Top-down algorithm for computing GeometryMapper cache data

Chris Harrelson

This document spells out the exact algorithm for computing the caching data that drives the GeometryMapper class.This is a sub-document of Web Page Geometries; see that document for much more detail on GeometryMapper and its use cases.

Our task is to implement this method from the signature:

```
// Precomputes intermediate results so that future calls to any of the
// above methods with the given |ancestor_space| run in O(1) time.
void PrecomputeClippedRectsAndTransforms(const PropertyTreeState&
ancestor_state);
```

in a time- and space- efficient manner.

Implementation

```
struct TransformAndScrollState {
  int transform_id;
  int scroll_id;
}

struct ClipAndEffectState {
  int clip_id;
  int effect_id;
}

struct PrecomputedDataForAncestor {
  // Maps from a transform and scroll node pair that is a descendant of
  // the ancestor to the transform from any property tree state with those
  // two tree nodes to the ancestor state.
  // (Non-invertible matrices have a special representation, say a bit
```

```
  // in the transform ore representing as nullptr.)
  hash_map<TransformAndScrollState, gfx::Transform>
      to_ancestor_transforms_;

  // Maps from clip and effect node id to visual rect.
  hash_map<ClipAndEffectState, gfx::RectF> clip_to_ancestor_clip_rects_;

  // The intersection of all clips between local and ancestor, in ancestor
  // space.
  gfx::Rect clip_in_ancestor_space;
}

struct PrecomputedData {
  // Maps from ancestor property tree states for which we have precomputed
  // data to the data.
  hash_map<PropertyTreeState, PrecomputedDataForAncestor> data_;
  gfx::RectF ClipInAncestorSpace(PropertyTreeState descendant_state) {
    return data_.clip_to_ancestor_clip_rects_[
        descendant_state.ClipAndEffectState()];
  }

  gfx::Transform TransformToAncestorSpace(
      PropertyTreeState descendant_state) {
    return data_.to_ancestor_transforms_[
        descendant_state.TransformAndScrollState()];
}
```

The dependency DAG of the property trees is a graph with one node for each PropertyTreeState. It has pointers from a child node to its parent in the same tree, and also along between a node and the related ones it depends on (ScrollNodeData depends on a transform node, an EffectNodeData depends on a transform and clip, and a clip depends on a transform).

We will compute the above by recursively traversing the property tree states that are descendants of the ancestor, in topologically sorted order w.r.t. the dependency DAG of the property trees with pointers reversed. Call this order *top-down property tree recursion*.

Computing to_ancestor_transforms_

The transform between a descendant state and the ancestor state depends only on transform and scroll. Thus it suffices for `to_ancestor_transforms_` to map from TransformAndScrollState to the actual transform.

These transforms can be easily computed in a single top-down property tree recursion starting at the ancestor state, and multiplying the ancestor transform/scroll's cached value by the transform that represents a transform node, or the gfx:Transform matrix that represents a scroll offset, also taking into account perspective and transform origin, which are also representable as gfx::Transforms.

We also apply flattening top-down for nodes which have flattening. This may sound odd because conceptually, flattening is a bottom-up concept, but because we have restricted to the case when the ancestor state flattens, top-down and bottom up are equivalent. In other words, for any two linear transforms:

```
flatten(A * flatten(B)) == flatten(flatten(A) * B)
```

TODO(chrishtr): copy the written proof into this document.
The right-hand-side formulation in the above equation is a top-down representation.

Computing clip_to_ancestor_clip_rects_ and effect_to_ancestor_clip_rects_

As above, we compute these in a single top-down property tree recursion. Since the clip rect in the ancestor space is infinite up to the nearest enclosing clip, or effect node which applies a clip[1] , we need only map from the enclosing clip and effect state to the ancestor.

Child property tree states that modify the clip rect are easily computed by transforming the local clip into the ancestor space by using the `to_ancestor` value cached in `to_ancestor_transforms_`.

Putting it all together

Finally, it should be clear that the transform and clip top-down property tree recursions can easily be combined into one recursion, for maximum efficiency.

[1] Such as rounded rect clip, CSS clip or clip-path.