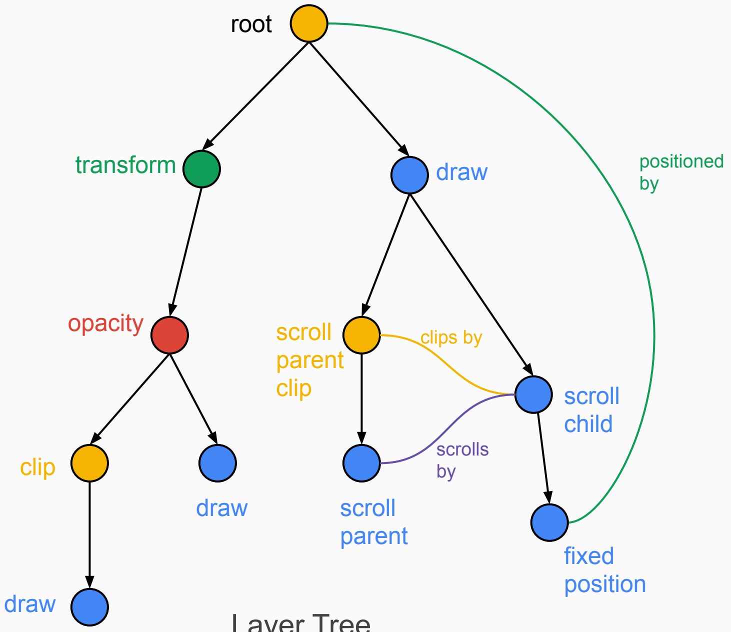
Compositor Property Trees

Property tree overview

Transforms and clips and effects, oh my!

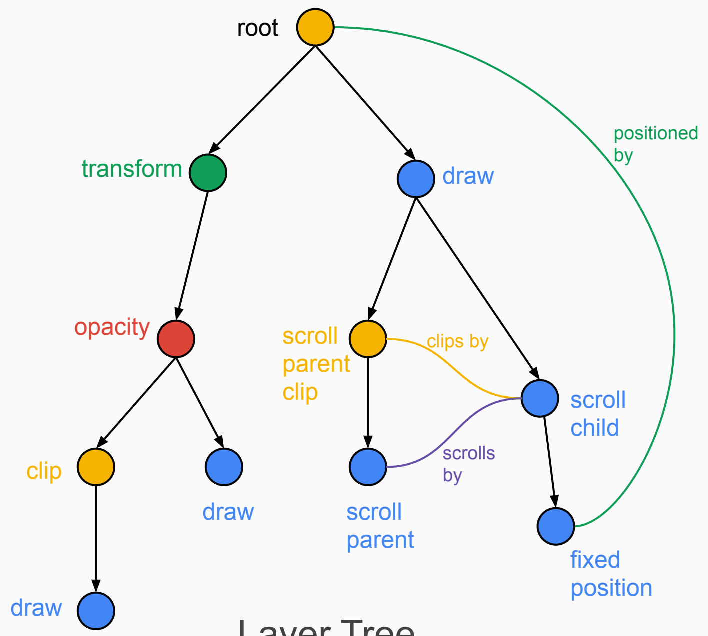


Layer Tree

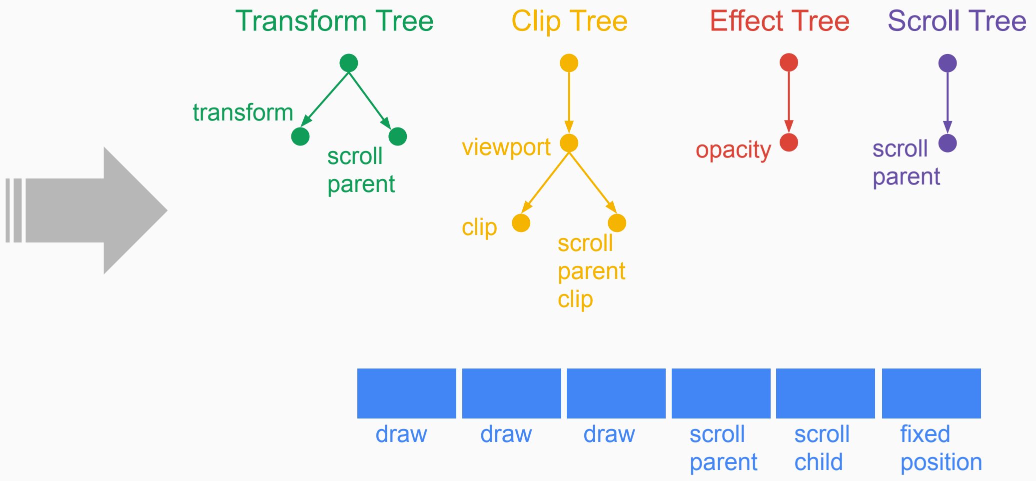
Before:

- Single layer tree
- Topology determines drawing order
  - Pre-order traversal
- But scrolling, clipping, positioning don't necessarily follow the same hierarchy
  - Workarounds: scroll_parent, clip_parent

Multiple hierarchies with property trees

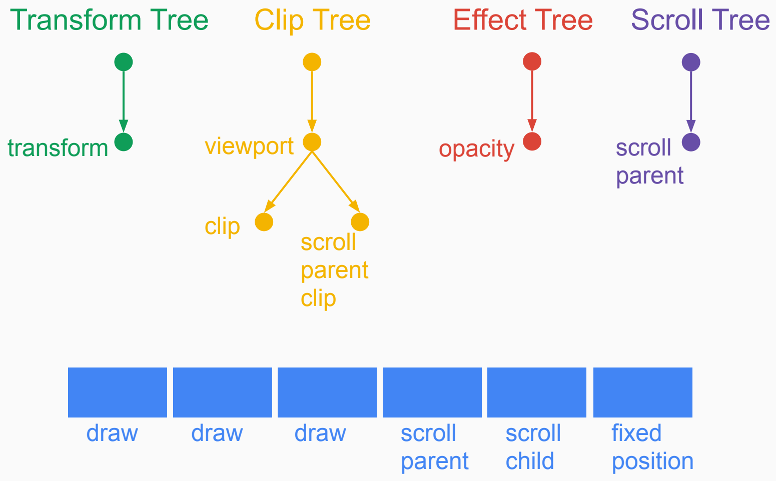


Layer Tree



Property trees + Layer List

Property trees



Property trees + Layer List

Trees are sparse -- not every layer has an interesting transform, clip, effect, or scroll

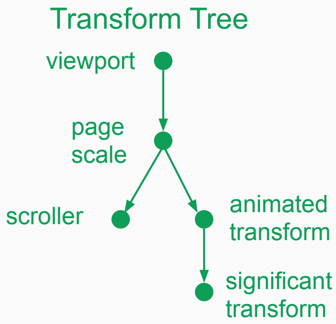
But cross-tree dependencies do exist

- e.g. clips and effects happen in a particular transform space

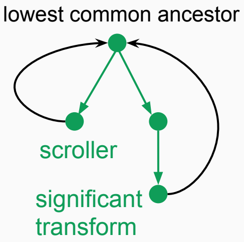
Layer list still has non-drawing layers (for now)

Transform tree

- Each node defines a new space
    - by defining how to map to parent node's space
- The root node defines viewport space
- To map between two nodes, use mapping to their lowest common ancestor



A transform is *significant* if it isn't a 2d translation



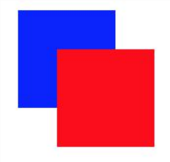
Render surfaces

Buffers that hold the output of drawing operations

Some operations require their inputs to first be drawn to an intermediate buffer:

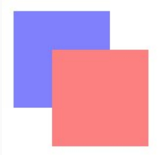
- opacity
- filter
- mask
- blending
- non-axis-aligned clipping
- copy requests

Inputs



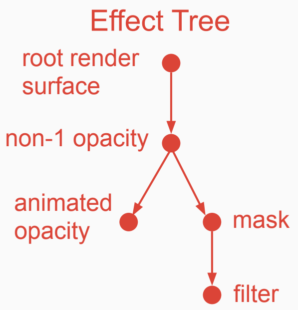
Opacity 0.5, no render surface

Opacity 0.5, with render surface



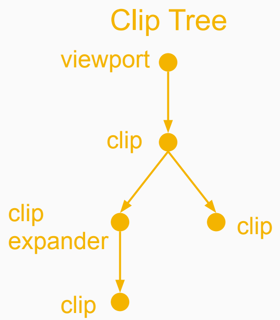
Effect tree

- Each node represents a drawing operation that may require a render surface
- Each node's output is an input to its parent node
- The root node represents the root render surface



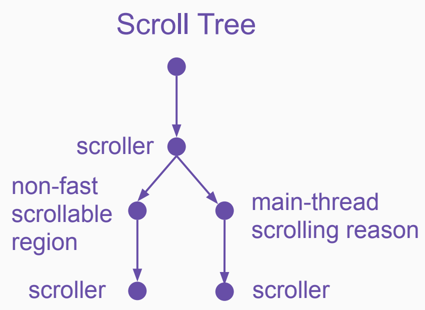
Clip tree

- Each node represents a clip defined in some transform space
  - Or a “clip expansion” caused by pixel-moving filters
- The root node represents the viewport clip
- Layers and effects that point to clip nodes respect all clips from that node to the root



Scroll tree

- Each node is scrollable or contains information needed to make scrolling decisions
  - reasons we can't scroll on the compositor
  - maximum scroll offset
- Path from node to root defines a scroll chain
- Scroll offsets are owned by the scroll tree
  - SyncedScrollOffset instances used to sync scroll deltas between main and compositor threads



Layers

Each layer points to a node from each of the four trees

- `transform_tree_index`, `clip_tree_index`, `effect_tree_index`, `scroll_tree_index`

Each layer also has

- 2d translation to transform node's space
  - `offset_to_transform_parent`
- whether it needs to "flatten" the transform node's space
  - `should_flatten_transform_from_property_tree`

Compositor driven effects

Updating property trees

Property trees are only built/re-built on the main thread

- Rebuilding on the main thread happens only when needed
    - Layer::SetNeedsCommit vs Layer::SetNeedsCommitNoRebuild

- Trees are copied at commit and activation

- Compositor-driven effects update existing nodes

Scrolling

Given an input point in screen space:

- Hit test on the layer list
  - Map from screen space to layer space using transform tree
  - Check if hit point is clipped out using clip tree
- Get scroll chain from scroll tree
  - Make sure nothing in the chain has a main-thread scrolling reason
  - Find the first scrollable node, update its scroll offset
- Update the corresponding node in the transform tree

Animation

Every compositor animation is associated with a property tree node

Each time an animation ticks:

- find the corresponding property tree node
    - using an ElementId
- update it using the new animated value

Property tree outputs

Property tree outputs: draw properties

Layer draw properties

- Screen space transform
- Draw transform
- Draw opacity
- Visible layer rect
- Clip rect and is_clipped
- Drawable content rect
- Animation scale

Render surface draw properties

- Screen space transform
- Draw transform
- Draw opacity
- Content rect
- Clip rect and is_clipped
- Drawable content rect

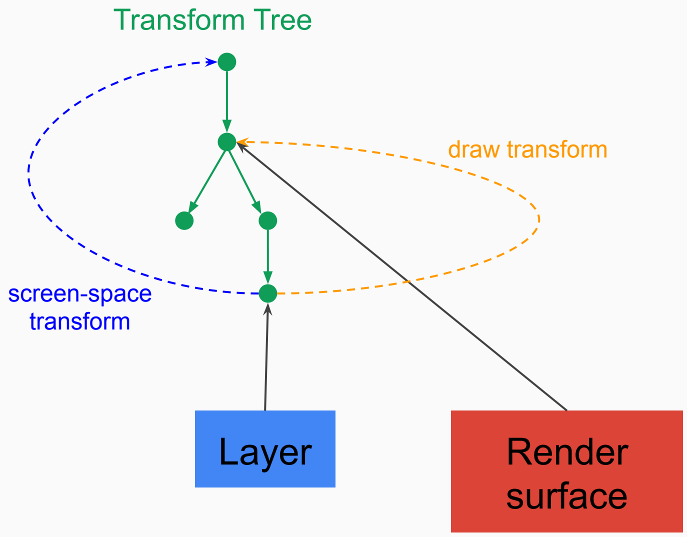
Screen-space transform and draw transform

Screen-space transform: maps from local space to screen space

- used for hit-testing

Draw transform (target-space transform): maps from local space to space of target render surface

- applied when drawing
- used to compute a raster scale



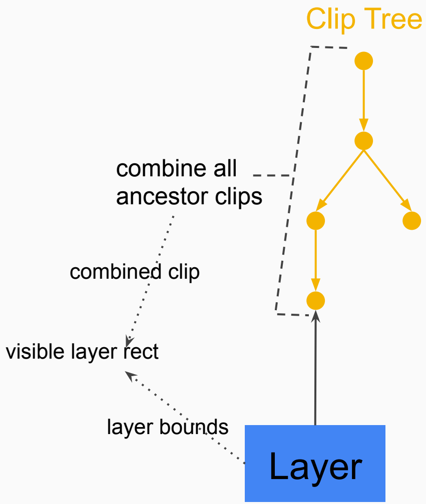
Visible layer rects

The part of a layer that's visible, taking into account all clipping, expressed in layer space

Used to decide:

- what part of a layer to raster
- which layer quads to include at AppendQuads time

Overestimating hurts performance but not correctness

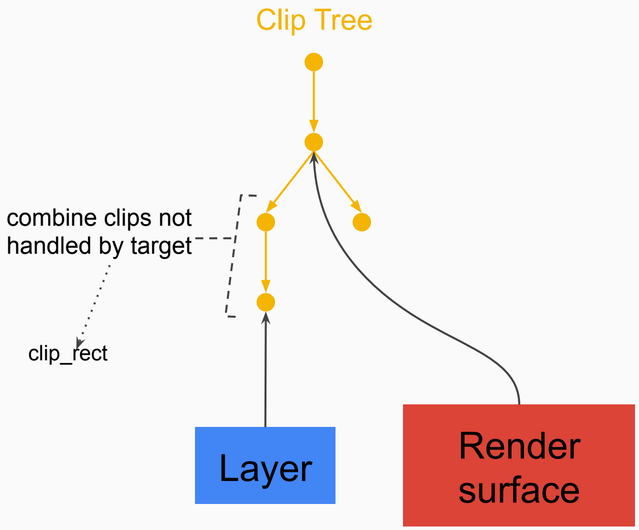


clip_rect and is_clipped

is_clipped: whether a clip needs to be applied at draw time

clip_rect: the clip to apply, expressed in target space

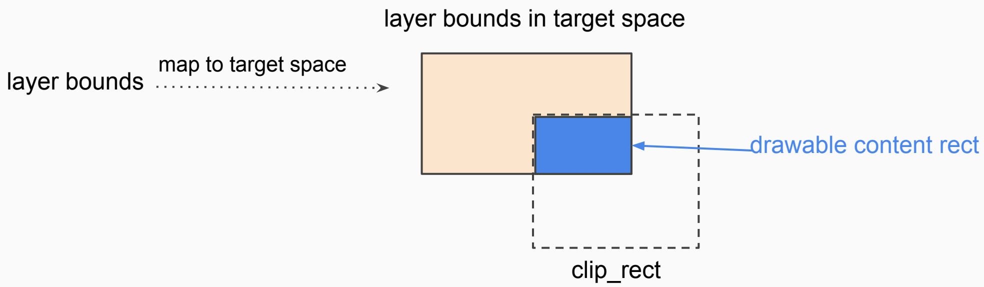
Unlike visible layer rect, this needs to be computed exactly for correctness



Drawable content rect

Size of layer in target space, intersected with clip_rect if is_clipped

Used to compute target surface's content rect



Implementation details

Where’s the code?

Files in `cc/trees`:

- `property_tree.{h, cc}, {clip, effect, scroll, transform}_node.{h, cc}`
  - tree implementation and update logic
- `draw_property_utils.{h, cc}`
  - draw property computation
- `property_tree_builder.{h, cc}`
  - builds trees (but won’t be used for renderers in SPv2)
- `layer_tree_host_common.{h, cc}`
  - logic for constructing the render surface layer list
- `layer_tree_host_common_unittest.cc`
  - tests for draw property computation

Property tree implementation

```
template <typename T>
class PropertyTree {
 public:
  int Insert(const T& tree_node, int parent_id);
  T* Node(int id);
  T* parent(const T* t);

 private:
  std::vector<T> nodes_;
};
```

Property tree node implementation

Conceptually, each node type is:

```
template <typename ValueType>
struct PropertyTreeNode {
  int id;
  int parent_id;
  int owning_layer_id;  // stable across property tree rebuilds
  ValueType value;
};
```

But nodes are much larger in reality...

Transform node implementation

Additional fields:

```
gfx::Transform pre_local
gfx::Transform local
gfx::Transform post_local
gfx::Transform to_parent
int sticky_position_constraint_id
int source_node_id
int sorting_context_id
bool needs_local_transform_update
bool node_and_ancestors_are_animated_or_invertible
bool is_invertible
bool ancestors_are_invertible
bool has_potential_animation
bool is_currently_animating
bool to_screen_is_potentially_animated
bool has_only_translation_animations
```

```
bool flattens_inherited_transform
bool node_and_ancestors_are_flat
bool scrolls
bool should_be_snapped
bool moved_by_{inner, outer}_viewport_bounds_delta_{x,y}
bool in_subtree_of_page_scale_layer
bool transform_changed
float post_local_scale_factor
gfx::ScrollOffset scroll_offset
gfx::Vector2dF snap_amount
gfx::Vector2dF source_offset
gfx::Vector2dF source_to_parent
```

Inputs
Cached data

Avoiding tree walks when computing transforms

ToScreen and FromScreen are computed and cached for every node

Then, to compute the transform from node i to node j:

FromScreen(j) * ToScreen(i)

...unless flattening gets in the way

Flattening

Flattening is a non-linear operation:

a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

a	b	0	d
e	f	0	h
0	0	1	0
m	n	0	p

$(\mathrm{flatten}(A))^{-1} \neq \mathrm{flatten}(A^{-1})$

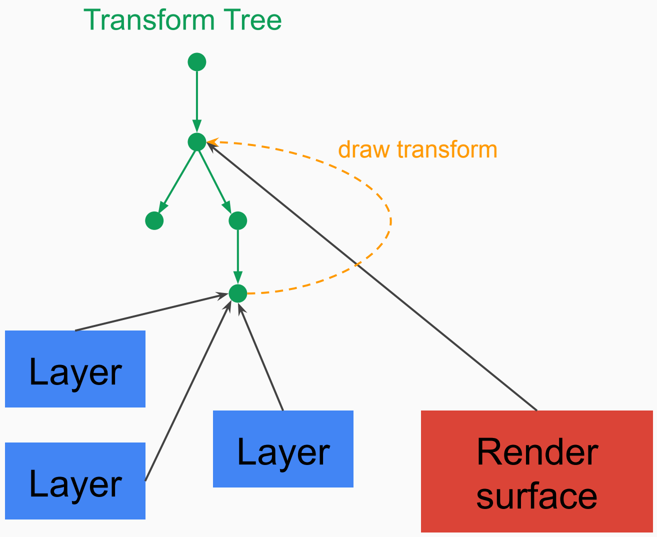
Flattening between two nodes breaks the ToScreen/FromScreen trick

Need to tree walk... but the result can be cached

Transform caching

Multiple layers with the same target can point to the same transform node

The first time we compute a draw transform involving a particular pair of nodes, the result gets cached



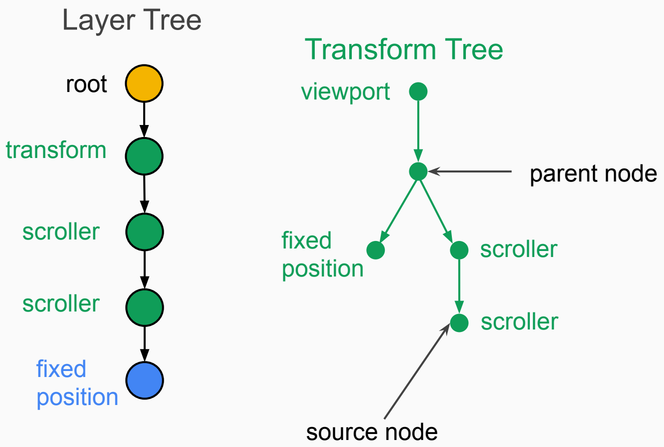
Fixed position

Old layer-tree-based logic: undo scroll deltas in between a fixed-position layer and its container

New logic: rely on transform tree topology

Catch: Blink still positions fixed-position layers wrt their layer tree parent, undoing ancestor scrolling

- This repositioning would ordinarily require a property tree rebuild
- Solution: track source_to_parent_offset



Effect node implementation

Additional fields:

```
float opacity
float screen_space_opacity
FilterOperations filters
FilterOperations background_filters
gfx::PointF filters_origin
SkBlendMode blend_mode
gfx::Vector2dF surface_contents_scale
gfx::Size unscaled_mask_target_size
bool has_render_surface
RenderSurfaceImpl* render_surface
bool surface_is_clipped
bool has_copy_request
bool hidden_by_backface_visibility
bool double_sided
bool is_drawn
bool subtree_hidden
```

```
bool has_potential_{filter, opacity}_animation
bool is_currently_animating_{filter, opacity}
bool effect_changed
int num_copy_requests_in_subtree
bool has_unclipped_descendants
int transform_id
int clip_id
int target_id
int mask_layer_id
```

Inputs

Cached data

Clip node implementation

Additional fields:

```
ClipType clip_type
gfx::RectF clip
std::unique_ptr<ClipExpander> clip_expander
gfx::RectF combined_clip_in_target_space
gfx::RectF clip_in_target_space
int transform_id
int target_transform_id
int target_effect_id
bool layer_clipping_uses_only_local_clip
bool layers_are_clipped
bool layers_are_clipped_when_surfaces_disabled
bool resets_clip
```

Inputs

Cached values -- will be removed when caching is moved

Inputs -- will be removed when caching is moved

Clip caching

Current approach: caching at each clip node before computing each layer's draw properties

- **combined_clip_in_target_space**: used for visible layer rect
- **clip_in_target_space**: used for clip_rect

New approach: combine clips on-demand, cache results in separate structure

- removes dependency of clip tree on render surfaces
- logic is easier to understand

Scroll node implementation

Additional fields (all inputs):

```
bool scrollable
uint32_t main_thread_scrolling_reasons
bool contains_non_fast_scrollable_region

gfx::Size scroll_clip_layer_bounds
gfx::Size bounds
bool max_scroll_offset_affected_by_page_scale

bool is_{inner, outer}_viewport_scroll_layer

gfx::Vector2dF offset_to_transform_parent
bool should_flatten

bool user_scrollable_horizontal
bool user_scrollable_vertical
```

Building property trees

Now:

- cc::PropertyTreeBuilder builds property trees given a layer tree
- TreeSynchronizer converts layer tree to layer list

In SPv2, for renderers:

- blink::PaintArtifactCompositor will build a layer list + property trees

ui will continue using cc::PropertyTreeBuilder for now

Unit tests need cc::PropertyTreeBuilder too...

Unit tests

cc has **lots** of unit tests that construct layer trees!

Tests that construct trees of `Layer` (e.g. LayerTreeTests):

- use PropertyTreeBuilder to convert to property trees
- on the compositor side, access layers using ids instead of hierarchy

Tests that construct trees of `LayerImpl`:

- members needed only for property tree building moved to LayerImplTestProperties
- LayerTreeImpl::BuildLayerListAndPropertyTreesForTesting

Android WebView

Applies changes on the compositor thread

Easy to handle:

- external transform
- external viewport

Trickier: Resourceless software mode

- disallows non-root render surfaces
- target-dependent property tree logic needs special casing, for now

Roadmap

Completed work

- Main thread property trees, shipped M44
- Compositor-thread property trees, shipped M49
    - Caching inside property trees
    - Main goal was to match output and performance of CalcDrawProps
- Compositor layer lists, shipped M53

Performance

Draw properties computation time

	M48 (CDP)	M53 (Property trees + layer lists)	
Android - 50th percentile	0.19 ms	0.16 ms	-15.8%
Android - 99th percentile	2 ms	1.55 ms	-22.5%
Mac - 50th percentile	0.04 ms	0.034 ms	-15%
Mac - 99th percentile	0.4 ms	0.34 ms	-15%
Windows - 50th percentile	0.033 ms	0.027 ms	-18.2%
Windows - 99th percentile	1 ms	0.97 ms	-3%

Current and future work

- Finish moving caching out of tree nodes
- Finish removing dependencies that don’t fit with SPv2
  - remove dependencies on layers owning property tree nodes
  - remove render target information from clip and transform trees
- Renderer property trees built by Blink
- Finish moving fields from LayerImpl to LayerImplTestProperties

Questions?