Compositing in Blink / WebCore

From WebCore::RenderLayer to cc:Layer

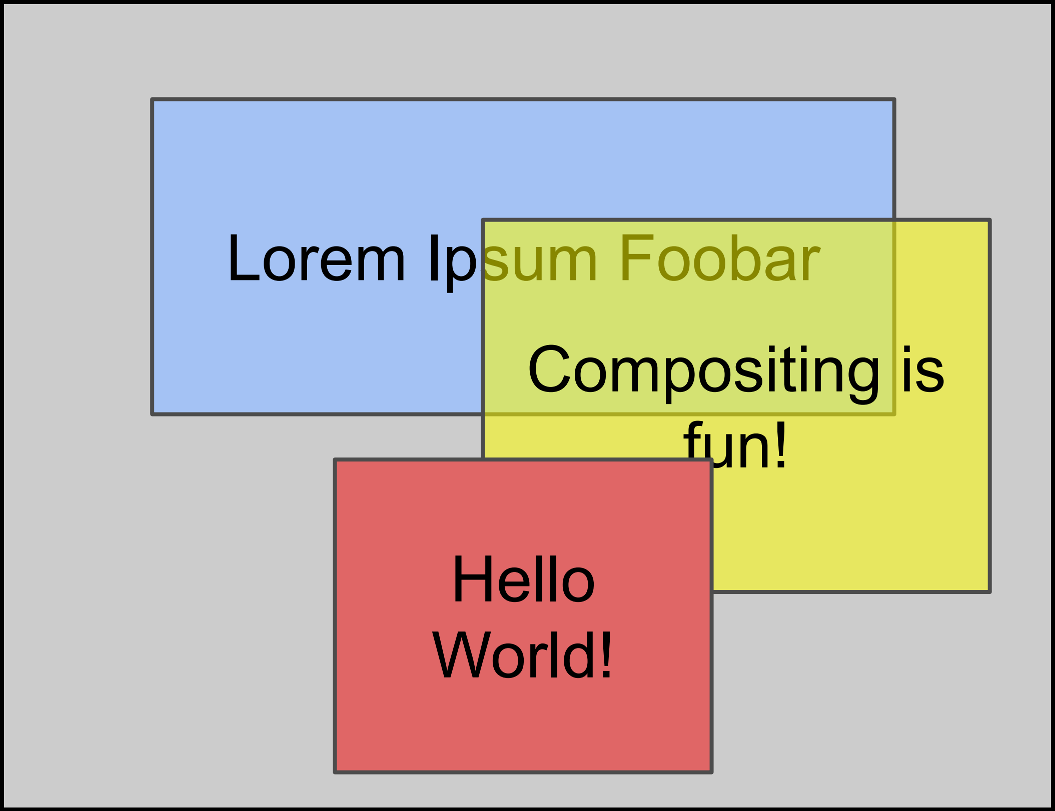
January 2014 update: Basic information here is still relevant. An updated perspective, specifically about upcoming changes to overlap and "squashing", can be found here

(Last updated October 2013)

(Hint - use presentation mode to see animations)

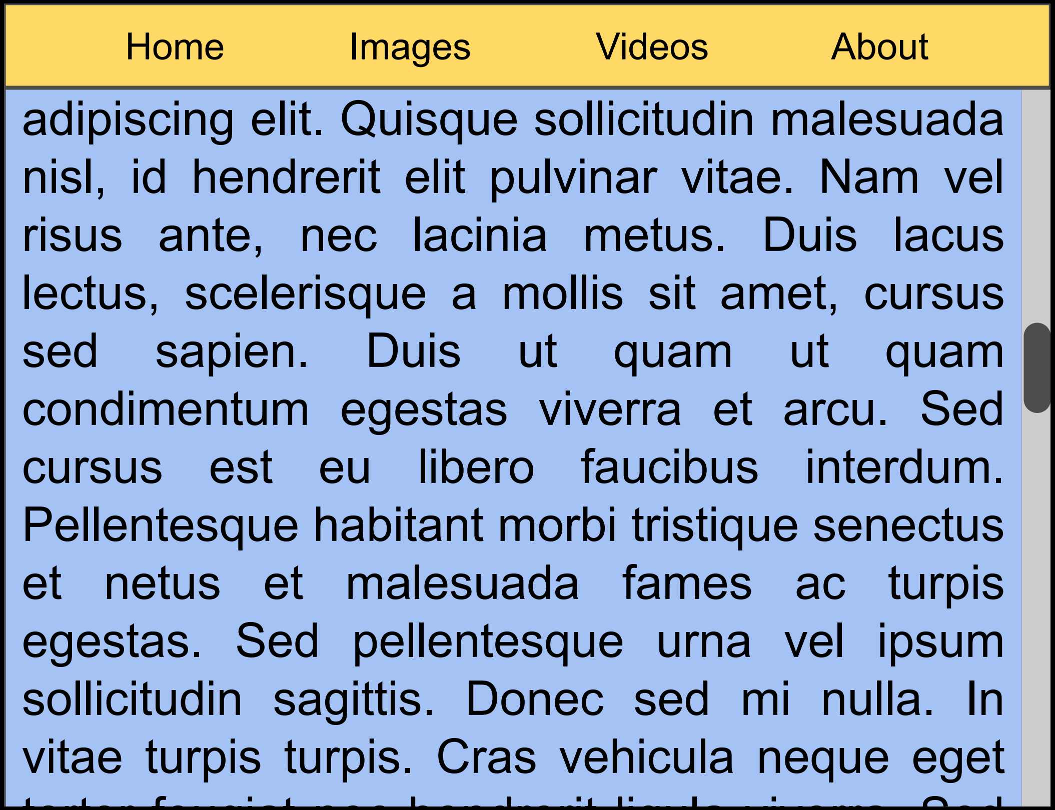
High-level overview

Example 1



What needs to be repainted as this animates?
With one backing store (pixel buffer):
Portions of all four layers!

Example 2



adipiscing elit. Quisque sollicitudin malesuada nisl, id hendrerit elit pulvinar vitae. Nam vel risus ante, nec lacinia metus. Duis lacus lectus, scelerisque a mollis sit amet, cursus sed sapien. Duis ut quam ut quam condimentum egestas viverra et arcu. Sed cursus est eu libero faucibus interdum. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Sed pellentesque urna vel ipsum sollicitudin sagittis. Donec sed mi nulla. In vitae turpis turpis. Cras vehicula neque eget

What needs to be repainted as this scrolls?

With one backing store (pixel buffer):

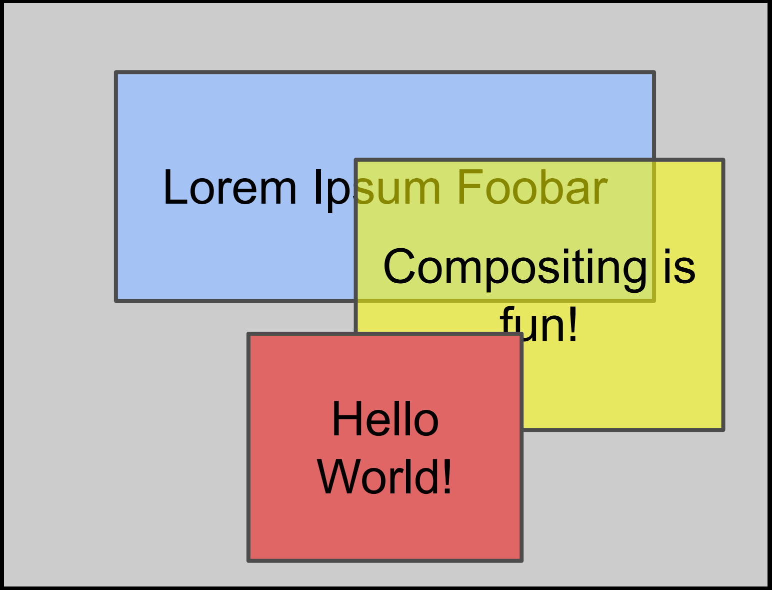
Almost the entire page!

Compositing:

(in the context of rendering websites)

The use of multiple backing stores to cache and group chunks of the render tree.

Why Compositing?



- Avoid unnecessary repainting
  - If yellow and red have their own backing stores, then nothing needs "repainting" while this example animates.
- Makes some features more efficient or practical
  - Including: Scrolling, 3D CSS, opacity, filters, WebGL, hardware video decoding, etc.

Three Primary Compositing Tasks

1. Determine how to group contents into backing stores (i.e. *composited layers*).

2. Paint the contents of each composited layer.

3. Draw the composited layers to make a final image.

The focus of this talk is Step 1, with a little bit about Step 2.

References for the Big Picture

- WebCore guts, including `WebCore::RenderLayer`
  - Eric Seidel's talk - http://www.youtube.com/watch?v=RVnARGhhs9w
- Chromium and Skia side of painting
  - Brett Wilson's talk - http://www.youtube.com/watch?v=A5-aXfSt-RA
- How Chromium's compositor works, from `cc::Layer` to GPU process
  - GPU accelerated compositor design doc - http://dev.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome
  - Reducing jank from costly repaints - http://jankfree.org/
- Connecting `WebCore::RenderLayer` to `cc::Layer`
  - This talk!

Background: Stacking Contexts and Paint Order

(Refer to CSS 2.1 spec for more information)

Flow, Positioning, and Z-index

- Normal flow: children are laid-out according to inline-level, block-level, float, and other formatting.
- Relative positioned elements: positioned relative to their intended position as part of normal flow.
- Absolute positioned elements: positioned with respect to containing block. Not part of normal flow.
- Fixed-position elements: positioned with respect to viewport or other container. Not part of normal flow.

Flow, Positioning, and Z-index

- Relative positioned elements
- Absolute positioned elements
- Fixed-position elements

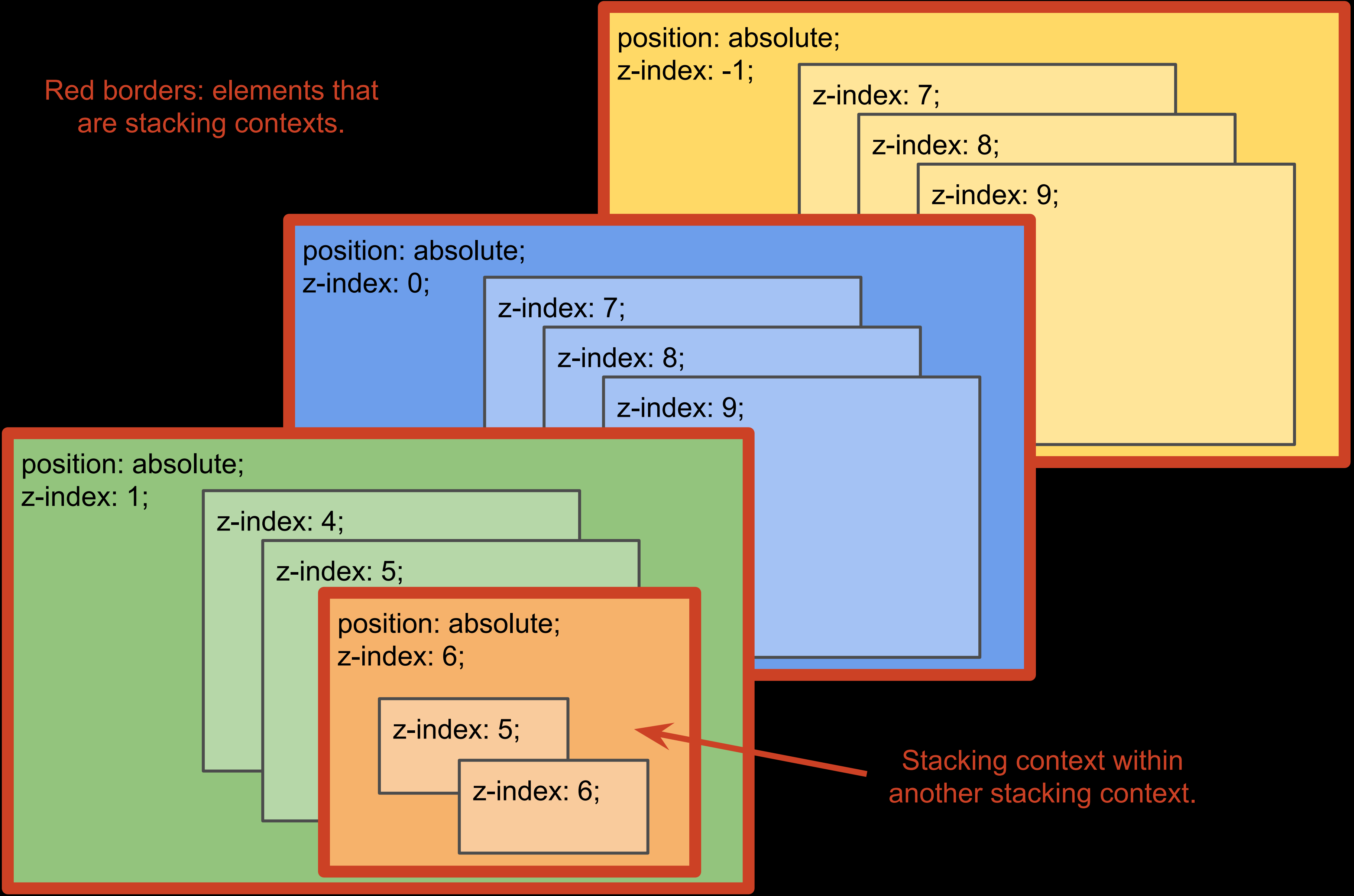
Z-index: allows control over how elements are ordered.

Positioned (*relative* or *absolute* or *fixed***) elements with a z-index become a *stacking context***.

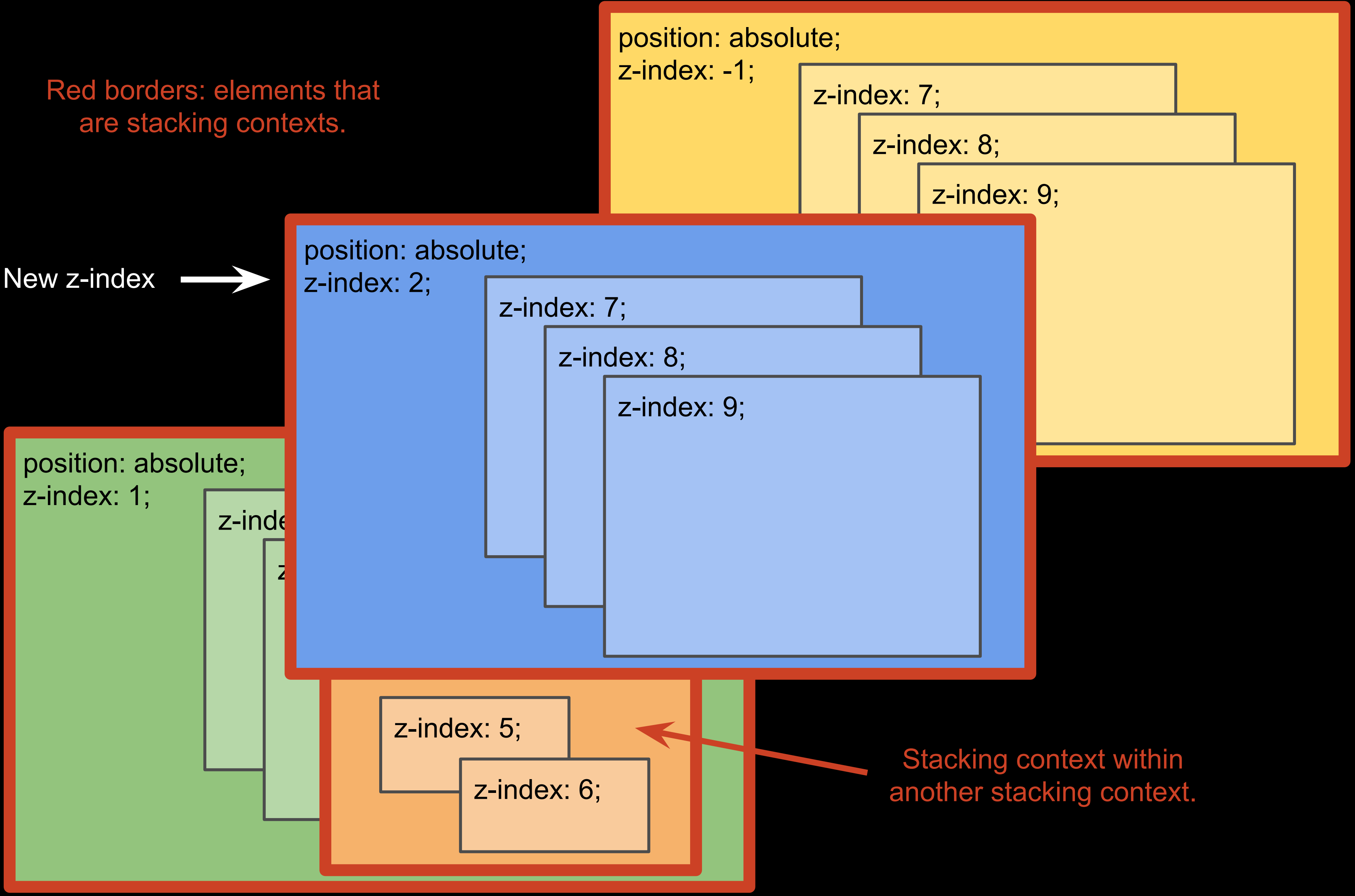
** These days fixed-position elements may also create stacking contexts for many browsers, though it is not part of CSS 2.1 spec. Other features also create stacking contexts extending the CSS 2.1 spec, such as transforms.

Stacking Contexts Example 1

Red borders: elements that are stacking contexts.



Stacking Contexts Example 2



Stacking Contexts - Intuition

A stacking context "flattens" the element's subtree, so nothing outside of the subtree can paint between elements of the subtree.

In other words: The rest of the DOM tree can treat the stacking context as an atomic conceptual layer for painting.

With this property, stacking contexts are an ideal place to define paint order explicitly.

Paint Order of a Stacking Context

1. Backgrounds and borders
2. Negative z-index children
3. Normal flow elements**
4. Z-index == 0 and/or absolute positioned children
5. Positive z-index children

** This is not the complete picture, see CSS 2.1 Section 9.9 and Appendix E.

Equivalent Order as Implemented

1. Backgrounds and borders

2. Negative z-order list of children

3. The RenderLayer's own contents

4. Normal flow list of children

5. Positive z-order list of children

Implementation detail: the "paint-order tree" is not the same topology as the DOM tree. A parent-child pair in the DOM may be siblings in paint order.

Choosing How to Composite Layers

Reasons to Make a Composited Layer

Composite when the render subtree could benefit from being cached or grouped:

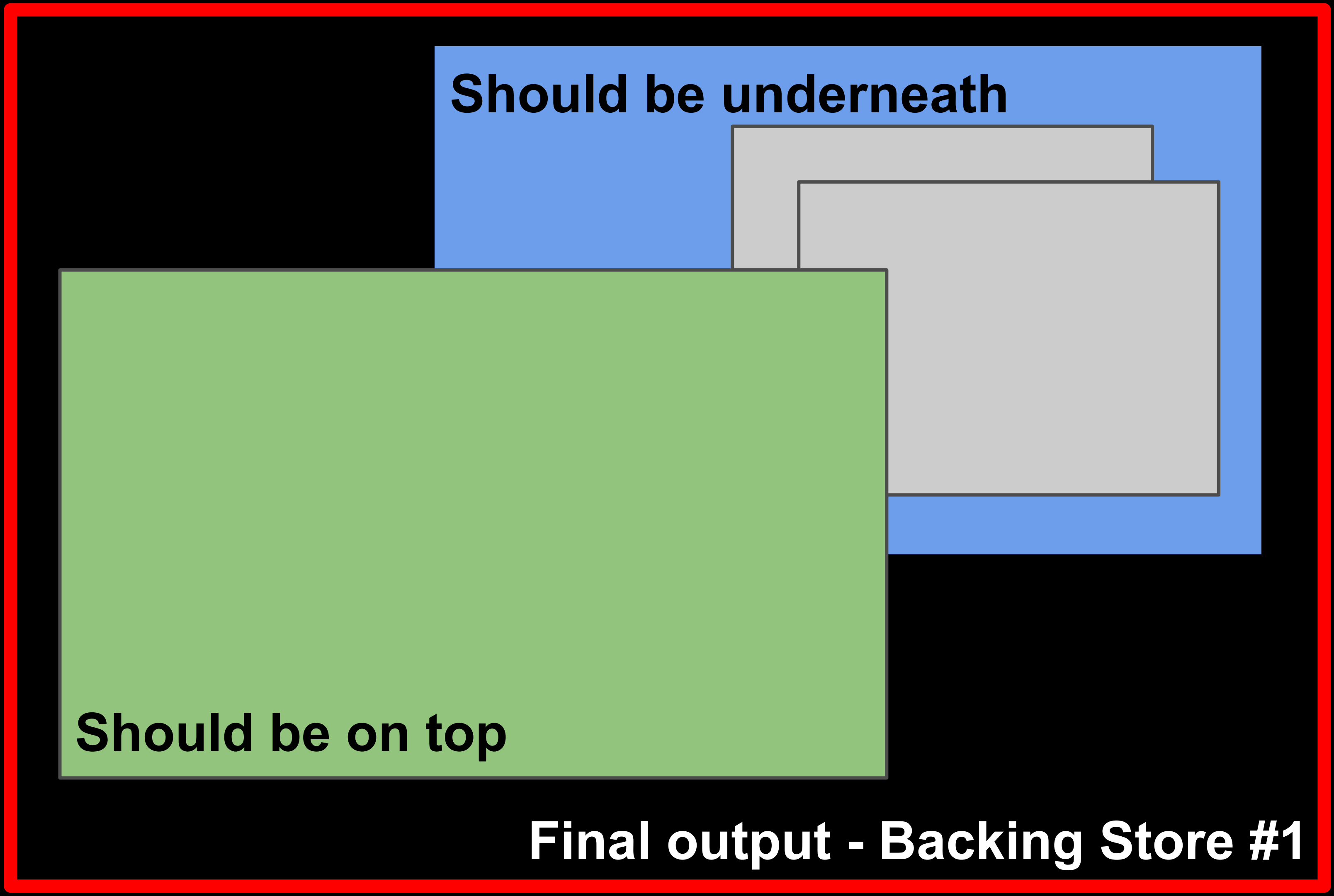
- Easier to apply certain effects to a subtree
  - e.g. opacity, transforms, filters, reflections
- Elements can move without repainting
  - e.g. scrolling, fixed-position elements
- More practical for hardware accelerated content
  - e.g. video, webGL, no need to read-back the pixels
- Potentially isolate content that repaints a lot
  - Just speculation at this point

Reasons to Make a Composited Layer

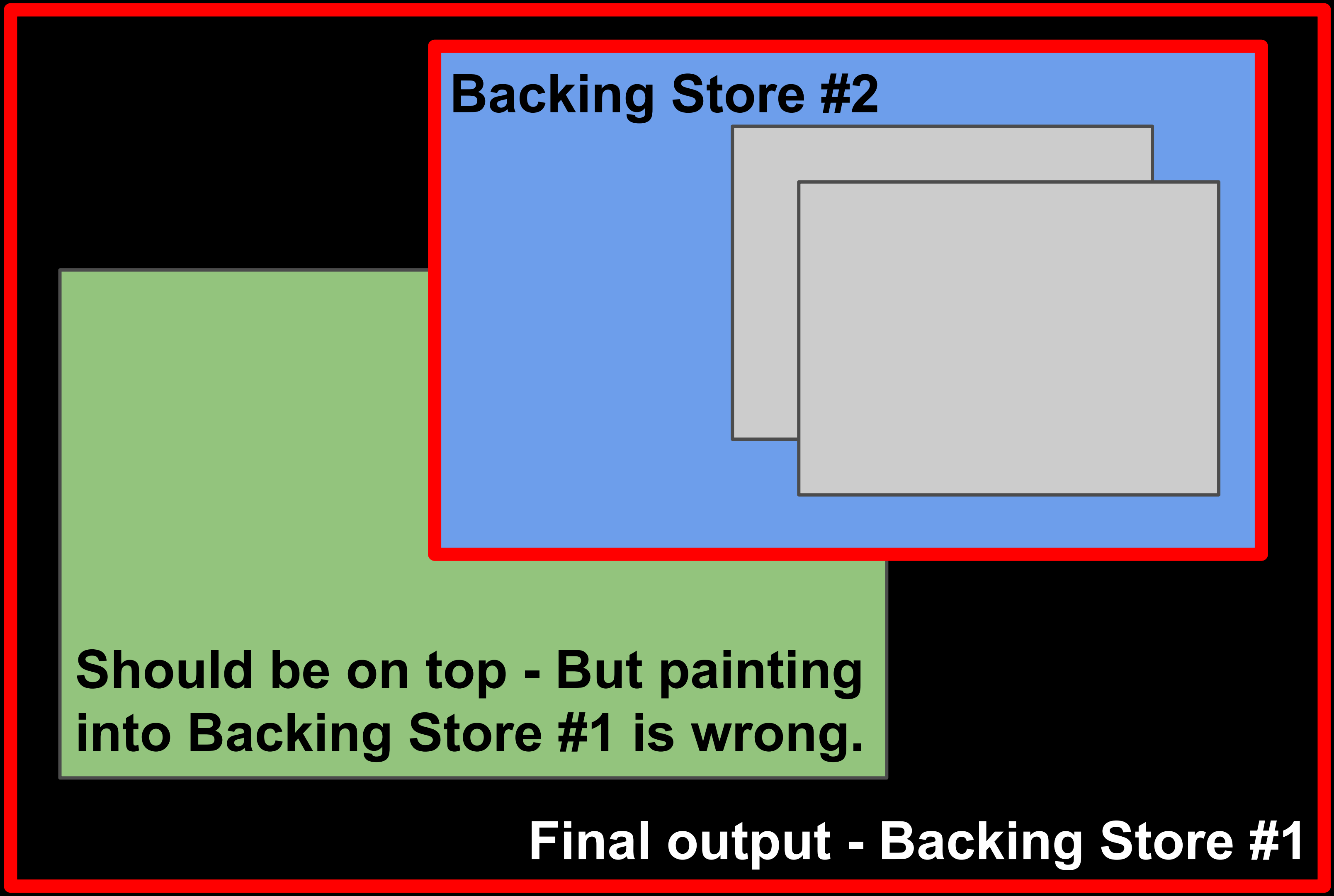
Composite when it is necessary to maintain correctness:

- To maintain correct paint order
  - Overlapping content must be on top of composited content.
  - Example shown next
- To ensure style properties correctly propagate to the composited layer tree
  - For example, a parent that clips a composited descendant must also be composited.
  - Requires knowing the compositing reasons of descendants

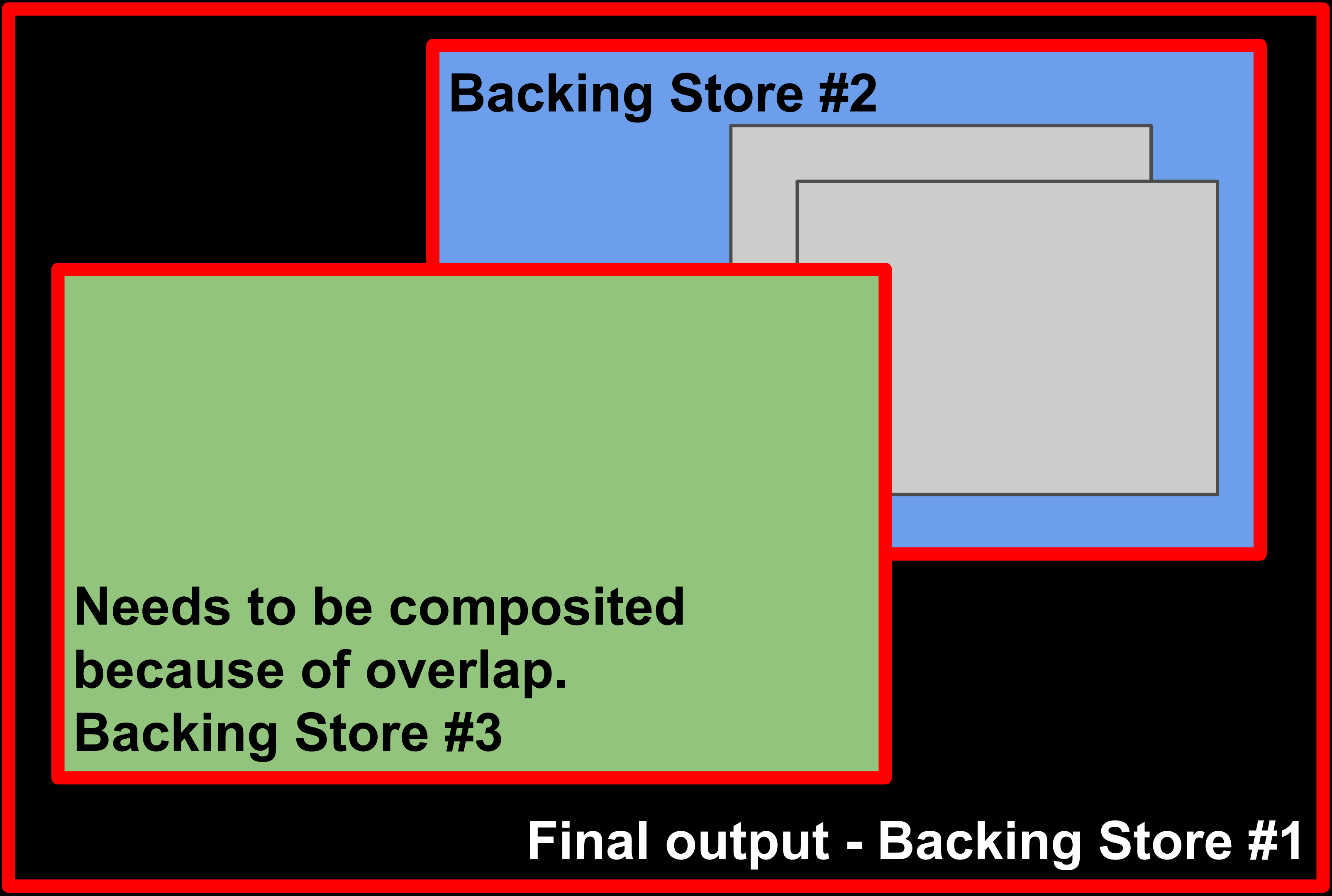
Overlap: Basic Example - Desired



Overlap: Basic Example - Wrong!!


Backing Store #2
Should be on top - But painting into Backing Store #1 is wrong.
Final output - Backing Store #1

Overlap: Basic Example - Correct


Backing Store #2
Needs to be composited because of overlap.
Backing Store #3
Final output - Backing Store #1

More Overlap Cases

- Composited negative z-index child
  - Requires parent to be composited, too
- Overlap during animations
  - Don't bother testing overlap, just assume everything subsequently does overlap.
- Children of a composited container
  - Don't need to check for overlap against anything outside of their composited container.

Simplified view of the algorithm

Iterate over children in the "paint-order tree". For each layer:

1. Determine if the layer needs compositing due to overlap
   a. If something may be animating behind it, assume it overlaps and skip the computation
   b. Otherwise, iterate over a list of bounding boxes of previous composited content, and test intersection

2. Determine if the layer needs compositing based on its own properties
   a. A long list of conditions including 3d transforms, opacity, fixed-position elements (sometimes), etc.
   b. `RenderLayerCompositor::directReasonsForCompositing()`

3. Recurse over children in paint order, repeat all these steps for each child.

4. Determine if the layer needs compositing due to status of the subtree
   a. In particular, if the layer needs to be composited so that clip, transform, or other information needs to propagate to the composited tree.
   b. `RenderLayerCompositor::subtreeReasonsForCompositing()`

Chromium flags for insight into the compositor

`--force-compositing-mode`

Pages that don't "require" compositing will still use it

`--show-composited-layer-borders`

Visualize borders (and tiles) on composited layers.

`--show-paint-rects`

Visualize what layers required repainting

`--show-property-changed-rects`

Visualize what layers required redrawing without repainting

The frame viewer is extremely insightful, too!

http://www.chromium.org/developers/how-tos/trace-event-profiling-tool/frame-viewer

Real-world compositing examples

- Poster Circle
  - Animations disable overlap testing and conservatively composite - try adding a stacking context that does not overlap anything - it still gets composited!
- MapsGL
  - HTML controls and popups easily overlayed on top of WebGL content.
- Android apps page
  - See composited layers come and go while transition animations are playing. Notice clipping elements and 3d elements usually become layers.

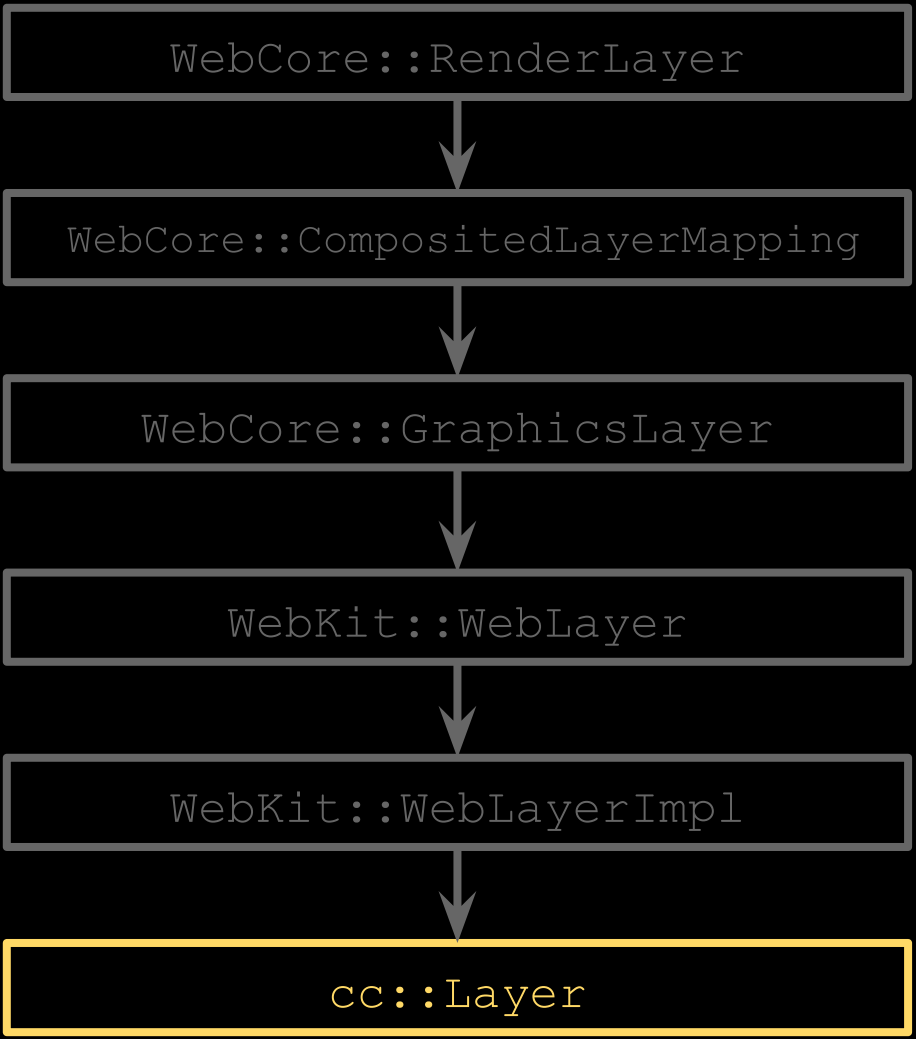
Data Structures and Code Paths

(As of October 2013 - code is likely to evolve and change)

Data Structures

`cc::Layer`

The public object of the Chromium Compositor's interface that represents a composited layer.

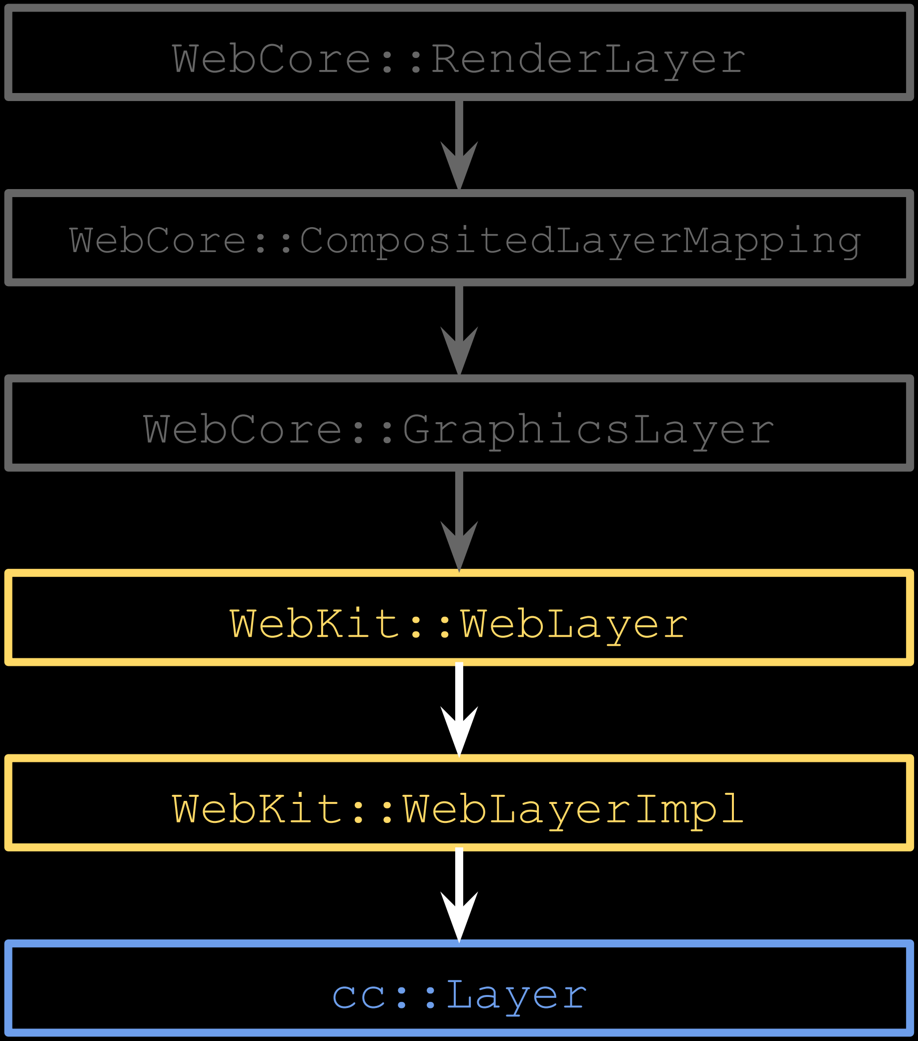


Data Structures

`WebKit::WebLayer`
`WebKit::WebLayerImpl`

This is essentially the `cc::Layer` interface made available to WebKit code, so that it can be used by `GraphicsLayer`

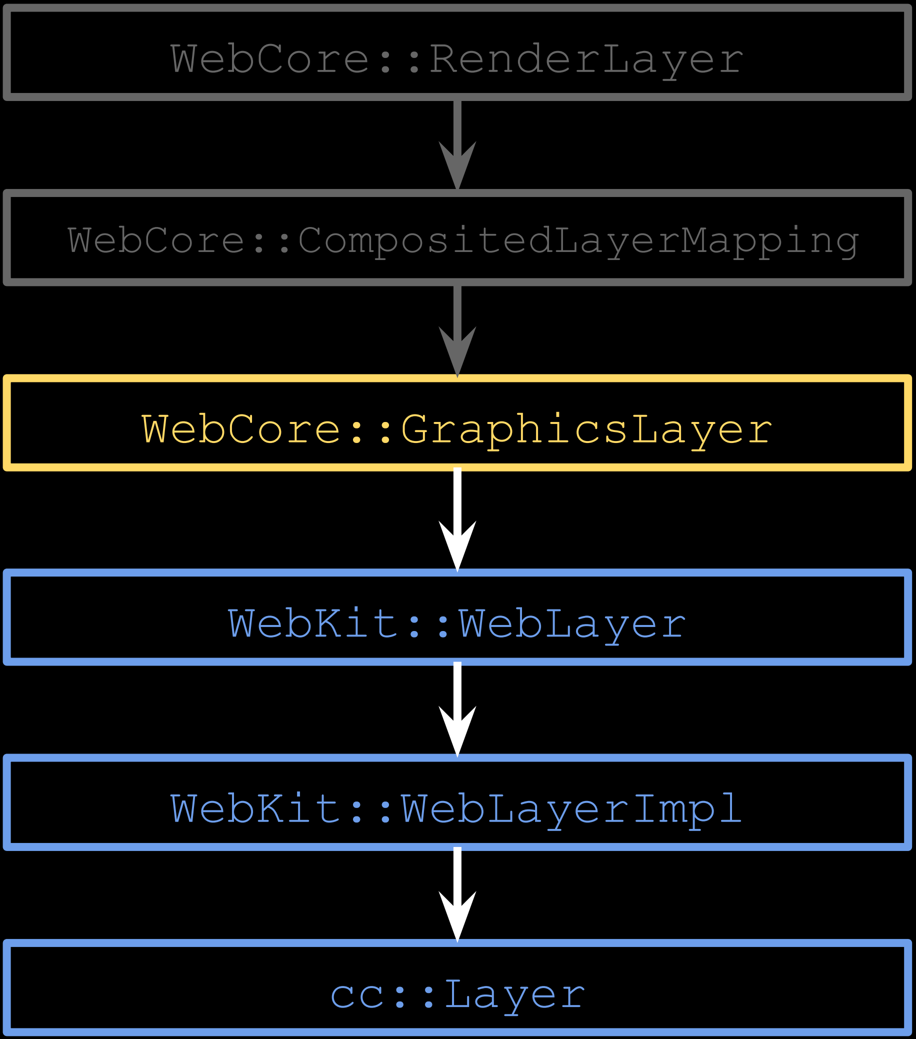
`WebLayer` directly maps to the `cc::Layer` interface.



Data Structures

`WebCore::GraphicsLayer`

WebCore's abstract class for representing a composited layer.

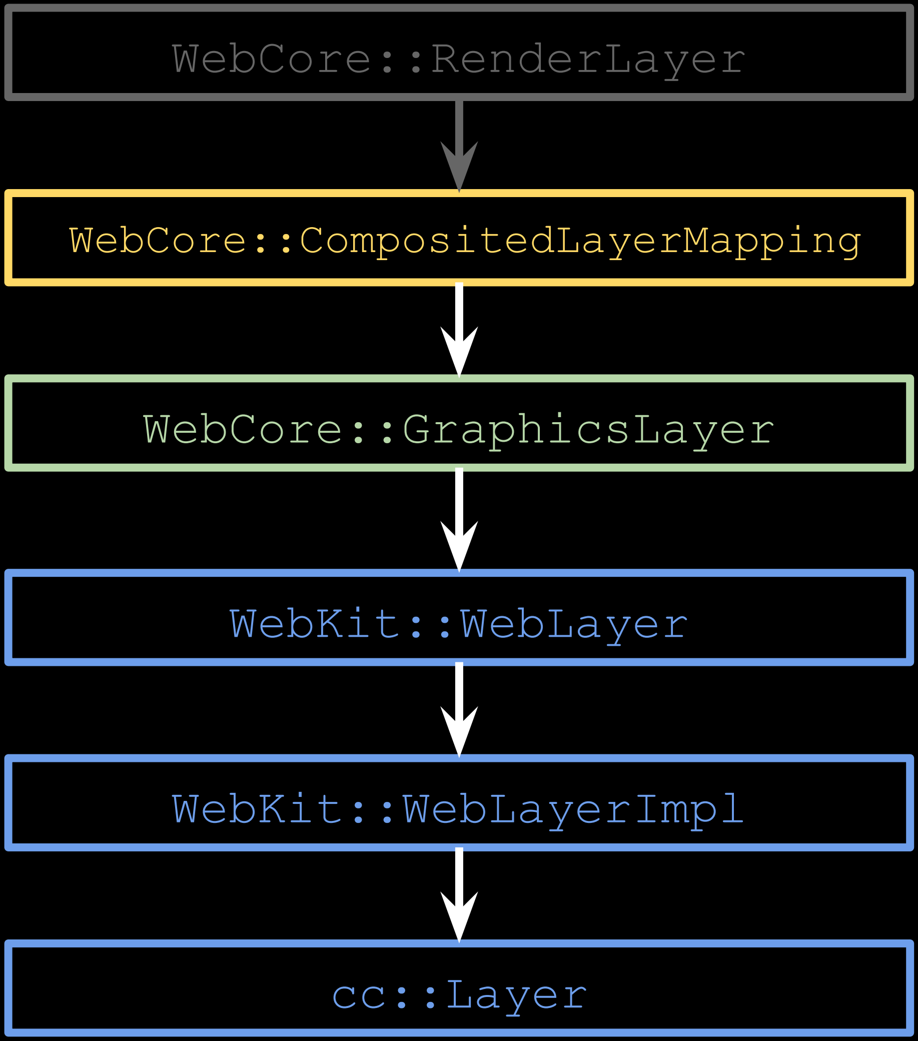


Data Structures

`WebCore::CompositedLayerMapping`

The bridge between the RenderLayer tree and the GraphicsLayer tree.

Manages a local cluster of `GraphicsLayers`, and defines how `RenderLayers` map to `GraphicsLayers`.

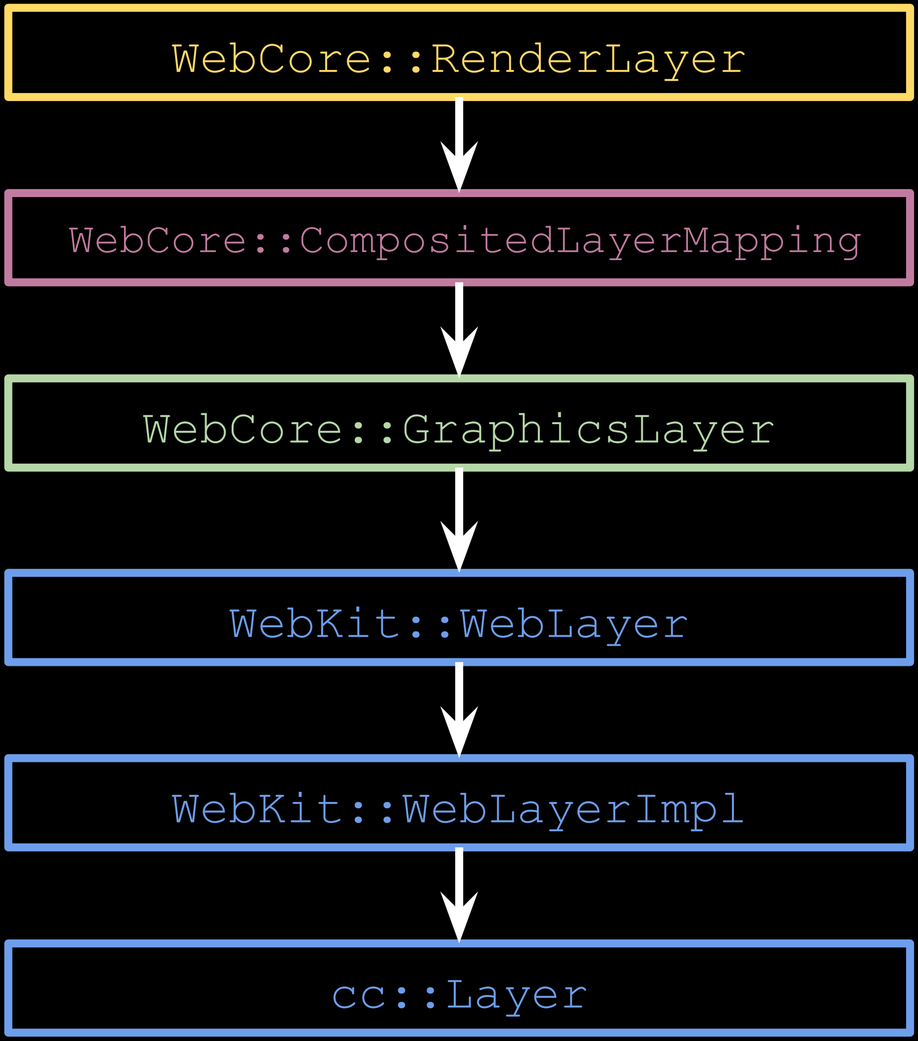


Data Structures

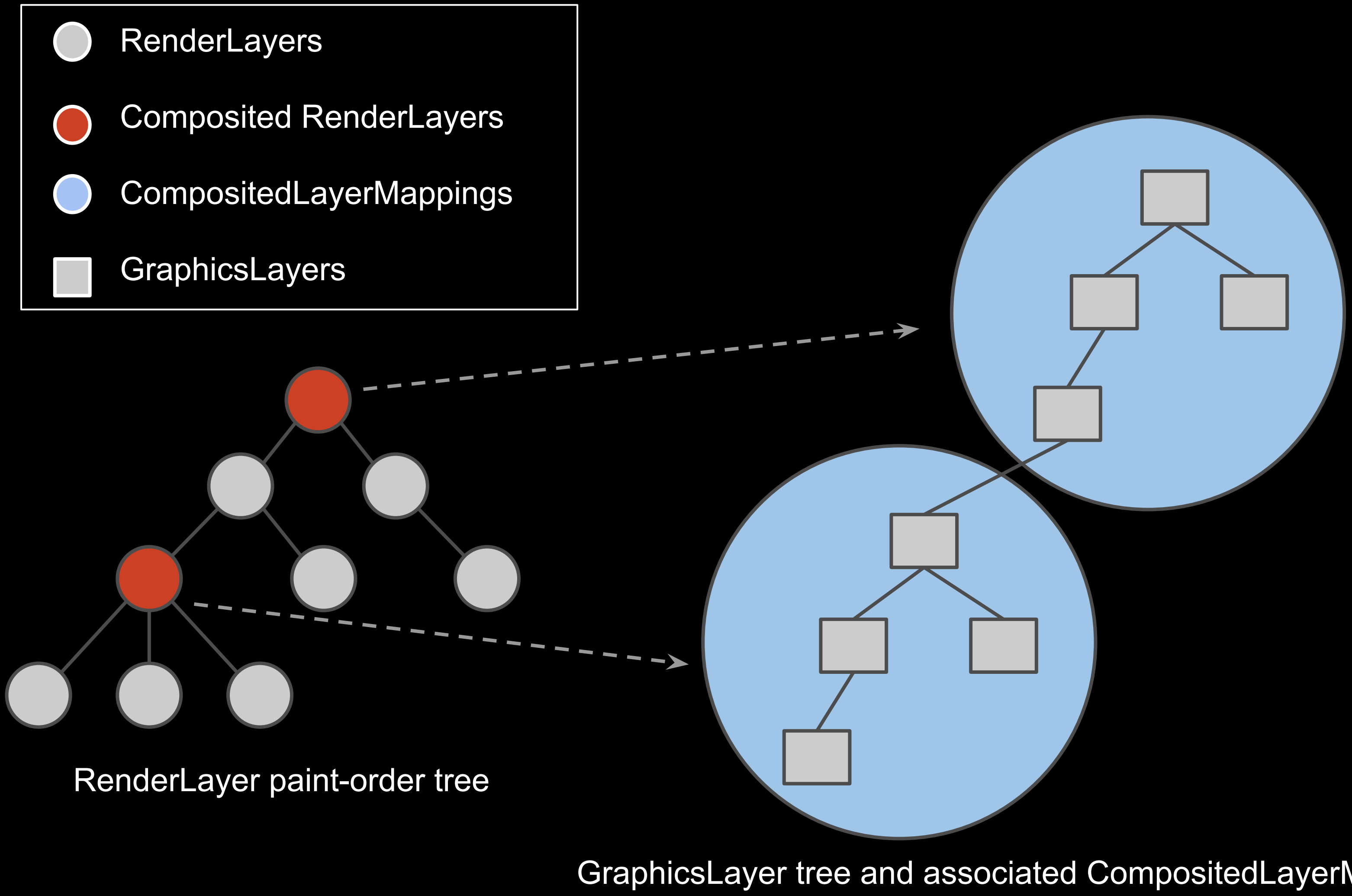
`WebCore::RenderLayer`

From compositor's perspective, this is the class that paints into backing stores when requested.

Some `RenderLayers` own a `CompositedLayerMapping`

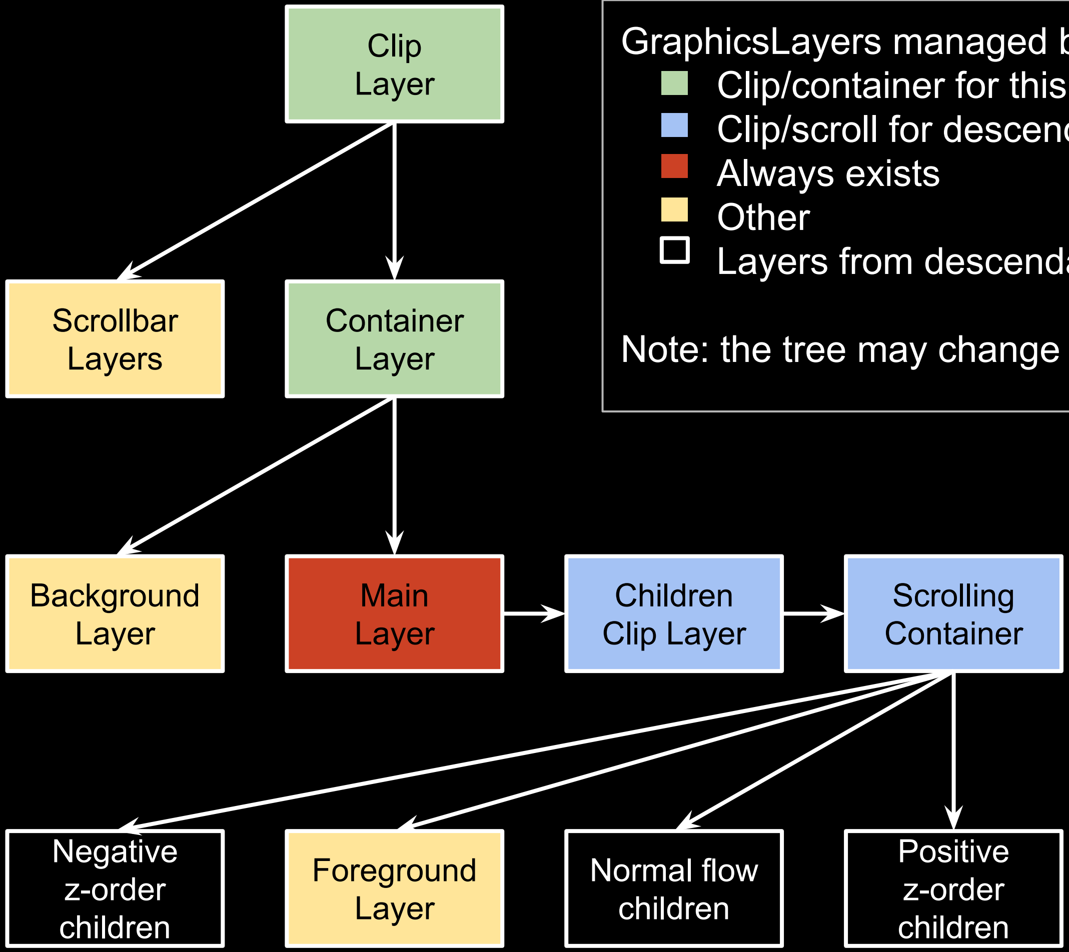


Data Structures - Layer Trees



● RenderLayers
🟥 Composited RenderLayers
🟦 CompositedLayerMappings
■ GraphicsLayers
RenderLayer paint-order tree
GraphicsLayer tree and associated CompositedLayerMappings

CompositedLayerMapping Internals



GraphicsLayers managed by CompositedLayerMapping:

- 🟩 Clip/container for this subtree
- 🟦 Clip/scroll for descendants
- 🟥 Always exists
- 🟨 Other
- □ Layers from descendant CompositedLayerMappings

Note: the tree may change depending on which layers exist.

This is the portion of the GraphicsLayer Tree that corresponds to one composited RenderLayer.
NOTE: this slide may be out of date and inaccurate.

Why does CompositedLayerMapping have so many GraphicsLayers?

- Creates layers for scrollbars to keep them separate from the content itself.
- Adds layers to clip a subtree or group siblings into a container
  - typically these layers do not draw content.
- Separates the Background and/or Foreground from the main layer, if convenient/necessary
  - Currently, background only used for background-attachment: fixed.
  - Foreground used when negative z-order children have composited content
- To group together contents that would scroll
- For mask and reflection layers (these are implicit layers in the composited tree)

Code Paths - Overview

- `GraphicsLayerChromium::paint()` - invokes painting for a particular composited layer.
    - `RenderLayer::paintLayerContents()` - recurses through the entire subtree (except descendant composited content), and paints any content that matches the "phase" of this particular composited layer.
- `RenderLayerCompositor::updateCompositingLayers()` - computes the composited layer tree given the render tree.
    - `RenderLayerCompositor::computeCompositingRequirements()` - determines which `RenderLayers` should be composited, allocates/de-allocates backings accordingly.
    - `RenderLayerCompositor::rebuildCompositingLayerTree()` - reconstructs the composited layer tree (including tree topology and composited layer properties) based on which `RenderLayers` were chosen to be composited.

Code Paths

`RenderLayer::paintLayerContents(...)`

- Code reaches here from a sequence of callbacks from `GraphicsLayerChromium::paint()`
- Receives a `GraphicsContext*` that represents the composited layer's backing store
- Receives flags that allow the entire subtree to filter what it draws
  - `GraphicsLayerPaintingPhase`

The compositor takes control of when to paint, and gives `RenderLayers` the appropriate context and paint phase.

Code Paths

`RenderLayerCompositor::computeCompositingRequirements()`

- Recursively checks children, in correct paint order, to decide if any child should become composited.
  - `RenderLayerCompositor::needsToBeComposited()`
  - `RenderLayerCompositor::requiresCompositingLayer()`
  - Maintain `CompositingState` during recursion
    - Tracks whether overlap testing is needed, or can be avoided to save computation
    - Indicates whether something in the subtree is already composited, allowing subsequent layers / ancestors to make more intelligent decisions

- Maintain an `OverlapMap` of rectangles from previously composited content
  - Anything that overlaps existing composited content (in the same compositing container) must also be composited
  - Uses a `RenderGeometryMap` to keep track of how to convert bounds into a common space for overlap testing.
  - Maintains a stack of lists. Stack is pushed when entering a container layer that doesn't need to test overlap outside its context (i.e. a composited container)

Code Paths

`RenderLayerCompositor::rebuildCompositingLayerTree()`

- Calls methods for each `CompositedLayerMapping` to configure its chunk of the composited `GraphicsLayer` tree.
  - Recall previous slide about `CompositedLayerMapping` internals.
  - `CompositedLayerMapping::updateGraphicsLayerConfiguration()` - Determines which internal `GraphicsLayers` will exist, and computes the tree topology among the existing layers.
    - Each `GraphicsLayer` is assigned a specific "phase" to paint
      - explicit phases - background, foreground, mask
      - implicit phases by having a different paint code path than the usual - scrollbars
  - `CompositedLayerMapping::updateGraphicsLayerGeometry()` - Sets various properties for each composited layer, such as bounds, position, visibility, transforms, opacity, etc.

- Children `GraphicsLayers` are collected by recursively calling `rebuildCompositingLayerTree()`
  - These children are appended to the correct `GraphicsLayer` from the ancestor `CompositedLayerMapping`.
  - This is the step that stitches together the entire composited tree
  - Scrollbars are handled here in some cases.

The Dark Side of Compositing

Compositing Philosophy

The ideal academic approach:

Optimize the tradeoff between gains of compositing versus the additional computation costs and extra memory costs.

The real implementation, currently:

Composite layers when it might be beneficial, and blindly accept the resulting overhead.

Overheads of Compositing

- Computational cost of determining how to group content into composited layers
  - Checking for overlap to ensure correctness is a significant part of the cost.
- Computational cost of managing yet another layer tree (actually, three more layer trees for Chromium)
- Increased memory cost of providing a backing-store for each composited layer that draws content

Some Compositing Issues

In theory, compositing should:

- Render exactly the same as it would without compositing
  - Reality: Antialiasing around edges, but not inside
  - Reality: LCD Text Antialiasing is not always feasible to do on composited layers
- Render at least as fast, and usually faster
  - Reality: Compositing may not help if we have to repaint everything all the time, anyway
  - Reality: Rasterization / shaders / blitting can sometimes become a bandwidth bottleneck that is more costly than painting to a single backing store in the first place

Thank you!

Questions and feedback:
graphics-dev@chromium.org