Layerization based on display lists

ajuma@, vollick@, October 2014

This document describes how a display list sent from Blink to the Chromium Compositor (cc) will be converted by cc into a set of layers, and how cc will update this layerization when the display list changes. The construction of display lists in Blink is described in a separate document.

Overview

The partitioning of content into compositor layers currently happens in Blink (most of the logic is found in `Source/core/rendering/compositing`). Blink constructs compositor layers that consist of one or more `RenderLayers`. As part of the Slimming Paint project, the creation of compositor layers is being de-coupled from the concept of `RenderLayers`, and the responsibility for partitioning content into layers is moving to cc.

Blink will provide cc with a display list containing all content, in paint order. Blink will also provide separate *effect lists* containing transforms, clips, filters, and compositing hints. The items in these effect lists will refer to the positions in the display list of the content that they affect. cc will use the information in this lists to decide how to partition the display list into layers. When the display list or the effect lists change, Blink will provide the new lists together with additional information that allows cc to efficiently update and invalidate its layers.

Input format

The inputs provided by Blink are a display list, a list of compositing hints, lists of transforms, clips, and filters, and information about the differences between the current display list and the previous one.

The display list consists of a vector of `ContentDisplayItems`, in paint order.

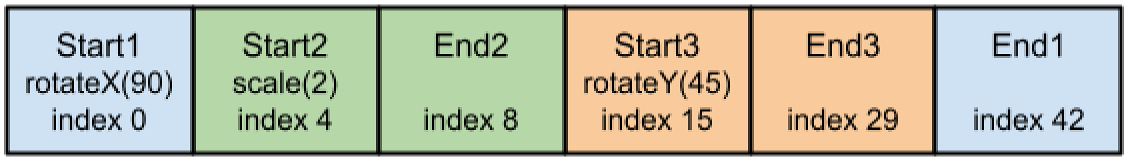
```
struct DisplayItem {
  int id;  // unique and stable identifier
  int source_id;  // stable identifier of source RenderObject
};

struct ContentDisplayItem : public DisplayItem {
  SkPicture* content;

  // These are wrt this item’s transform parent, which can be determined
  // using the TransformList.
  gfx::SizeF bounds;
  gfx::Vector2dF offset;
};
```

```
typedef std::vector<ContentDisplayItem> ContentDisplayList;
```

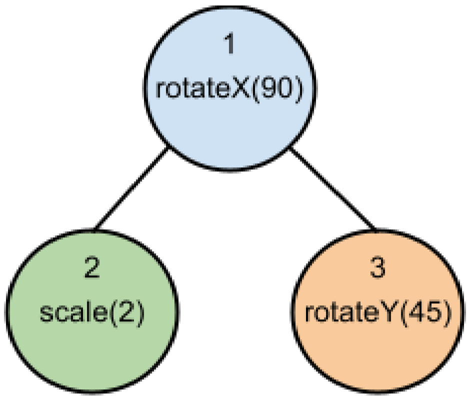
Additional information about the display list is provided in effect lists: the compositing hint list, and the transform, clip, and filter lists. Each effect is represented by two list items, which contain the start and end indices of the contiguous sequence of display list items that are affected.

```
struct EffectDisplayItem : public DisplayItem {
  enum EffectItemType {start, end};
  EffectItemType item_type;

  int matching_id;  // id of the corresponding start/end

  // Position in the ContentDisplayList corresponding to this effect.
  int display_list_index;
};

struct HintDisplayItem : public EffectDisplayItem {
  enum HintType {root, transform3D, animation, video, canvas,
                 will_change, filter, scroll};
  HintType hint;
};

struct TransformDisplayItem : public EffectDisplayItem {
  gfx::Transform value;
};

struct ClipDisplayItem : public EffectDisplayItem {
  gfx::Rect clip_rect;
}

struct FilterDisplayItem : public EffectDisplayItem {
  FilterOperations value;
}

// These lists are sorted by display_list_index.
typedef std::vector<HintDisplayItem> HintList;
typedef std::vector<TransformDisplayItem> TransformList;
typedef std::vector<ClipDisplayItem> ClipList;
typedef std::vector<FilterDisplayItem> FilterList;
```

Start and end items in an effect list are always properly nested. This means each effect list can be interpreted as a flattened representation of an effect tree (where every matched pair of start and end items in the list corresponds to a tree node, and where nested items correspond to tree descendants). For example, consider the following transform list:



This represents the following transform tree:



The items at positions [0,...,3], positions [9,...,14], and positions [30,...,42] in the display list have node 1 as their transform parent. Similarly, the items at positions [4,...,8] have node 2 as their transform parent, and the items at positions [15,...,29] have node 3 as their transform parent.

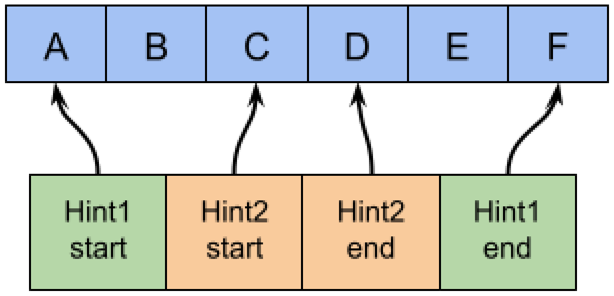
To allow cc to update layers and compute invalidation regions efficiently, Blink also provides information about changes to the display list. This consists of two lists: a list of indices of items in the previous list that have been deleted, and a list of items in the current list that are newly-added. Insertions and deletions are the only changes allowed; this means that any modification of a previously existing item (including moving the item to another position in the list) is treated as a deletion followed by an insertion.

```
// A list of indices of ContentDisplayItems that have been deleted
// from the previous ContentDisplayList.
typedef std::vector<int> DeletedItems;

// A list of indices of ContentDisplayItems that are new in the
// current ContentDisplayList.
typedef std::vector<int> InsertedItems;
```

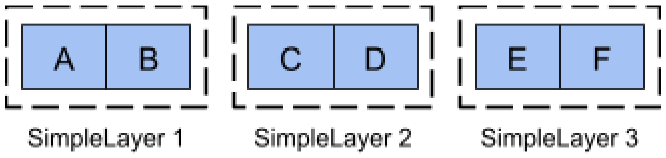
Output format

The output consists of a list of `Layers`, in paint order. Each `Layer` consists of one or more `SimpleLayers`, in paint order. A `SimpleLayer` represents a single contiguous sequence of items from the display list. This will be a sequence delimited on both sides by compositing hints, but with no contained compositing hints. This means that each `SimpleLayer` consists only of items that are contiguous in paint order and have no intrinsic need to be composited separately from each other. Then, rather than having to consider each display list item separately when constructing `Layers`, we only need to work at the `SimpleLayer` level; this is a significant advantage, as we expect there to be far fewer `SimpleLayers` than display list items (since we expect the number of compositing hints to be far fewer than the number of display list items).

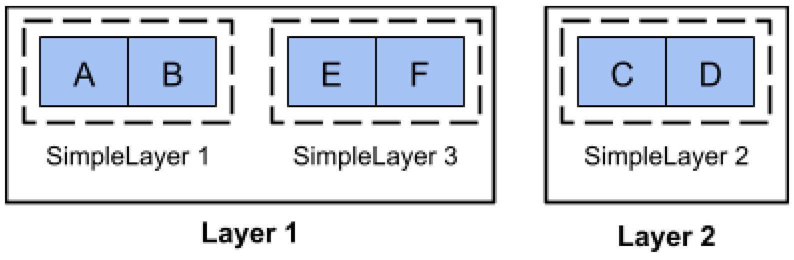
For example, consider the following display list and hint list, where arrows are used to represent the `display_list_index` of each hint item:



This produces the following `SimpleLayers`:



Assuming that `SimpleLayer 3` and `SimpleLayer 2` do not overlap (we would use the transform list and the bounds stored at each item to determine this), we get the following `Layers`:



The layerization algorithm is described in greater detail in the next section.

Each `Layer` also includes an invalidation region; this is the portion of the `Layer` that needs to be re-rastered because of changes in layerization or content. This region is expressed in the space of the Layer's transform parent, which is defined to be the same as the transform parent of its first SimpleLayer.

```
class SimpleLayer {
  // The global display list.
  const ContentDisplayList& display_list;
  // Indices defining this SimpleLayer's portion of the display list.
  int start_index;
  int end_index;

  // id of the start or end hint immediately preceding this
  // layer's display items. This value uniquely identifies
  // this layer.
  int hint_id;

  // id of the hint that immediately follows this layer's items.
  // Used for detecting changes in layerization.
  int closing_hint_id;

  // source id of the nearest start hint preceding this layer's
  // display items that is matched by an end hint that appears after
  // this layer's display items.
  int source_id_of_nearest_open_hint;

  // The lowest-common ancestor of all contained items.
  int transform_parent_index;
  int clip_parent_index;
  int filter_parent_index;

  // In the space defined by this SimpleLayer's transform parent.
  gfx::SizeF bounds;
  gfx::Vector2dF offset;

  // Layer this SimpleLayer paints into.
  Layer* layer;
  gfx::Transform tranform_to_layer;

  void SetDisplayItems(ContentDisplayList display_list,
                       int start, int end);
};

class Layer {
  vector<SimpleLayer*> layer_list;
  // These are in the space of this Layer’s transform parent.
  Region invalid_region;
  gfx::SizeF bounds;
  gfx::Vector2dF offset;

  // Matches the first contained SimpleLayer’s
  // source_id_of_nearest_open_hint.
  int hint_source_id;

  // These match the values in the first contained SimpleLayer. The
  // merging algorithm ensures that all other contained SimpleLayers
  // have these nodes as ancestors in the transform, clip, and filter
  // trees.
  int transform_parent_index;
  int clip_parent_index;
  int filter_parent_index;

  void AddSimpleLayer(SimpleLayer* simple_layer);
  void RemoveSimpleLayer(SimpleLayer* simple_layer);
};
```

Layerization algorithm

The layerization algorithm consists of several parts:

1. Identify changes to the transform, filter, and clip trees.
2. Identify the `SimpleLayers` we need.
3. Determine how the `SimpleLayers` we already have correspond to the ones we need.
4. Create `Layers` consisting of one or more `SimpleLayers` that can be combined.
5. Determine how the `Layers` we already have correspond to the ones we need.
6. Determine the invalidation regions for each `Layer`.

We describe these as distinct parts for the sake of simplicity, but some of these parts will actually be implemented together (e.g. parts 2 and 3 will be implemented together, as will parts 4 and 5).

Part 1: Identify changes to the transform, filter, and clip trees

We are given new effect lists (for transforms, filters, clips, and compositing hints) as input, and we also have the corresponding lists which we’ve retained from the previous frame. Since these lists are expected to be much shorter than the display list, we can efficiently identify additions and deletions without necessarily requiring Blink to send a list of changes (since taking time proportional to the lengths of these lists to identify changes is acceptable). For example, we can walk both lists, and when we encounter a difference, determine whether this is an insertion, a deletion, or a move by checking whether the item is present in the other list (we can efficiently determine presence in the other list by maintaining an item-to-index hash map).

After identifying changes, we update our transform, filter, and clip trees accordingly. These trees represent the same information as the effect lists, but in a format that makes it easier to determine the combined result of nested effects; for example, the transform tree makes it possible to efficiently determine a display item's screen-space transform, by walking from the item's transform parent to the root of the transform tree, multiplying the transforms found at each node.

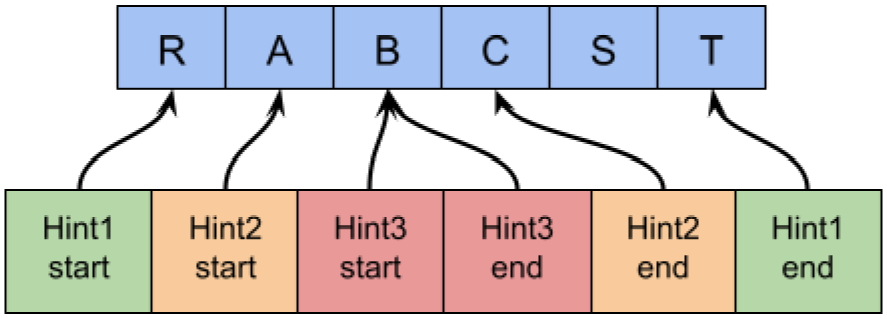
We also retain a copy of the previous set of trees since these will be needed later when computing invalidations. Specifically, we need these trees to compute the bounds (in `Layer`-space) of deleted items, and to compute the old bounds (in `Layer`-space) corresponding to items whose transform, filter, or clip has changed (when such changes don't apply to entire layers). For example, since the transform parent of a `Layer` is only guaranteed to be a transform *ancestor* of contained items, we need to use the previous transform tree to convert a deleted item's bounds into `Layer`-space. Invalidations are discussed further in Part 6.

Part 2: Identify the `SimpleLayers` we need

The main idea is to use the compositing hint list to partition the display list into contiguous sequences of items. Each such contiguous sequence will correspond to a single `SimpleLayer`. We will later (in Part 4) discuss merging multiple `SimpleLayers` into a single `Layer` (where possible, depending on overlap and on the hints that led to the creation of these `SimpleLayers`). Consider the following example:

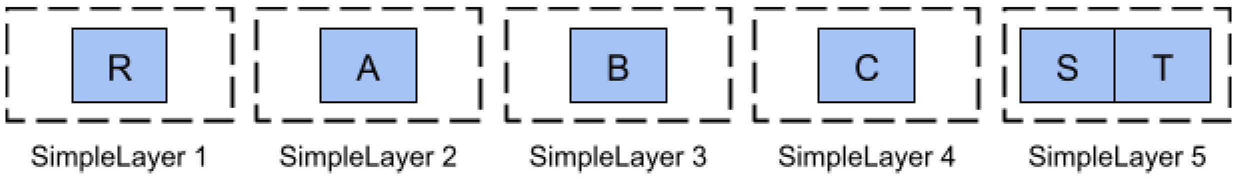
```
<body>
    <div id="R">
       …
    </div>
    <div id="A" style="will-change: transform">
        <div id="B" style="will-change: transform"> … </div>
        <div id="C" style="position: relative"> … </div>
    </div>
    <div id="S" style="position: relative">
       …
    </div>
    <div id="T" style="position: relative">
       …
    </div>
</body>
```

This leads to display list[1] and hint list:



The compositing hint list will has three pairs of hint items: a hint for the root, and a hint for each "will-change: transform".

In this case, we partition the display list into the following five `SimpleLayers`:



Observe that `SimpleLayers` either start at a start hint or start after an end hint (in particular, `SimpleLayers` 1, 2, and 3 each start at a start hint, and `SimpleLayers` 4 and 5 each start after an an end hint).

Here's some pseudo-code implementing this idea. We assume that hints are properly nested, and that every display item is contained within some hint (in particular, this means there must a top-level "root" hint).

Inputs: `HintList hint_list, ContentDisplayList display_list`.
Output: `vector<SimpleLayer*> layer_list`.

```
vector<SimpleLayer*> layer_list;
stack<int> open_hints;
int current_display_index = 0;
for (int i = 0; i < hint_list.size(); ++i) {
  HintDisplayItem hint = hint_list[i];
  // We are not obligated to create layers for each hint -- see
```

[1] The 1-1 correspondence between divs and display list items in this example is a simplification. In reality, a single div may have several display list items. See the display list construction design doc for more details.

```
  // the discussion about ignored hints and OOM in Part 4.
  if (ShouldBeIgnored(hint))
    continue;

  if IsStartHint(hint) {
    int next_display_index = hint.display_list_index;
    open_hints.push(i);
    if (layer_list.size()) {
      layer_list[layer_list.size()-1]->SetDisplayItems(
          display_list, current_display_index, next_display_index - 1);
      layer_list[layer_list.size()-1].closing_hint_id = hint.id;
    }
    current_display_index = next_display_index;

    SimpleLayer* new_layer = new SimpleLayer();
    new_layer->hint_id = hint.id;
    new_layer->source_id_of_nearest_open_hint = hint.source_id;
    layer_list.push_back(new_layer);

  } else {
    // ASSERT(IsEndHint(hint));
    int next_display_index = hint.display_list_index + 1;
    layer_list[layer_list.size()-1]->SetDisplayItems(
        display_list, current_display_index, next_display_index - 1);
    layer_list[layer_list.size()-1].closing_hint_id = hint.id;
    current_display_index = next_display_index;
    open_hints.pop();

    SimpleLayer* new_layer = new SimpleLayer();
    new_layer->hint_id = hint.id;
    new_layer->source_id_of_nearest_open_hint = open_hints.top().source_id;
    layer_list.push_back(new_layer);
  }
}
```

Part 3: Determine how the `SimpleLayers` we already have correspond to the ones we need

Rather than unconditionally creating new layers in the algorithm given in Part 2, we will instead first check whether an existing layer can be re-used. Each `SimpleLayer` is uniquely identified using its `hint_id` (the id of the hint that immediately precedes its display list items); to keep track of existing layers, we will use a hash map whose key is this identifier. Each “`new`

SimpleLayer()” call will be replaced by a call to a CreateOrReuseSimpleLayer function defined as follows:

```
typedef hash_map<int, SimpleLayer*> LayerHashMap;

SimpleLayer* CreateOrReuseSimpleLayer(LayerHashMap* existing_layers,
                                      int hint_id) {
  if (existing_layers->find(hint_id)) {
    return (*existing_layers)[hint_id];
  }
  SimpleLayer* new_layer = new SimpleLayer();
  new_layer.hint_id = hint_id;
  (*existing_layers)[hint_id] = new_layer;
  return new_layer;
}
```

Each SimpleLayer also has a closing_hint_id (the id of the hint that immediately follows its display list items) but we don’t use this for layer identification. Instead, when a SimpleLayer’s closing_hint_id changes, we know that layerization has changed. This means we need to invalidate for items that have been added or removed from the layer (even though such items might not appear in the list of indices sent by Blink, since the items themselves have not changed, only the hints around them have).

We need to re-compute bounds for SimpleLayers that have changed in any way (and for newly-created SimpleLayers).

Part 4: Create Layers consisting of one or more SimpleLayers that can be combined

This involves merging SimpleLayers where possible, considering effects and overlap. Merging multiple SimpleLayers into a single Layer is optional -- without it, we’re guaranteed to have at most 2h layers, where |h| is the total number of compositing hints (excluding any hints we’d like to ignore); with this step, this can be reduced to as few as h layers. While it seems likely that we’ll need to merge in order to avoid the memory cost associated with a large number of layers, it’s worth noting that merging does add some complexity and introduces layer instability[2]. We should re-evaluate this once we have data about per-layer costs in the new system.

If we’re not merging, we create a separate Layer for each SimpleLayer from Part 2.

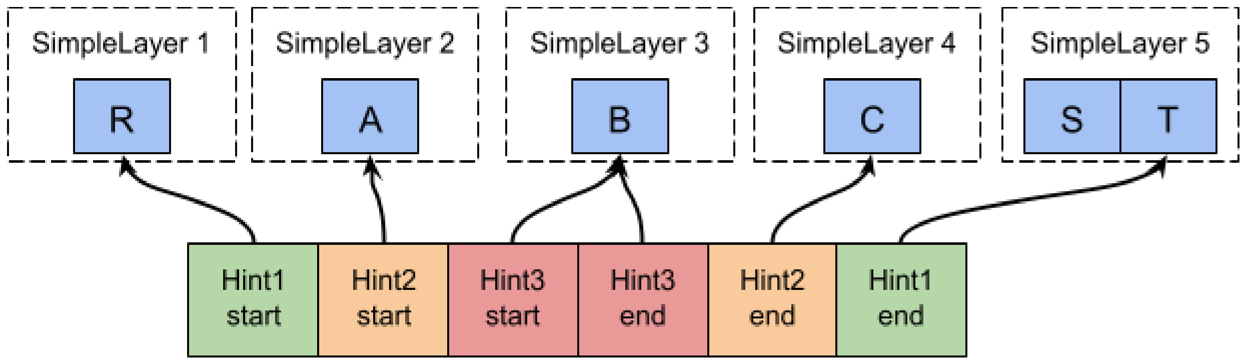
[2] Merging layers creates instability since a change in the position of a single intermediate layer can, by changing overlap, cause previously merged layers to no longer be mergeable, or conversely, can cause previously unmergeable layers to be mergeable.

If merging, only layers with the same value of `source_id_of_nearest_open_hint` are merged. This ensures, for example, that the contents of an element with a will-change hint aren't merged with unrelated items (such merging would mean that animating the element's transform or opacity would require re-rasterization, defeating the purpose of the will-change hint).

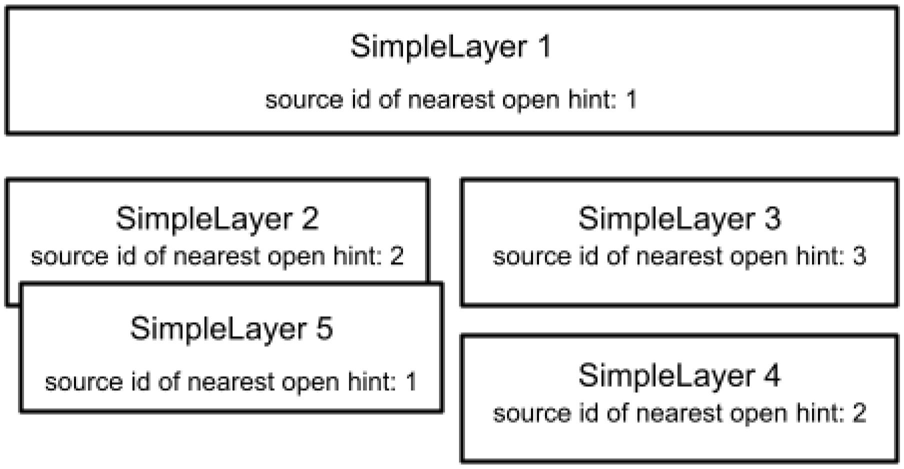
The high level idea is to maintain a stack of `Layers`, arranged bottom-to-top in paint order (so that the bottom-most `Layer` is first in paint order). We iterate over the list of `SimpleLayers`. (this list is already in paint order). For each `SimpleLayer`, we try to find an existing `Layer` to merge into. We start at the top of the stack, and move downwards until we either find an overlapping `Layer`, or one we can merge into. If we find an overlapping `Layer` before finding one we can merge into, we start a new `Layer`, which is added to the top of the stack.

A `SimpleLayer` can be merged into a `Layer` if the `Layer`'s `hint_source_id` matches the `SimpleLayer`'s `source_id_of_nearest_open_hint`, and if the `Layer`'s transform, filter, and clip parents are transform, filter, and clip ancestors of the `SimpleLayer`. (A `Layer`'s `hint_source_id` and its `transform`, `filter`, and `clip` parents are chosen to match the corresponding values in its first `SimpleLayer`.)

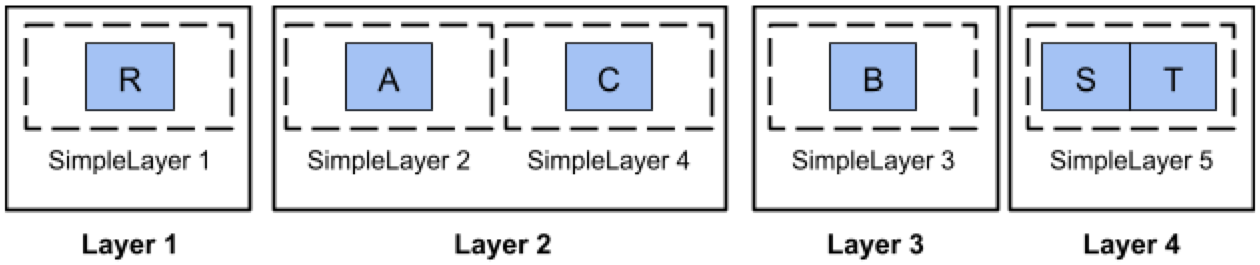
Returning to the example from Part 2, we start with the following display list, hint list, and `SimpleLayers`:



Suppose we're not ignoring any hints, and suppose that the `SimpleLayers` are positioned in the following way:


SimpleLayer 1
source id of nearest open hint: 1
SimpleLayer 2
source id of nearest open hint: 2
SimpleLayer 3
source id of nearest open hint: 3
SimpleLayer 5
source id of nearest open hint: 1
SimpleLayer 4
source id of nearest open hint: 2

`SimpleLayer` 5 overlaps `SimpleLayer` 2, but all other pairs of `SimpleLayers` have no overlap. We will create the following `Layers`:



`SimpleLayer` 4 is able to merge into `Layer` 2 since it shares `SimpleLayer` 2's `source_id_of_nearest_open_hint` (that is, Hint2's `source_id`), and doesn't overlap `Layer` 3. However, `SimpleLayer` 5 cannot merge into `Layer` 3 or `Layer` 2 since its `source_id_of_nearest_open_hint` (Hint1's `source_id`) doesn't match these `Layers`' `hint_source_ids` (`Layer` 3's `hint_source_id` is Hint3's `source_id`, and `Layer` 2's `hint_source_id` is Hint2's `source_id`). Further, since `SimpleLayer` 5 overlaps `Layer` 2 (since it overlaps `SimpleLayer` 2), it cannot merge into `Layer` 1.

One additional wrinkle to consider is that we'd like to avoid creating sparse `Layers` (that is, `Layers` whose area is much larger than the sum of the areas of the contained `SimpleLayers`). This can happen when merging `SimpleLayers` that are far apart. For example, suppose a long page has a header and footer that each get separate `SimpleLayers`. If these two `SimpleLayers` were to be merged into a single `Layer`, this `Layer` would have the same area as the entire page. To address this problem, we can set a maximum allowed sparsity and check whether this condition is satisfied before merging (Blink's current layer squashing does something similar). It's also worth considering whether merging into the bottom-most `Layer` we can merge into (rather than merging into the top-most such `Layer`) will tend to produce denser `Layers`. Merging into the bottom-most `Layer` is more expensive (since we need to keep looking for `Layers` to merge into rather than early-outing once we've found one), so we'll need to run experiments to determine whether it's worth doing.

Here's some pseudo-code for `Layer` creation:

Input: `vector<SimpleLayer*> simple_layer_list`
Output: `vector<Layer*> layer_list`

```
vector<Layer*> layer_list
for (int i = 0; i < simple_layer_list.size(); ++i) {
  for (int j = layer_list.size() - 1; j >= 0; --j) {
    if (simple_layer_list[i]->CanMergeInto(layer_list[j])) {
      layer_list[j]->AddSimpleLayer(simple_layer_list[i]);
      break;
    }
    if (simple_layer_list[i]->Overlaps(layer_list[j]) || j == 0) {
      Layer* new_layer = new Layer();
      new_layer->AddSimpleLayer(simple_layer_list[i]);
      layer_list.push(new_layer);
      break;
    }
  }
}
```

Ignoring hints to avoid OOM
We might choose to ignore some of the hints sent by Blink (e.g., if we decide the cost of a layer outweighs the additional rasterization cost when we don't have the layer). When we decide to do this ahead of time, we can simply omit ignored hints when identifying needed `SimpleLayers` in Part 2. However, we might also decide to ignore certain hints after creating `Layers` and realizing we have too many of them (e.g., the number of layers might be large enough to cause OOM). In this case, we'd repeat the above merging algorithm with the following change: rather than using a `SimpleLayer`'s `source_id_of_nearest_open_hint` to determine which other `SimpleLayers` it can merge with, we'd consider the `source_id` of the nearest open *unignored* hint. This can be found efficiently by creating a hint tree (completely analogous to the transform, filter, and clip trees); the nearest open unignored hint would be the nearest unignored hint ancestor.

Part 5: Determine how the `Layers` we already have correspond to the ones we need

`Layers` are identified by their first contained `SimpleLayer`. As in Part 3 where we discussed re-using `SimpleLayers` rather than always creating new ones, we'll check if we have an existing `Layer` for a `SimpleLayer` before creating a new layer for it in Part 4. When we merge a

`SimpleLayer` into an existing `Layer`, we'll check if it's the same `Layer` that it was merged into the previous frame; if not, we need to invalidate its old bounds in the previous `Layer` and its current bounds in its new `Layer`. Also, we need to invalidate the old bounds in previous `Layer`s for any `SimpleLayer`s from the previous frame that no longer exist in this frame.

Part 6: Determine the invalidation regions for each layer

Invalidation regions are stored on each `Layer`, in the `Layer`'s space (this is defined to be the space established by the transform parent of the first `SimpleLayer` contained by the `Layer`).

There are several types of invalidations to consider:

- Creating or deleting a `SimpleLayer`, or moving a `SimpleLayer` from one `Layer` to another.
    - We invalidate the `SimpleLayer`'s old bounds in its old `Layer` and its new bounds in its new `Layer`.

- Removing an item that was in the previous display list.
    - We are given a list of indices (for the previous list) of such items. We invalidate each such item's previous bounds in its previous `Layer`.

- Adding an item to the display list.
    - We are given a list of indices (for the current list) of such items. We invalidate each such item's bounds in its `Layer`.

- Moving display list items from one `SimpleLayer` to another because of the insertion or removal of hints.
    - These are detected when a `SimpleLayer`'s `closing_hint_id` changes. We invalidate each such item's previous bounds in its previous `Layer` and its new bounds in its new `Layer`.

- Changes to the transform, filter, or clip trees that aren't applied to entire layers.
    - We have start and end items for transforms, filter, and clips. When a change cannot be applied at the layer level, we invalidate the entire sequence of display list items from the start till the end; we need to invalidate both the previous and the new bounds of such items.

Since we need to find the previous bounds of items in many of the cases above, we need to retain information about the previous layerization (e.g., the previous transform, filter, and clip trees, and the previous mapping of display list items to `SimpleLayers`) until we finish computing invalidation rects.