Compositing Corner Cases

ajuma@, vollick@, 9/22

Here's a list of odd compositing cases not well covered in previous design docs.

1. Composited scrolling and scrollbars.
2. Pinch zoom.
3. Masks.
4. Reflection.
5. Filters.
6. Preserve-3d / 3d-sorting.
7. Hit testing.
8. TODO: Persistent resources (mailboxes, video provider)
9. TODO: Scroll snapping

The goal of this doc is not to settle conclusively on the details, but tell a plausible story, both about how these cases can be handled correctly, but also how the implementation could be staged.

Guiding principle: blink should know as little as possible about these auxiliary data structures. The recording code should simply interact with a display list API that permits the insertion of a variety of start and end markers.

For example,

- This is the start of stuff that will scroll like <blah>, and this is the end of that stuff.
- This is the start of stuff that will animate like <blah>, and this is the end of that stuff.

Blink display list API additions

Below is a sketch of what this display list interactions for these corner cases could look like it. The remainder of the document will describe how the compositor could support this API. Bikeshedding welcome, but keep in mind that I've intentionally not sweated the details.

It's important to note that, as Slimming Paint works today, we may have a "`pushXXX`" in one `blink::DisplayList` and the corresponding "`popXXX`" in another. Once the micro recordings are flattened, though, we will have proper bracketing.

```
// Masks
Token maskToken = context.beginMask();
// Add commands for mask.
context.endMask();
context.pushMask(maskToken);
context.popMask();
```

```
// Filters
context.pushFilter(...); // Opacity is a filter.
// Add commands for the filtered stuff.
context.popFilter();

// Preserve-3d
context.begin3DRenderingContext(contextId);
// Draw operations for visuals in this rendering context.
context.end3DRenderingContext(); // Unambiguous as per spec.

// Wheel-event, touch-event, and non-fast-scrollable regions are added
// as one shots.
context.addWheelHandlerRect(...);
context.addTouchHandlerRect(...);
context.addNonFastScrollableRect(...);

// Animations
Token transformToken = context.pushTransform(...);
Animation anim = context.addAnimation(transformToken, ...);
anim.pause(); // All the WebAnim controls we need.

// Clipping
Token clipToken = context.createClip(...);
context.pushClip(clipToken);
// Record clipped stuff.
context.popClip(); // Returns popped clip token.

// Scrolling
Token clipToken = context.createScrollClip(...);
context.pushClip(clipToken);
// Record clipped stuff.
context.popClip();
context.addScrollbar(...);
context.addScrollCorner(...);
```

We're punting on reflections. We'll simply record that content twice. If the content contains video, plugins, etc, we'll insert two copies of the placeholder for that content. If the reflection is involved in a compositor-driven effect (scrolling, animation, eg), we will force those effects to the main thread. Down the road, if we decide that we want to do properly instanced reflections, that API wouldn't be too tough to support. It might look like this.

```
// Reflections
Token instanceToken = context.beginInstanced();
// Add commands for the stuff that will be instanced.
context.endInstanced();
// Set up other parameters like ctm and mask. Eg, you might have a
// reflection with a gradient. You'll want to inform the context
// of the reflection transform and handle the application of the mask.
context.pushTransform(...);
context.pushMask(reflectionGradientMaskToken);
context.drawInstance(instanceToken);
context.popMask();
context.popTransform();
```

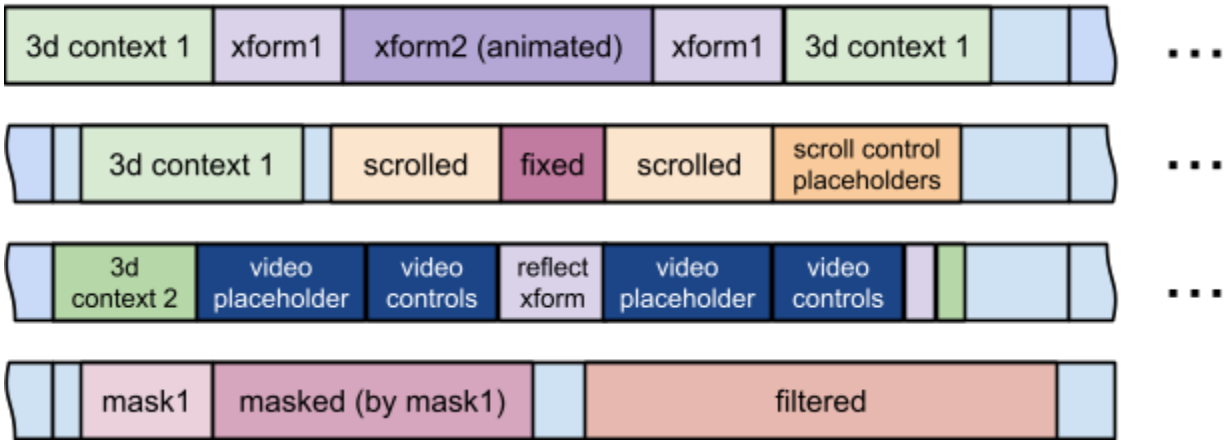
The Forest

We propose that, instead of a layer tree, the compositor operate on the following data structures,

- A transform tree.
- A clip tree.
- A filter tree (encompassing opacity, blur, etc).
- A scroll tree.
- A flat list of "draw objects" (layers at first, ultimately display list items).

The presumption is that nodes in the trees will be extremely rare data and that updating them will be, for all intents and purposes, a trivial constant cost. These trees will be built by processing the display list[1]. If we take all the recorded information, including the decorations added by the API above, and arrange it in paint order, we'll get our display list. It might look something like this,

[1] After Slimming Paint we will cache micro-SkPictures per paint phase wrapped in `blink::DisplayList` objects. This terminology is a bit unfortunate. In this document we'll use the term *display list* to describe the entire flattened list of everything that's been recorded so far and *display list item* to refer to these micro recordings.



The rectangles show the extent of the effect of various properties. Each rectangle would contain many draw operations. For example, the “xform1” rectangles in the above diagram would contain all the draw operations affected by xform1. I was also attempting to illustrate the nesting of transforms with “xform2”. All of the draw operations in the xform2 rectangle would be affected by xform1 * xform2(t) (remember, xform2 is animated!). Essentially, we will have a `pushTransform` at the beginning of the first instance of the xform1 and xform2 rectangles, and a `popTransform` at the end of the last instance of each of the xform1 and xform2 rectangles.

Since the appropriate pushes and pops will be included in this stream of data, constructing the various trees that will be described in the next section can be accomplished in a linear pass across the document.

Incremental updates should also be possible provided we have a stable identifier to the display list items. What this identifier should be is debatable, but a proof of existence is the following: `str(hash(&element)) + “my-paint-phase”`.

Hit Testing

Currently hit testing is done on `LayerTreeImpl`. That is, we walk the entire layer tree, including some layers that may not draw, in order to find the layer behind a screen space point.

Although it may be expensive, it is certainly plausible that we could iterate through all the elements of our display list and, because we know their bounds and transform, determine if they cover the given point. Unfortunately, we haven’t yet accounted for things with no visual content that do participate in hit testing. Touch handler regions, for example.

We propose that display list items with bounds, but no recording, be added to the display list for hit testing purposes.

If the naive approach of walking the whole display list turns out to be prohibitively expensive (and we expect that it will), we could certainly employ a standard spatial data structure.

The Scroll Tree

We propose the addition of a scroll tree to facilitate simpler compositor driven scrolling and bubbling. There will be two flavors of scroll node, scrollers and scroll inhibitors (bikeshedding welcome).

A scroller will know the following

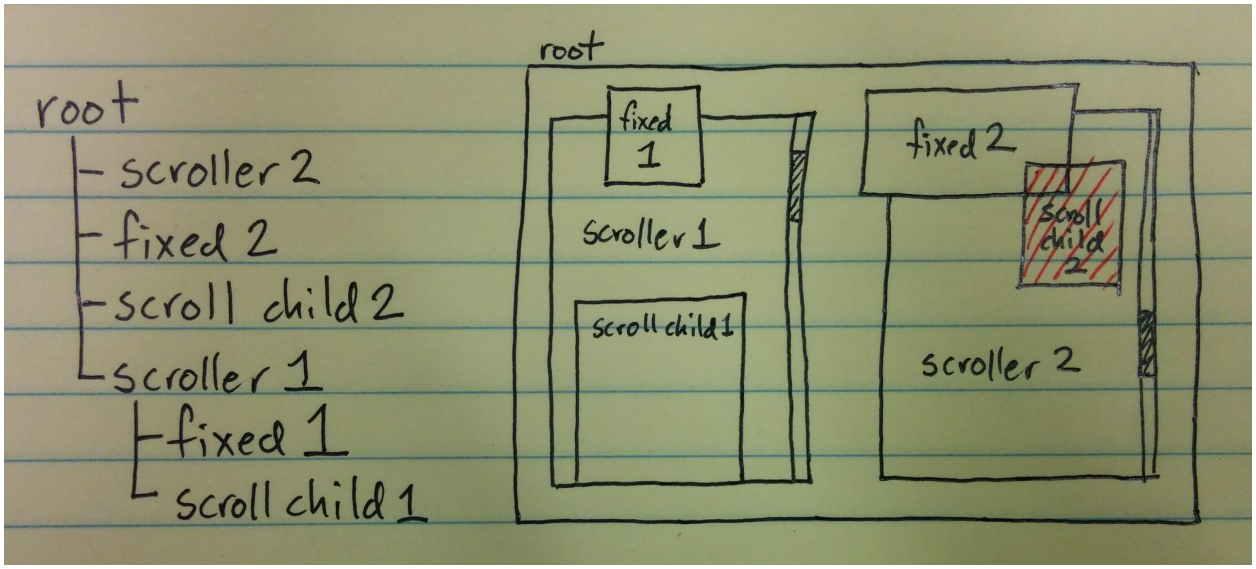
- Their corresponding transform tree node (this is how it effects a scroll).
- Scroll extents.
- Scroll delta. i.e., the bit of scrolling done on the compositor that the main thread has not yet been informed of.
- Corresponding scrollbars, so that thumbs can be updated, etc.

A scroll inhibitor would encompass the role of non-fast-scrollable, touch-handler and wheel-handler regions as well as main-thread-scrolling-reasons. These nodes would know about

- Their transform and their region (if any) for hit testing purposes.
- Main thread scrolling reasons (if any).

Given this information, the compositor thread will, when it needs to effect a compositor-driven scroll, use hit testing to find the affected display list item, find the corresponding scroll tree node, then walk upward to see if any ancestor scroll inhibitor will prevent the application of the scroll delta on the compositor thread. If not, the scroll delta will be applied the the scroll tree node, and, if there is any unconsumed scroll delta, this will be propagate up the scroll tree to the ancestor scroller causing overscroll glow, if necessary. This logic mimics the current behavior of LTHI.

This scheme greatly simplifies the compositor logic. Consider the following case.



A scenario that used to be complex

Here we have a scroller that did not form a stacking context (scroller 2) and its scroll child therefore needed a special pointer so that the compositor could update the scroll transform for both layers. We've also got a problem with fixed 1, a fixed position descendant of scroller 1. It needed to be positioned as it is in the layer tree so that it would stack correctly, but it also means that inherits an unwanted scroll transform that, in compositor parlance, needs to be "compensated" for.

This all becomes much simpler in the new scheme. The display list will roughly look as follows. I've color-coded the association between the display list items and the transform tree below.

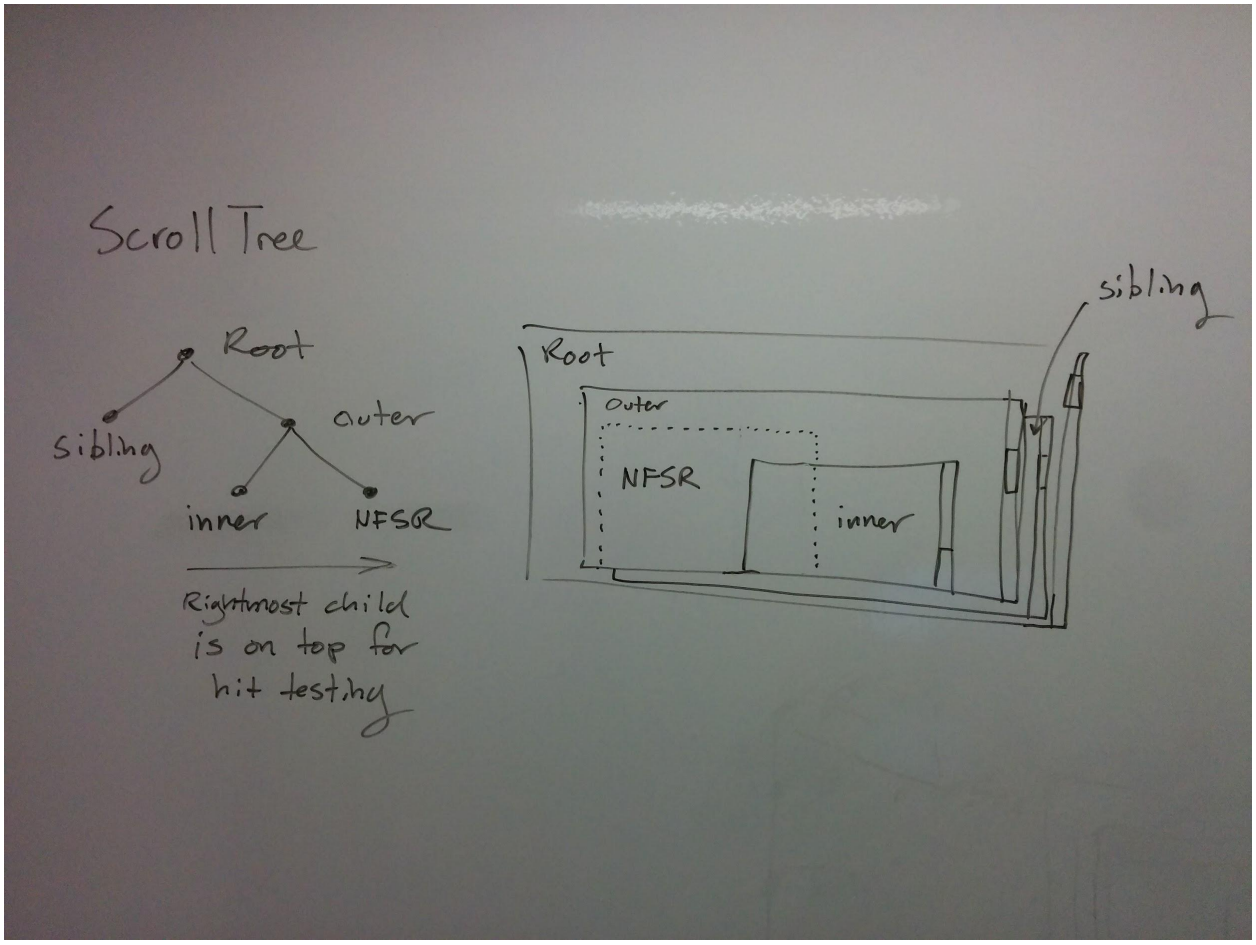
- root
- scroller 1
- scroll child 1
- fixed 1
- scrollbar placeholder for 1[2]
- scroller 2
- fixed 2
- scroller 2
- scrollbar placeholder for 1

The transform tree would look like this:

- root transform
    - scroll transform 1
    - scroll transform 2

[2] Scrollbars will be treated much like replaced content such as video. Blink will record placeholders for the scrollbars in the display list so that stacking is established for the compositor

In this scheme, to effect a scroll, we only need to update a transform of the appropriate color. No special logic for scroll compensation or scroll children is required. Here is a more detailed example involving scroll inhibitors and its corresponding scroll tree.



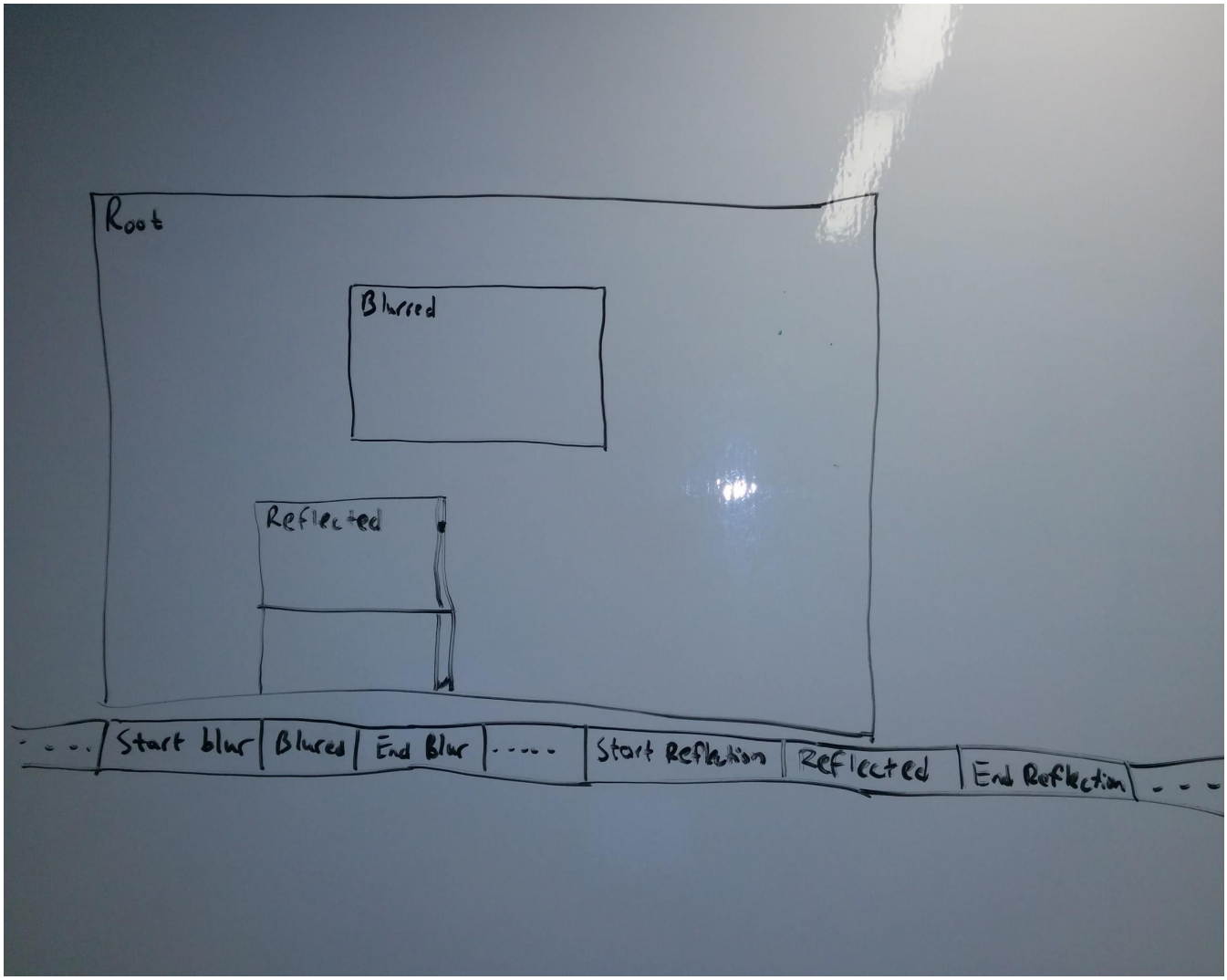
A complex scrolling scenario

Pinch Zoom

One option for handling pinch zoom is to add a third flavour of scroll node in the scroll tree to handle the pinch zoom stuff. Another option is to special case it and treat it independently of scrolling.

Masks, reflection, and filters

Masks, reflection, and filters will be annotated within the display list. The annotations will appear just before and just after the display items involved in the effect. For example:



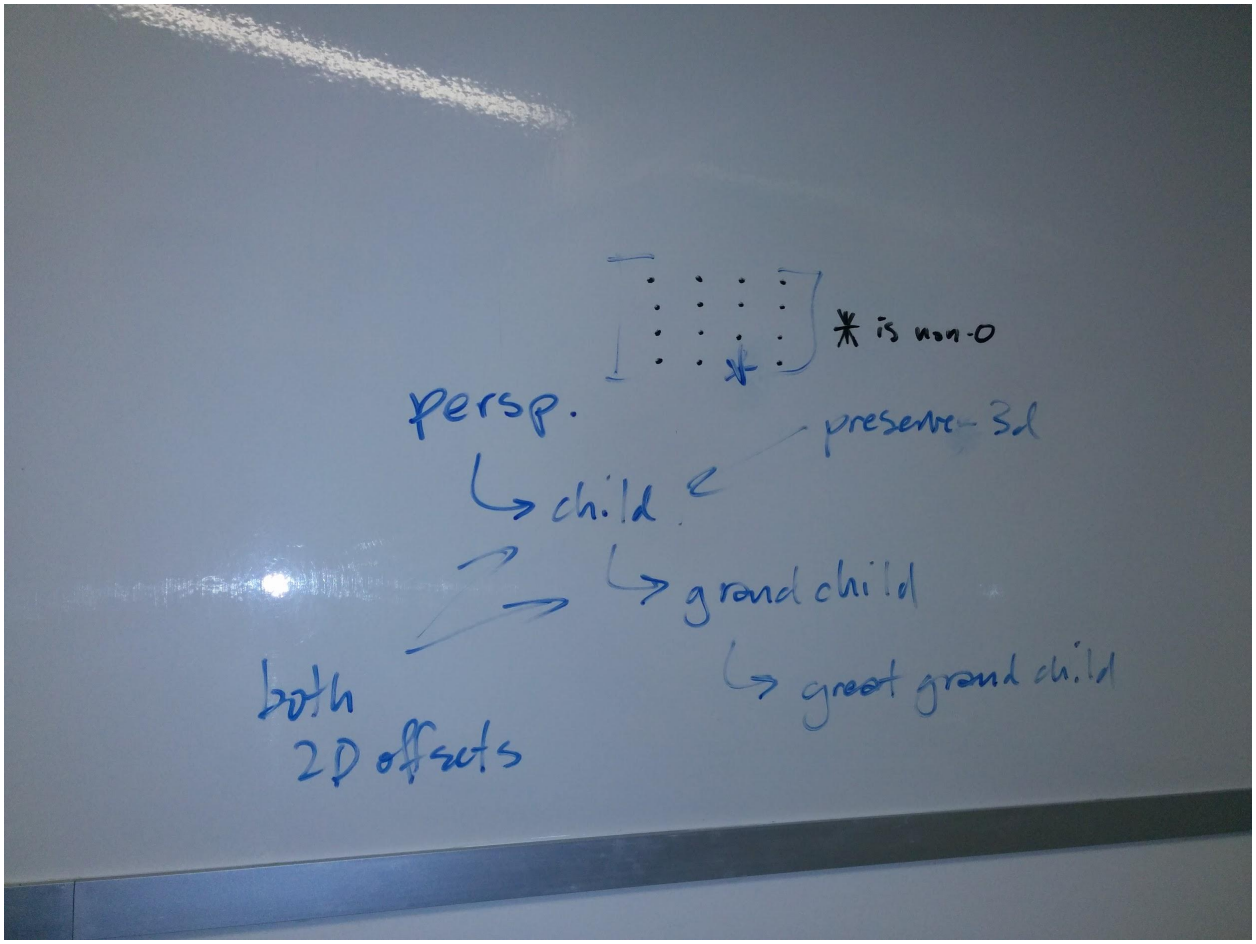
A scenario with a filter and a reflection

Opacity will be treated as just another kind of filter. The compositor may use these annotations in order to construct a filter tree, in order to more efficiently determine which subtrees can be skipped (those that have opacity 0), which subtrees require LCD AA to be turned off (those that have opacity less than 1), and to track compositor-driven animations of these properties.

We've decided to punt on reflection. We will simply rerecord the reflected content and disable any compositor driven effects that may involve the reflection (eg, reflected videos, scrollers, etc).

Preserve-3d and 3d sortinghandwaving pass decoding twice

Handling preserve-3d correctly is tricky, and requires more information than we have in the transform tree, as illustrated by the following example:



A scenario with perspective and preserve-3d

In this case, we only have a node in the transform tree for persp (because it has a truly 3d transform). child has preserve-3d, but grandchild doesn't. This means that an unflattened transform needs to be propagated as far as grandchild, but not as far as great grandchild[3].

The transform at great grandchild is formed as follows:
flatten(M_persp * T_child * T_grandchild) * T_greatgrandchild

where M_blah represents an arbitrary matrix at node blah, and T_blah is a translation.

We could conceivably demarcate the draw ops for great grandchild with the following annotations

- Associated (in a flattened sense) with transform node persp.
- Needs to apply offset for grandchild before flattening.
- We will also need to decorate the display list with 3d rendering context identifiers so that 3d sorting can happen on the appropriate sets of quads.

[3] See the transform spec for details.

Animations

Animations will be created by annotating the display list with a structure similar to WebCompositorAnimation, which will contain all the information needed by the compositor to drive the animation. The compositor will associate animations with nodes in the transform or filter trees. The ticking of animations will cause the values stored in these nodes to get updated (that is, the nodes will be the observers of the animation). Blink will be able to modify in-progress animations using the unique id associated with the display item that owns the animation.

Staging Plan

- Finish main thread refactor.
    - I.e., construct clip, transform and filter trees and use them to compute visible content rects. Old layer tree will still be used and committed.
- Construct the scroll tree from the layer tree and use it to drive compositor scrolling.
- Transition to computing all draw properties using the new data structures. This will be done on both threads. TODO(ajuma): what's the story for render surfaces on the compositor thread?
- Transition to constructing new trees from the display list items.
    - Prior to this step, we've been effectively using the paint order list of cc::Layers as our display list, gleaning all the information we need to construct our trees by asking questions of the layers as we traverse the layer tree in paint order. Bit by bit, blink will start to decorate the micro recordings using the new API. As this happens, we'll conditionally use the information from the micro recording rather than the layer. As all cc clients begin to supply this information in the micro recordings, we can eliminate the layer properties altogether and the associated conditionals.
    - After this step, the compositor should be constructing the new data structures by traversing the layer tree, but only accessing information from the micro recordings.
- Flip the switch (on a per-client basis).
    - No more layers in the compositor API. All compositor clients will instead supply a decorated display list.
    - Texturization/layerization/squashing will now be done in the compositor.
    - After this step cc::Layers will still be used, but they will be an implementation detail of the compositor; they will be the unit of squashing. This should let us continue to take advantage of existing impl-painting machinery.

Appendix1: Plan for re-factoring main-thread CalcDrawProps

The plan and a proposed patch series can be found at:
https://docs.google.com/a/chromium.org/document/d/1SAJI1F1OX98lYrDPewe5X2JIG-vzEbmRZGVHKP5dNnA/edit

Appendix2: Open Questions

Should canvas be treated the same as video?

What replaced things can be transparent? Can <video> and <canvas> be transparent?

Answer: Yes, both <video> and <canvas> can be transparent.