Unrolling the effect tree, adding render surface triggers

As described here, it is required that the tree of render surfaces in the compositor is a sparsification of the compositor's effect tree.

The Blink effect tree represents all effects necessary to draw the content to the screen, but is represented in a compact, "rolled" form that suffices for representing the hierarchical relationships between the effects, but does not necessarily map 1:1 to the sequence of tasks necessary to draw[1] the painted content to the screen. In particular, paint order of content on the web page may require interleaving content with different effect tree states, due to the complicated interaction of effects with stacking contexts and containing block chains.

We need the render surface "tree" to be a sparse representation of the effect tree from which it is generated. Another way of saying "sparse representation" is that:

A. each render surface can be associated 1:1 with an effect node[2]
B. and the parent-child relationship between the render surfaces is the same as the parent-child relationship between their triggering effect nodes.

To satisfy both A and B, we need to expand the compact effect tree, and insert additional effect nodes in a few cases. These two needs are covered in detail in the following two sections.

Unrolling the effect tree

Since effects generally must be applied atomically to contained content, this means that we may need more render surfaces than there are nodes in the *compact* form. In particular, we need to add additional copies of some effect nodes when interleaving occurs. We call this process *unrolling* the effect tree. The resulting effect tree is called the *unrolled effect tree*. Mapping from unrolled effect tree to the RenderSurfaceLayerList in draw order is described here.

[1] *Painting* is turning a tree of LayoutObjects into a sequential display list with references to property tree nodes. *Compositing* is chopping up the display list into pieces. *Drawing* is the sequence of GPU commands to put composited layers up on the screen, taking into account the (hierarchical) requirements of the property trees. *Render surfaces* are the intermediate bitmaps used, if necessary when drawing.

[2] We say an effect node "has" a render surface when the node is the trigger for that render surface.

The algorithm below duplicates subtrees of the effect tree in when paint order requires interleaved drawing of the same effect node twice (said another way: it duplicates a subtree when a violation of the requirement of a *correctly nested tree walk of the effect tree, without repetition*, is found).

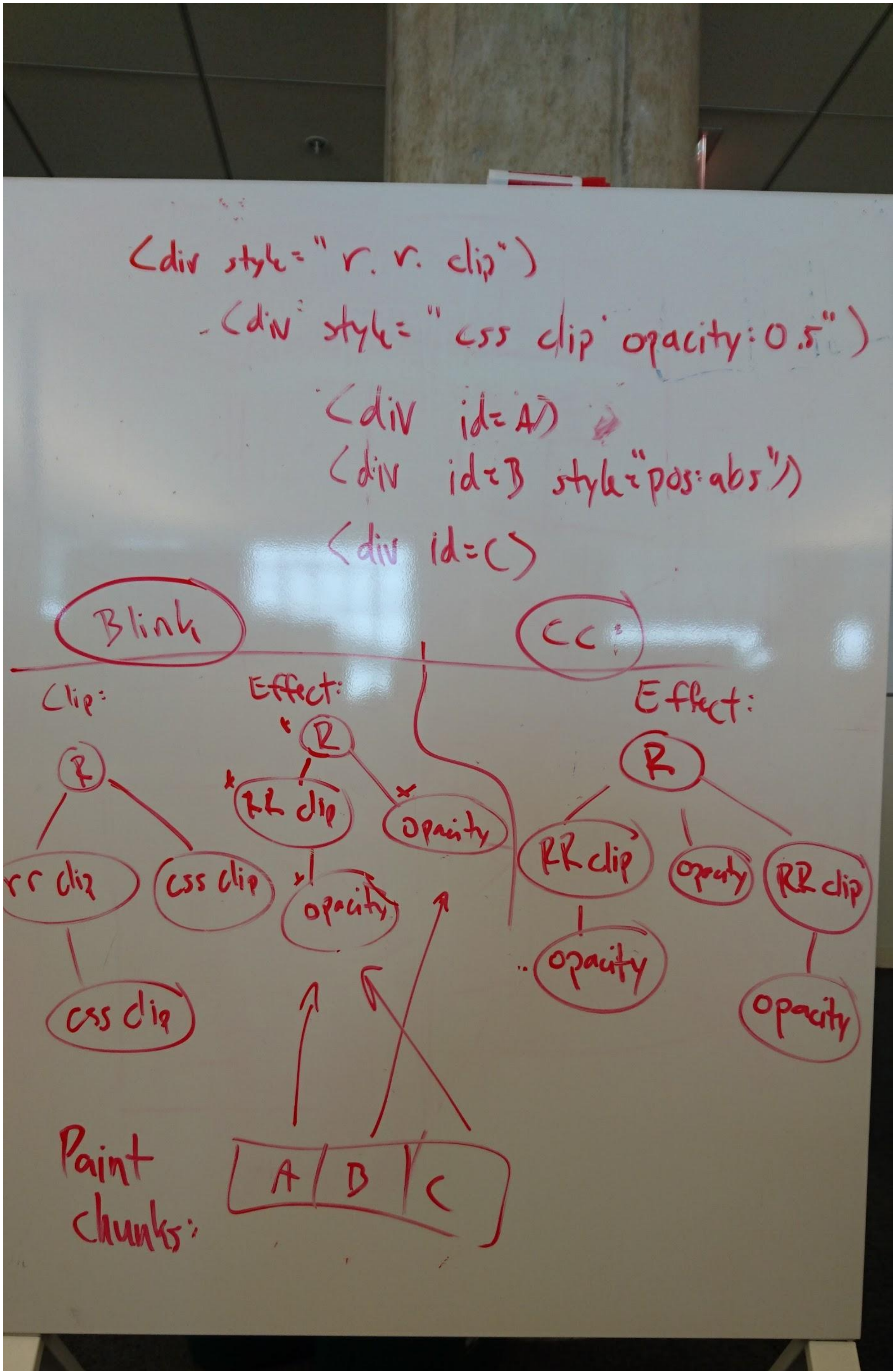
Unroll algorithm

```
// Inputs:
// EffectTree rolledEffectTree: this is the compact effect tree
representation from Blink
// vector<PaintChunk> paintChunks: the sequence of paint chunks. Each paint
chunk has a
// different PropertyTreeState than the one preceding it.
algorithm UnrollEffectTree(rolledEffectTree, paintChunks):

int currentPostOrderIndex = -1
hash_map<EffectNode, actualEffectNode> actualEffectNodes;
EffectNode previousEffectNode = nullptr
for (paintChunk : paintChunks):
  EffectNode effectNode = actualEffectNode(paintChunk.effectNode(),
      actualEffectNodes)
  if (previousEffectNode) {
    EffectNode leastCommonAncestor =
      rolledEffectTree.leastCommonAncestor(effectNode, previousEffectNode)
      Mark all nodes between leastCommonAncestor and previousEffectNode in
          rolledEffectTree as “done”
      if first child on path from leastCommonAncestor to effectNode is
          “done”:
        Copy the path from leastCommonAncestor effectNode into a new path
            which is rooted at leastCommonAncestor
        For each newNode on this new path
            actualEffectNodes[originalNode(newNode)] = newNode
            (where originalNode is the effect node newNode came from back at
             the start of the algorithm)
  previousEffectNode = effectNode


function actualEffectNode(effectNode, actualEffectNodes):
  if (actualEffectNodes.contains(effectNode)
    return actualEffectNodes[effectNode]
  else
    return effectNode
```

An example input and output is pictured here:



```
<div style="r. r. clip")
   <div style="css clip opacity:0.5")
        <div id=A)
        <div id=B style="pos:abs")
        <div id=C)
```

Adding in effect nodes for all render surface triggers

The current triggers for render surfaces (as of February 19, 2016) are encoded in cc::ShouldCreateRenderSurface. Note that they are currently defined in terms of cc::Layers, not effect nodes. (We will need to redefine these in terms of the effect tree.) They are:

1. Layer has a mask (usually for CSS border-radius, but also CSS clip-path and some other cases)
2. Layer has a replica layer (we can ignore this one, since SPv2 obsoletes it)
3. Layer has CSS filter
4. Has animation which might add a CSS filter
5. Layer is in a 3D rendering context and should flatten the transform from the ancestor[3] (i.e. parent is in a 3D rendering context, but elements inside it in the DOM are flattened into it).
6. Layer has blending
7. Layer clips children and is not 2D axis-aligned with respect to its parent ("clips children" means having a mask, or several other cases all of which don't seem to apply to SPv2)
8. Layer has opacity, flattens from the ancestor, and has at least two children which draw content.
9. Layer is the root for a blend mode "isolation group". See cc::Layer::SetIsRootForIsolatedGroup.
10. We decided to force a render surface on this layer (this is only used for testing).
11. The layer has a "copy request" (think: pixel-readback is going to be requested for the layer contents).

Render surface triggers 1, 3, 6, 7, and 8 already have natural effect tree nodes associated with them. Render surface 2 is irrelevant.

This leaves triggers 4, 5, 9, 10 and 11 without as yet any effect tree node representing them, yet one is required. Here is what we should do for each:

- Condition 4: force Blink to add a placeholder effect tree node (in general, it is a requirement for such animations to modify property tree nodes, not add or remove them)
- Condition 5: in the language of SPv2, a PaintChunk takes the place of a cc::Layer. Let's call the PaintChunks whose transform property tree node is equal to or a descendant of the condition-5 transform node its *transform-descendant PaintChunks*.
  It can be shown that:

[3] cc defines flattening top-down instead of bottom-up. This is actually equivalent to the natural web definition. I have a proof but it doesn't fit in this footnote...

    a. All of the transform-descendant paint chunks of a condition-5 paint chunk paint contiguously[4];
    b. all of the transform-descendant paint chunks also have effect nodes equal to or descendants of the condition-5 effect node[5]; and
    c. any of the effect nodes referred to in (b) are used only by transform-descendant paint chunks of the condition-5 paint chunk (this follows pretty directly from (b).

  As a result of the above, we can safely insert an effect node for the condition-5 PaintChunk and reparent the effect nodes from (b) under it.

  This can be done when converting from the compact effect tree to the unrolled effect tree. Alternatively, we could represent it directly in the compact effect tree without undue harm, except perhaps a small layering violation.
- Condition 9: we will create a no-op effect node for stacking context that has one or more exotic blending child stacking contexts.
- Condition 10: who cares, testing.
- Condition 11: copy requests are added for web content only on the root layer (see RenderWidgetCompositor::UpdateLayerTreeHost), which already has an effect node. The UI compositor also appears to apply copy requests to the root layer. Thus we can enforce that a copy request is only valid for the root effect node.

[4] This is a consequence of transforms forming a stacking context and being a containing block for all descendants (even position:fixed).

[5] This is because all effects which apply to the transform-5 node apply to all descendants, since it's a containing block