Web page geometries

Chris Harrelson
February 2016

Summary

What

This design doc proposes to centralize geometry code in Blink and cc (the Chromium compositor) into one common class called GeometryMapper, and use property trees as the data structure that drives it. Existing code will be migrated to use this class.

A side-effect of this design doc will also be a shared property tree representation between Blink and cc.

By geometry code we mean computing:

1. Visible rects of an object/rect in some containing space.
2. Transformed rects of an object/rect in some containing or contained space.

In the case of Blink, the "object" referred to above is a `blink::LayoutObject` or `blink::InlineBox`. In the case of cc, the "object" referred to above is a `cc::Layer` or `cc::LayerImpl`.

Visible rects are currently computed by `blink::LayoutObject::mapToVisibleRectInAncestorSpace` in Blink and `cc::CalculateVisibleRects (&` cc::ComputeClips) in cc.

Transformed rects are currently computed by `LayoutObject::localToAncestorQuad/FrameView::contentsToRootFrame` and `blink::LayoutObject::absoluteToLocalQuad/FrameView::rootFrameToContents` in Blink, and `cc::TransformTree::UpdateTransforms` (plus mapping, projection and flattening code in `math_util.cc` & `gfx::Transform`) in cc.

Why

This proposal aims to:

- Simplify many code paths which re-implement the same concepts under different names and code, into three simple methods and three simple concepts.
- Apply a common language to all geometry code across Blink and cc.
- Improve clarity of code by narrowing interfaces.
- Improve correctness of all code by having only one implementation.[1]
- Improve performance of all code, due to using a hyper-efficient geometry implementation, and better caching.

[1] For example, during the researching of this design doc, at least four significant bugs were found in the Blink implementation of visible and transformed rects.

Definition of paint property trees

Transform effect, scroll and clip hierarchies

Elements in a web page can have a **transform**:

```
<div style="transform: rotateZ(45deg)"> ... content … </div>
```

or various **effects**:

```
<div style="opacity: 0.6"> … content … </div>
```

clips:

```
<div style="overflow:hidden"> … content … </div>
```

or **scrolling**:

```
<div style="overflow:scroll"> … content … </div>
```

Transforms, effects, clips and scrolls are **paint properties**, and the hierarchy of each of them forms a **paint property tree**. Each web page element has a **property tree state** (actually, sometimes more than one, depending on what question is asked), meaning the four node in each of the hierarchies which applies to it (plus all ancestors of those nodes).

Four different hierarchies

The four paint property trees (transform, effect, clip, scroll) do not necessarily have the same topology as each other, and each has special semantics.

For example, s*crolling* and *clipping* are similar to transforms or effects, but have special behavior that makes them different. The important difference for the purposes of this design document is that scrolling and clipping only apply to descendants for whom the scroll or clip are on the containing block chain, whereas transforms and effects apply to all children.

Tree representation

The property trees will be represented by a vector of data nodes that refer to each other by index, copying the representation of `cc::PropertyTree`. The four types of data nodes in each of the four trees are as follows[2]:

```
struct TransformNodeData {
  gfx::Point3F transform_origin;
  gfx::Transform to_parent;
  // Specifies whether the content below should be flattened.
  bool should_flatten;
  // Specifies the 3d-sorting context in which this transform participates.
  int sorting_context_id;
};

struct EffectNodeData {
  float opacity;
  // Will also include filters, etc.

  // The transform and clip nodes that specify the local space of this
  // effect.
  int transform_id;
  int clip_id;
};

struct ClipNodeData {
  // The clip rect that this node contributes, expressed in the space of its
  // transform node.
  gfx::RectF clip;

  // The id of the transform node that defines the clip node's local space.
  int transform_id;
};

struct ScrollNodeData {
  // The id of the transform node that defines the scroll node's local space.
  int transform_id;
  gfx::Size bounds;
  gfx::Size offset;
```

[2] Blink and cc may subclass these classes, in order to add fields required for use-cases specific to their codebase. See property_tree.h for many examples of possible fields a subclass might contain. However, subclassing should not be preferred, instead we should look deeply for whether all users require the same concept.

```
};

template <typename T>
struct TreeNode {
  TreeNode() : id(-1), parent_id(-1), owner_id(-1), data() {}
  int id;
  int parent_id;
  int owner_id;
  T data;

  bool operator==(const TreeNode<T>& other) const;

  void ToProtobuf(proto::TreeNode* proto) const;
  void FromProtobuf(const proto::TreeNode& proto);
};

template <typename T>
class CC_EXPORT PropertyTree {
  vector<TreeNode<T> > nodes_;
}
```

We also define a struct combining nodes for all four trees that apply to some object:

```
struct PropertyTreeState {
  int transform_id;
  int effect_id;
  int clip_id;
  int scroll_id;
}
```

The wrapper data structure is then:

```
class PropertyTrees {
  vector<PropertyTree<TransformNodeData> > transform_tree;
  vector<PropertyTree<EffectNodeData> > effect_tree;
  vector<PropertyTree<ClipNodeData> > clip_tree;
  vector<PropertyTree<ScrollNodeData> > scroll_tree;
}
```

Geometry interface

The new, central geometry code will implement the following interface.

```
class GeometryMapper {
   GeometryMapper(const PropertyTrees& trees);

  enum ResultCachingBehavior {
    DoNotCacheIntermediateResults,
    // Specifying CacheIntermediateResults means that extra data should be
    // cached, to ensure that performance of a sequence of calls to the
    // methods below is guaranteed to run in O(n + m) time, where n is the
    // number of nodes in all property trees, and m is the number of calls
    // to the methods below.
    CacheIntermediateResults
  };

  // Returns the flattened axis-aligned rect in |ancestor_state|
  // that is equal to the visible area in that space into which |rect| may
  // draw. |rect| starts out in |local_state|. |ancestor_state| must
  // not be a descendant of |local_state| in any of the four property
  // trees. ancestor_state must be a state that flattens.
  gfx::Rect LocalToVisibleRectInAncestorSpace(const gfx::Rect& rect,
      const PropertyTreeState& local_state,
      const PropertyTreeState& ancestor_state,
      ResultCachingBehavior caching_behavior =
          DoNotCacheIntermediateResults);

  // Returns the flattened, axis-aligned rect in |ancestor_state|
  // which is the result of applying the intermediate transforms between it
  // and |local_state|. |ancestor_state| must not be a descendant of
  // |local_state| in any of the four property trees.
  // ancestor_state must be a state that flattens.
  gfx::Rect LocalToAncestorRect(const gfx::Rect&,
      const PropertyTreeState& local_state,
      const PropertyTreeState& ancestor_state);

  // Returns the flattened, axis-aligned rect in |local_state|
  // which is the result of applying the inverse of the transform between
  // it and |ancestor_state|. If non-invertible transforms were encountered,
  // sets |success| to false and returns an empty gfx::Rect; otherwise sets
  // |success| to true. |ancestor_state| must not be a descendant of
  // |local_state| in any of the four property trees.
  // ancestor_state must be a state that flattens.
  gfx::Rect AncestorToLocalRect(const gfx::Rect&,
      const PropertyTreeState& local_state,
```

```
      const PropertyTreeState& ancestor_state, bool* success);

  // 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);
}
```

Implementation

Building precomputed data

```
class GeometryMapperImpl : public GeometryMapper {
  // Maps from the (ancestor, local) pair to cached state.

  // TODO(chrishtr): can we avoid a hash map? There may be a more
  // memory-efficient caching data structure.
  typedef hash_map<pair<PropertyTreeState, PropertyTreeState>,
      CachedPropertyTreeData> PrecomputedData;

  PrecomputedData cached_state_;

  // ...
  // See below for how the methods themselves are implemented.
}

struct PrecomputedData {
  hash_map<PropertyTreeState, PrecomputedDataForAncestor> data_;
  gfx::RectF ClipInAncestorSpace(PropertyTreeState descendant_state);

  gfx::Transform TransformToAncestorSpace();

}
```

This document describes in detail how to compute PrecomputedData.

LocalToVisibleRectInAncestorSpace

`LocalToVisibleRectInAncestorSpace` is implemented as follows:

1. Build `PrecomputedData` between ancestor and local if necessary.

2. Map `rect` to the ancestor space using `to_ancestor`, then clip it using `clip_in_ancestor_space`.

`LocalToAncestorRect` is similar to `PrecomputedData`, but just multiplies by `to_ancestor`.

`AncestorToLocalRect` is the inverse of `LocalToAncestorRect`, multiplying by the inverse of `to_ancestor`. If `to_ancestor` is not invertible, it returns the empty rect and sets `success` to false, otherwise sets `success` to true.

Detailed examination of use cases

Here we enumerate all use cases that will be replaced with calls into the above `GeometryMapper` interface.

Computing the render surface layer list from the effect tree

This document describes how to compute render surfaces from the effect tree. However, it assumes that (a) all render surface triggers are in the effect tree, and (b) the render surface hierarchy is a sparsification of the effect tree (meaning it has the same topology among the effects which trigger render surfaces). (b) is required because the algorithm assumes that a walk of the PaintChunks in draw order induces a correctly nested tree walk of the corresponding effect tree nodes, without repetition.

Enforcing property (b) requires extra work to create the right effect tree. To achieve this, we

- Add some effect nodes to the “unrolled” effect tree (and sometimes the “compact” one) to represent all of the render surface triggers (see here for the list, and detailed analysis)
- expand the “compact” effect tree as necessary to represent the hierarchical relationship of content to rounded rect overflow clips in the DOM. This will be done in exactly the same way as is done already for Blink clip nodes representing overflow clip.
- “unroll” the Blink (“compact” effect tree during the SPv2 Blink compositing step to generate the cc (“unrolled”) effect tree. This document gives a detailed explanation of how unrolling works.

The pipeline will look like this:

1. Create compact property trees
2. Paint (creates PaintChunks and associations with the compact property tree nodes)
3. Unroll the effect property tree
4. Walk the unrolled effect property tree to generate the RenderSurfaceLayerList

Determining visible rects in cc

Visible rects in cc are used to determine which layer tiles are on or near the screen, and therefore need to be rastered. See `PictureLayerTiling::ComputeTilePriorityRects`, which gets its visible layer rect from `PictureLayerImpl::UpdateViewportRectForTilePriorityInContentSpace`.

They are also used to optimize which `PictureDrawQuads` are necessary to include in the drawn output to the screen. See `PictureLayerTilingSet::CoverageIterator`, which takes in a `visible_quad_layer_rect` for pruning, that is in turn set in `LayerImpl::PopulateSharedQuadState` to the `visible_layer_rect`.

The visible rects for a cc layer are computed in target `RenderSurface` state. The target may not be a clip ancestor of the layer, so using the methods of `GeometryMapper` does not work naively, since `GeometryMapper` only knows how to map visible rects up the property tree hierarchy, not down.

In such cases, we need to compute clip rects in the least common clip node ancestor of the target property tree state and local property tree state, then use the inverse transform to map down with `AncestorToLocalRect`. (This approach is what `cc::CalculateVisibleRects` already does, see that method for details.)

Draw-time clipping in cc

Draw-time clipping in cc currently uses code that is entangled with visible rect computation. This section of that document gives pseudocode for how the `draw_content_rect` property of each layer are set. The `draw_content_rect` property is the clip that must be applied, one way or another, to a layer when drawing it to the screen.

Steps 2(a)(i) and 2(a)(ii) of that algorithm can be implemented efficiently using GeometryMapper, as follows:

```
GeometryMapper geometry_mapper(property_trees);
…
Step 2(a)(i):
combined_clip_rect =
geometry_mapper.LocalToVisibleRectInAncestorSpace(property_trees.clip_tree.No
deAt(layer_local_space.clip_node).clip, layer_local_space,
render_surface_clip_node)
..
Step 2(a)(ii):
geometry_mapper.AncestorToLocalRect(combined_clip_rect,
render_surface_clip_node, render_surface space)
```

Paint interest rect computation in Blink

Blink paints all content that is within 4000px of the screen. The interest rect for a composited layer is the visible layer rect for it, expanded by 4000px in all directions, and clipped to the layer size. This can be implemented as follows with `GeometryMapper`:

```
GeometryMapper geometry_mapper(property_trees)
visible_content_rect =
      geometry_mapper.LocalToVisibleRectInAncestorSpace(layer bounds, layer
property tree state, root)
geometry_mapper.AncestorToLocalRect(visible_content_rect, layer property tree
state, root);
```

Element.getClientRects

This web API returns the bounds of the element (multiple if the element is inline), relative to the containing frame's root element. In terms of `GeometryMapper` this is straightforward:

```
Vector<rect> local_bounds = element bounds, or vector
For (rect : local_bounds) {
      rect.moveBy(element offset from its PropertyTreeState)
      geometry_mapper.localToAncestorRect(rect, element PropertyTreeState,
root PropertyTreeState)
      out.push(rect)
```

Element.getBoundingClientRect

This web API returns the bounding rectangle that contains the rectangles in Element.getClientRects. Hence it's just the union of the rects as computed in the previous section.

Computing the containing frame viewport position of an element for a context menu, autofill, link highlight, find-in-page, or similar page overlay

Similar to `blink::Element.getClientRects`.

Computing the screen-space visible rect of an element for testing overlap

The Blink compositing code tests whether elements painted later overlap ones painted earlier that are composited. The input to the test is the visible rect in screen space of the element bounds being tested.

Element.scrollIntoView: scrolls an element into view

To do so requires first computing the transformed bounds of the element in the space of its containing scroller or frame.

IntersectionObserver

`IntersectionObserver` requires computing the visible rect for an element in the space of an ancestor, and efficiently updating that computation from frame to frame. It also requires mapping observation rects to screen space.

See blink::`IntersectionObservation::clipToRoot` and blink::`IntersectionObservation::computeGeometry`.

Plugin power saver

Part of this feature computes the bounds and visible portions of a plugin within the web page (which is the same as the visible rect).

See blink::`WebPluginContainerImpl::computeClipRectsForPlugin`

Touch adjustment and tap disambiguation

When processing touch events, screen-space rects are computed for candidate touch target elements, to determine which one is the best (this is “touch adjustment”). There is a second use case of determining a candidate set of touch targets, which result in a disambiguation overlay dialog (this is “tap disambiguation”).

See blink::`TouchAdjustment.cpp`.

Computing hit test rectangles for elements in the space of their containing composited layer

See blink::`projectRectsToGraphicsLayerSpaceRecursive`. This code is used to determine which areas of a composited layer need event processing in the main thread, because of the presence of blocking event handlers.

Offscreen iframe and animated image throttling

See blink::`FrameView::updateViewportIntersectionIfNeeded` and blink::`LayoutBox::intersectsVisibleViewport`.

Computing clip rects when painting fragments

The desired clip rect is the visible rect for an element (or fragment of an element, in the presence of pagination and multi-column), including clips up to a given ancestor (the painting root for a composited layer), and either in the space of the ancestor or in the local space.

See blink::`PaintLayerClipper::calculateRects`, `PaintLayerClipper::backgroundClipRect` and blink::`PaintLayerClipper::localClipRect`.

These rects are use to clip descendants when painting, and to update the cull rect for avoiding recursion into clipped-out descendants.

Also, the call sites to this method require ignoring overflow clip in some cases. We could potentially support that by moving CSS clip from the clip tree to the effect tree, and ignoring the clip tree. (Moving CSS clip from the one tree to the other has other motivations as well.)

Determining clip rects and sizes for composited clipping layers

See `blink::CompositedLayerMapping::updateAncestorClippingLayerGeometry`, `blink::CompositedLayerMapping::owningLayerClippedByLayerNotAboveCompositedAncestor` and `blink::CompositedLayerMapping::localClipRectForSquashedLayer`. These call sites will go away with Slimming Paint v2, but in the meantime can can also be satisfied by use of GeometryMapper::LocalToVisibleRectInAncestorSpace and seeing if anything is clipped, or what the clip is.

PaintLayer hit testing

These use cases are the same as for painting fragments, since the hit testing code mirrors paint.

Hit testing during event handling

Hit testing needs to adjust for scrolling, and map between coordinate spaces of frames. See the `EventHandler` class. cc has an equivalent use case, see `cc::LayerImpl::TryScroll`.

Some implementation details

Storing offset and size of objects in PropertyTreeState

The use cases above all rely on being able to, in O(1) time, compute the size of any object and the offset of it from the origin of its containing property tree space.

Storing this information on `cc::Layer` should be simple to achieve. Storing this information on `LayoutObject` is already planned in order to implement the upcoming rewrite of paint invalidation for Slimming Paint phase 2.

Performance optimizations

Caching GeometryMapper and property trees

In cc, Property trees and the equivalent of the `GeometryMapper` cache are already cached between animation frames if no state changes from one frame to the next.

`PaintLayerClipper` in some cases also stores similar cached information for painting, hit testing and compositing use cases (hit testing is probably the most important for performance).

We can achieve the same, and even more, benefits in cc and Blink by storing a `GeometryMapper` and the property trees on the `WebView` in Blink. In this way, many or all of the above use cases can get the same performance benefit of avoiding potentially expensive tree walks.