# RSLLs From Effect Trees Proposal

## Problem

How to shape the Blink<->cc API such that the compositor thread can determine which render surfaces need to exist, which layers draw into which render surfaces, and what order to draw render surfaces in.

Render surface determination needs to be done on the compositor thread. Some surfaces are temporary (copy requests). Other surfaces are conditional (opacity animation might need a surface for opacity<1 but not for opacity=1). It's also desirable not to expose render surface knowledge into Blink for abstraction reasons. Therefore, enough information needs to be passed along to cc to generate render surfaces.

## Proposed tl;dr solution

Blink hands cc via WebLayerTreeView/LayerTreeHost:

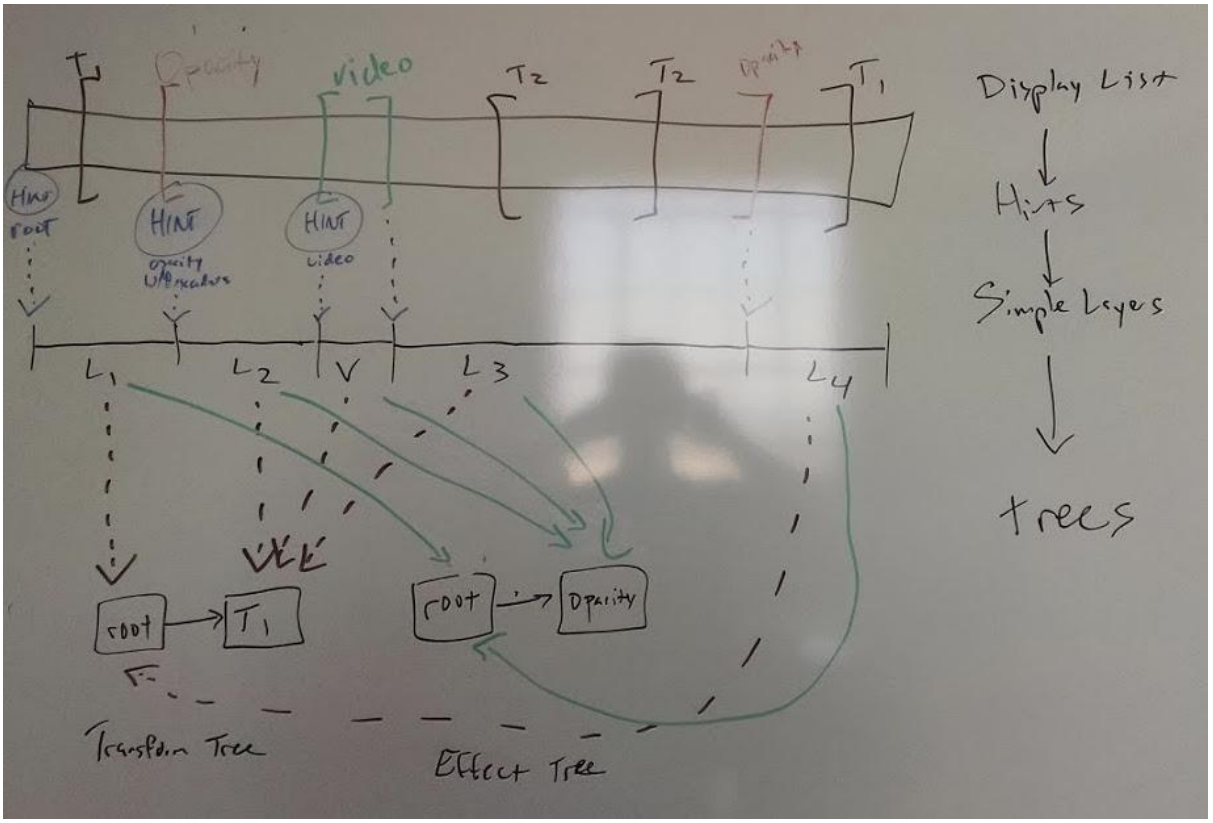
- A global display list
- Property trees
    - Transform tree
    - Clip tree
    - Effect tree
    - Scroll tree
- An ordered (in paint order) list of simple layers, each of which has:
    - indices into the subset of the global display list that it represents
    - index into each property tree
    - 2d offset from transform tree

cc will then take the effect tree and the ordered list of simple layers and the effect tree and will use that to generate the render surface layer list on each compositor frame.

Possible transition plan:

1. move RSLL generation out of CDP
2. generate effect trees on the main thread as part of building property trees
3. generate RSLL equivalent from the effect trees and ordered layer list
4. have Blink provide effect trees and layer list directly

## Example with pictures but also too many words <_<




An example display list with two transforms (T1, T2), one effect (opacity), and a video. It has three hints (root, opacity, video). This divides up into 5 simple layers (L1, L2, V, L3, L4). There's a transform tree with two nodes (root, T1) and an opacity tree with two nodes (root, opacity). Consider T1 to be a scale. It's not something that would create a layer like a 3d transform, but is something that simple layers below it would need to know about for scale purposes.

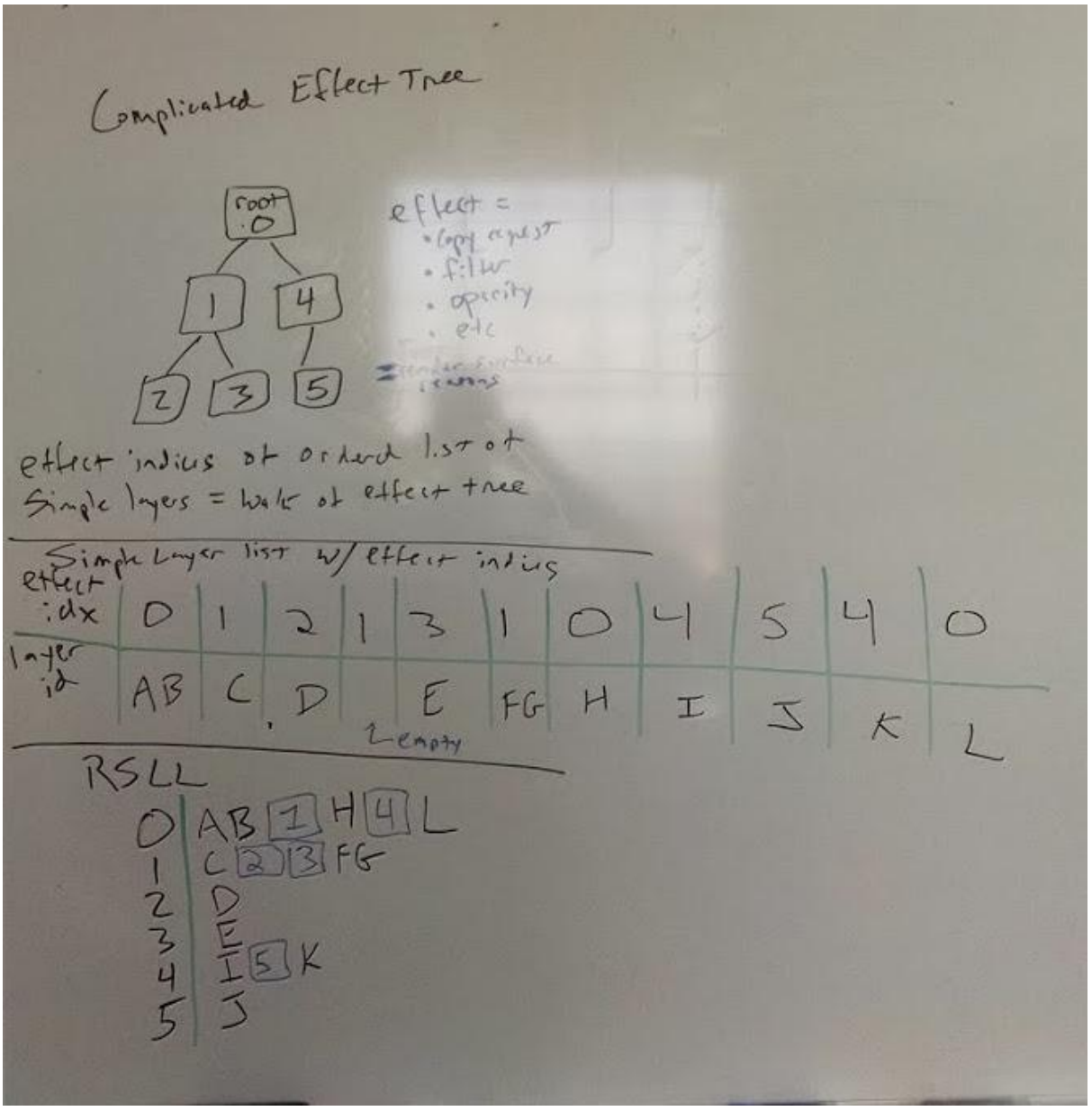
For transform indices, each simple layer points to either root or T1. T2 doesn't require a transform node, because there are no simple layers that need to refer to it. For effect indices, each simple layer either points to the root or to the opacity node, depending on where they are in the tree. Clip indices are omitted for simplicity.

It seems roughly possible to generate property trees from a display list with hints via an O(n) walk. The in-order walk needs to maintain a stack of transforms and clips that are used in painting. When encountering a hint boundary like opacity, a new transform node needs to be added (to make simple layers aware of T1) and a new opacity node also needs to be created (to make them aware of the opacity). The video is also a hint, but because it doesn't have a different transform/clip/effect, it doesn't require any new nodes to be created in the property trees to represent it. (It's possible that you'd need to add multiple transform nodes if there are intervening animations or scrollers, etc.)

Nodes in the effect tree can be considered to be potential render surfaces. All render surfaces are represented by a node in the tree, but not all nodes have to create render

surfaces (e.g. animated opacity effect, but opacity=1)  Another way to think about nodes in the effect tree is that they all represent a reason to create a render surface, which is reflected in this function: https://code.google.com/p/chromium/codesearch#chromium/src/cc/trees/layer_tree_host_common.cc&l=547

Here's a more complicated effect tree, with a hierarchy of effects:




Each simple layer will have an index into the effect tree (0-5 in this example).  The layers here are represented by letters, where alphabetical order is also paint order.  Multiple simple layers can use the same effect index, e.g. A and B both use the root effect 0.

One can see that the list of effect indices from the simple layers is an in-order walk of the effect tree (potentially revisiting parent nodes before/between/after each child).

Assume in this example that every node in the effect tree creates a render surface. There are 6 render surfaces (0-5). For example, render surface 4 is drawn by drawing layer I, drawing render surface 5's texture, and then drawing layer K after that. In order that render surface dependencies are met (e.g. 1 depends on 2 and 3, so they both need to be generated first), render surfaces are generated in reverse order.

This RSLL can be easily generated by walking the ordered list of layers. When encountering a new effect id that creates a layer, generate a new render surface and insert the render surface into the previous effect's list (e.g. encounter layer C with effect 1, so generate render surface 1 for C, and insert [1] into render surface 0.) If encountering a layer with a previous effect, append that layer to the layer list for that effect.

## Caveat: RSLL as-is cannot be used for slimming paint

It's clear that today's RSLL will not be sufficient for slimming paint v2, as it's currently possible to have a LayerImpl that creates a render surface but does not draw content. In slimming paint v2, the ordered list of layers will only be layers that draw content. Therefore, we need some alternative data structure to represent the RSLL.

One possibility is to create an interface called QuadGenerator that contains the AppendQuads function. LayerImpl and RenderSurface would both derive from this. Then LayerTreeImpl would own an ordered set of layers, own an ordered set of render surfaces, and each render surface would have weak pointers to an ordered list of quad generators. (Alternatively, maybe RenderSurface should derive from LayerImpl?)

## Caveat: effect nodes need clip node indices

Each effect node needs a pointer to the clip node that it represents. This is required so that child layers drawing into any render surface for that effect node can be put into the right space and clipped correctly.