Transform Trees in Blink

Jeremy Roman <jbroman@chromium.org>, February 2015

Background

For the compositor layerization phase of the Slimming Paint effort, the Chromium compositor (henceforth *cc*) will need to be able to obtain information about the transforms that apply to the layers it creates, how they relate to one another, and how they can be updated.[1]

While it is possible to include enough information to derive this structure inline in the display item list, being able to efficiently communicate between Blink and cc in terms of changes to transform nodes is expected to be useful (e.g. an element's CSS transform could be updated by notifying cc about the change to the transform node, similar to the current *direct compositing update* optimization).

For any layer that the compositor wishes to rasterize, the appropriate draw and screenspace transforms can be computed by finding the corresponding transform node and multiplying the transformation matrices of (some of) its ancestors.

Interface

Transform trees are exposed across the Blink API boundary through a class called `WebDisplayItemTransformTree` implemented in Blink, which contains a compact representation of the transform nodes and a sorted vector mapping ranges of the display item list to transform nodes. In almost-valid C++:

```
class WebDisplayItemTransformTree {
public:
    struct Node {
        size_t parentNodeIndex;
        OpaqueIdentity id;
    };

    struct RangeRecord {
        size_t displayListBeginIndex;
        size_t displayListEndIndex;
        size_t transformNodeIndex;
    };

    BLINK_PLATFORM_EXPORT size_t nodeCount() const;
    BLINK_PLATFORM_EXPORT const Node& nodeAt(size_t index) const;

    BLINK_PLATFORM_EXPORT size_t rangeRecordCount() const;
    BLINK_PLATFORM_EXPORT const RangeRecord& rangeRecordAt(size_t index)
const;
```

[1] See also enne@'s post, "Property tree update", to paint-dev on Feb 20, 2015.

```
};
```

The opaque identity of each node will be derived from information recorded during paint (e.g. the display item client responsible for the transform), and is expected to be stable over time.

Construction

The transform tree structure can be created during a linear walk through the merged tree (during a fresh paint, during the merge step, or afterward). In abstract, for simple cases:

1. Begin with the *current index* and *range begin index* both zero, and the *ignored begin stack* containing only a single zero.
2. Create a transform node with no parent (the *root node*) and set it as the *current transform node*.
3. For each display item:
    a. If it is the begin item for a significant 3D transform (or other important feature), then:
        i. If the range begin index is earlier than the current index (i.e. a non-empty range of display items exists), then create a new range record beginning at the range begin index, ending at the current index, and pointing to the current transform node. Regardless, update the range begin index to be the index of the next display item. (This step is called *finishing the current range*.)
        ii. Find an existing node whose identity matches the begin item, or if none exists, create one. Its parent should be the current transform node. Make it the new current transform node.
    b. If an end display item has been reached, and the top of the ignored begin stack is zero, then the item exits the current transform node.
        i. Finish the current range.
        ii. Set the current transform node to its parent.
        iii. Pop the ignored begin stack.
    c. If the item is begin item, but doesn't require a new transform node, increment the top of the ignored begin stack.
    d. If the item is an end item, and the top of the ignored begin stack is positive, decrement the top of the ignored begin stack.
    e. Increment the current index.
4. Finish the current range.

I believe that more complicated cases, like fixed-position elements, correspond to reasonably natural adaptations of this algorithm.

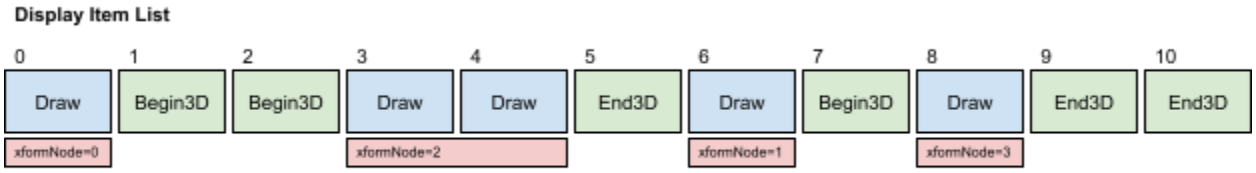
Merge

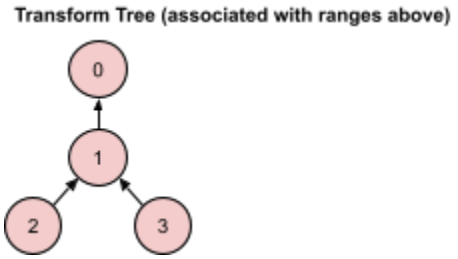
This structure also permits generating a new transform tree using information from a previous one. A new transform tree can be built as the merge proceeds. During most display items, it proceeds as above. When a cached subtree is merged, the transform tree for that interval from the original display item list can be consulted to identify which transform nodes apply to the cached subtree. For each such node, then, a corresponding node can be found or created in the new transform tree, based on its opaque identifier.

Thus, for each cached subtree, the transform tree merge can be done in time proportional to the number of transform nodes, rather than the number of display items, in the cached subtree.

Example

Here is a simple example of a display list consisting only of drawing and significant 3D transforms, showing the resulting transform tree and for each range of display items separated by begin/end transform items, which transform node applies.





Nodes:

- {index=0, parent=none}
- {index=1, parent=0}
- {index=2, parent=1}
- {index=3, parent=1}

Range records:

- {displayListBegin=0, displayListEnd=1, transformNodeIndex=0}
- {displayListBegin=3, displayListEnd=5, transformNodeIndex=2}
- {displayListBegin=6, displayListEnd=7, transformNodeIndex=1}
- {displayListBegin=8, displayListEnd=9, transformNodeIndex=3}

Extensions

Fixed-position elements

Fixed-position elements get their own `blink::Layer` (formerly `RenderLayer`) instance, because they paint before (in the negative z-order list) or after (in the positive z-order list) in-flow descendants, depending on z-index.[2] They are not within the scope of their parent layer's overflow clip; rather, `LayerClipper` is responsible for computing the appropriate clip to apply later on.
A *fixed position container layer* is a blink::Layer which establishes a containing block for fixed-position descendants. The root layer is a fixed-position container, as are layers for elements which have the CSS transforms, etc. To my knowledge, fixed position containers always establish stacking contexts.

Each fixed position container will emit bracketing display items which:

- Transform to a coordinate space where (0, 0) is at the appropriate corner. (CSS transforms already do this, and the viewport can do it implicitly.)
- Indicate the containment of fixed position descendants. (Ideally this is separate from the previous, but we could understand this implicitly.)

Each fixed position descendant will emit bracketing display items which identify it as fixed position, and give the offset from the current coordinate space to the coordinate space of the fixed position container. There can be no transformations other than a 2D translation between the two, since one could only have been created by a CSS transform, which would have established a closer fixed position container.

Because this allows "returning to" a transform node with an offset (since the fixed position container does not necessarily create a transform node, e.g. if the transform is the identity), each range record must be augmented to include a 2D offset from the origin of the transform node. Avoiding this would require making the painting code aware of the heuristics for creating transform nodes, which seems desirable to avoid.

Canvas replay

When replaying to a canvas, the fixed position containment display items is a no-op, and the fixed position display items perform the translation to the fixed position container's coordinate space (and the end item undoes that translation, either by applying the inverse or by restoring the canvas state).

---

[2] See the treatment of paint order for positioned elements in the CSS 2.1 spec.

This will generate a correct image of the state of the page (as it was at paint time). It does essentially involve counter-scrolling; however the canvas API does not provide a way of restoring just the transform from earlier, and this is equivalent.

Changes to transform tree construction

Fixed position content will be attached directly to the transform node of its fixed position container. This removes the need for counter-scrolling in the composited path. Rather, if any transform (e.g. scrolling) occurs in the compositor, it will occur on a transform node that is not an ancestor of the fixed position content, and thus not affect it.

The construction algorithm is adapted as follows:

1. The *current transform node* (whose ancestors implicitly formed by a stack) and *ignored begin stack*, are replaced with two explicit stacks:

    a. the *current transform node stack*, which contains (transform node index, 2D offset, ignored begin) tuples; the offset may be non-zero in a case where a new transform node was not created (e.g. a CSS transform by a 2D offset where animation is not anticipated)

    b. the *current fixed position container stack*, which contains (transform node index, 2D offset) pairs, and is a subsequence of the current transform node stack (except for the ignored begin value), corresponding to those elements of the current transform node stack which form fixed position containers

2. Range records include both the transform node index and the 2D offset from the top of the current transform node stack.

3. When a new transform node is created, (new node, (0, 0)) is pushed on the top of the current transform node stack.

4. When a transform is seen but a node is not created (e.g. because the transform is simply a translation), an entry is pushed on the top of the current transform node stack with the same transform node, but an offset adjusted for the translation.

5. Elements are popped from the current transform node stack for both transforms that are significant, and those that are not, in order to deal with the offset.

6. When a *begin fixed position containment* display item is encountered, push the current transform node and offset onto the current fixed position container stack. When an *end fixed position containment* display item is encountered, pop it.

7. When a *begin fixed position* display item is encountered, push the transform node and offset from the top of the fixed position container stack onto the current transform node stack. When an *end fixed position* display item is encountered, pop it. We ignore the

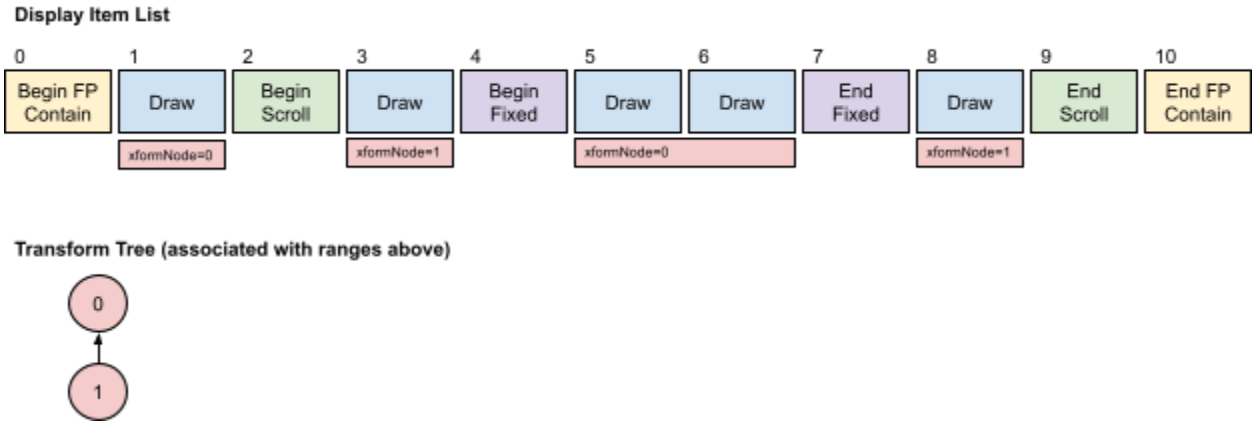
offset data from this display item, since by using the transform node from the fixed position container stack, we avoid having applied that change in the first place if the transform tree is respected.

Changes to transform tree merging

Any changes that would affect whether an element is a fixed position container, whether it is fixed position, or what the containing block of a fixed position element is, should cause paint invalidation of the subtree. (Tests for this case are certainly a good idea.)

As a result, this part of the transform tree will be reconstructed (rather than copied from a cached subtree) and should produce a valid transform tree equivalent to one constructed fresh.

Example



This is a simple example that might be generated for a scrollable area with fixed position content within it.