

Compositor-driven animations may want to have modifications to the transform, scroll or opacity hints that correspond to them. Otherwise there is nothing special.

Display items not painted by Blink

These items include videos, compositor-drawn Canvas, scrollbars, plugins, etc. For each of these we must output a special `DisplayItem` that represents them, and which contains information to identify them, and also serves as a hint to the compositor about separately compositing that content.

Optimization 1: avoid processing non-invalidated RenderObject subtrees

We can optimize this algorithm to avoid painting subsequences of display items for non-invalidated RenderObject subtrees. The most logical scenario in which this is possible is if the subtree root is a stacking context.

To implement this we need only modify the `iteratorForDisplayItemsOfRenderTree()` subroutine to skip over sequences of display items from the subtree, except for the last display item of the subtree. (This last display item would be cached explicitly from the previous frame for easy lookup.) Lines 8-12 of the algorithm above would add the display items in the sequence up to the last one.

With this optimization, it is possible to reduce the total computation time to O(n') + O(m), where n' is the number of render objects that are either invalidated or have at least one child which is invalidated.

Note: this optimization is **very important** for obtaining performance that is at least as good as the existing compositing system. In particular, it's important to implement the optimization for stacking contexts below other ones, to avoid scenarios where nested composited layers get unnecessarily re-painted.

Optimization 2: avoid processing RenderObjects that don't intersect an invalidated RenderObject

In this optimization, we would ignore all RenderObjects that do not intersect the bounds of an invalidated RenderObject. The paint order will be correct, since the relative order of display items is preserved. (Lines 8-12 do this for each display item that intersects but is not invalidated.) This optimization can be implemented by simply having `iteratorForDisplayItemsOfRenderTree()` ignore non-intersecting RenderObjects.

One downside of this approach is that large changes to the PaintList would sometimes result from small invalidations. For example, if one RenderObject that overlaps nothing was invalidated, then its display items would move all the way to the front of the new PaintList. This may result in a large false-positive raster diff in the compositor. One potential way to improve this would be to try to insert the invalidated items near their previous position in the PaintList.

Optimization 3: start iteration at the enclosing stacking context

If we can prove that all invalidated items are within given stacking context(s), we should be able to start the `iteratorForDisplayItemsOfRenderTree()` at the first display item for those stacking context(s). We should be able to be in this situation often (always?), since stacking contexts are essentially self-contained for paint (this is why the Blink compositing code is able to define GraphicsLayer boundaries at RenderLayer granularity).

This should allow us to reduce the total computation time to O(n'') + O(m), where n'' is the number of stacking contexts that contain an invalidated RenderObject.

Sending the display list to the compositor thread

The compositor thread needs its own copy of the paint list, because it operates asynchronously from Blink. To make this copy is easy: just walk the PaintList vector and copy it. Note that the `SkPicture` contents are not a deep copy, since those are thread-safe refcounted.

Representing sub-RenderObject damage rects

Today there are some optimizations in layout for sub-render-object changes. For example, if a block is enlarged by one pixel, then we issue a paint invalidation rect that is only one pixel wide in that dimension.

Fitting this into a PaintList architecture directly requires some changes. One reasonable possibility is to annotate each display item for an invalidated RenderObject with the (smaller) actual damage rect. The compositor would then use this information to obtain a more precise diff rectangle.

Alternatively, we can consider removing the ability for Blink to issue damage rects smaller than a RenderObject. This might be a better choice if it turns out the time to compute such rects and the complexity of doing so outweighs what win in rasterization time we can get.

For reference: implementation in Firefox

Firefox has an implementation that is very similar to this design, including display items for various phases. Further documentation is here. The nsDisplayList.h file gives some details, including noting that they reuse the same data structure for hit testing and composited scrolling. Types of items in the Firefox display list appear to be enumerated here.

This document also gives useful details on the way that display lists are represented in Firefox across stacking context boundaries (TL;DR: the same as described here, except they have an explicit tree representation of stacking contexts as sub-display lists).

Composited layers for Firefox are discussed here.