Why Composited Recordings?

We hope to teach the compositor to consume incrementally updated recordings. This is a large, boil-the-ocean project that will require significant engineering resources, so we must be confident that it is worthwhile. We believe it is, and we give our rationale below.

Problems solved by composited recordings:

1. **Flexible texturization**. Fixes the “paint inversion” problem, allowing us to take full advantage of textures and better align with other user agents.
2. **Compositor-driven texturization**. Moves texturization decisions to the compositor, permitting it to optimize for ganesh/memory.
3. **Less translation**. Connecting recording directly to compositing removes a great deal of complex, redundant and computationally-expensive code.
4. **Eliminates compositing chicken-egg issues**.
5. **Faster paint**. Can do targeted updates vs paint-everything-in-rect.
6. **Faster raster**. Larger opportunities for optimizing recordings for rasterization.
7. **More? TBD in meeting.**

Why is compositing recordings a better solution than incremental?

Many of these problems are in calcified / tricky parts of the code base, and solving them incrementally would take a great deal of effort and would in many cases incur even more technical debt. Because of these costs, many of these problems are not worth solving on their own.

Composited recordings is a way forward to erase old assumptions and fix a swath of problems at once and leave opportunity for performance gains in record and raster.

Isn’t there a simpler way?

That said, boiling the ocean is scary! Let’s outline the alternatives to problems listed above.

1. Flexible texturization (fixing the paint inversion problem)

A RenderObject requires a self-painting RenderLayer to be composited. Self-painting RenderLayers have an impact on paint order, so we must be conservative about compositing lest we invert paint order. Additionally, requiring RenderLayers causes us to behave oddly with respect to other UserAgents, such as requiring stacking contexts for fixed position elements so that we can make them composited.

Composited recordings solution: make compositing decisions from recordings; a better recording data structure could be abstracted from RenderObject/RenderLayer and can always be put in the right order.

Alternatives:

1. Create RenderLayers for every RenderObject. Pay for more work in RenderLayerCompositor and overlapping. Quite a bit of work too.
2. Lazy RenderLayer creation in compositing. Lots of work and potential bugs. In edge cases, could devolve into #1.
3. Don't fix this problem. This problem has been around for a long time, maybe it doesn't have to be fixed?

2. Compositor-driven texturization (OOM problems, layer performance)

Currently, Blink is the arbiter of texturization / what gets a layer and cc uses that as input. Because of catastrophic graphics drivers issues when using too much texture memory, cc tries to cut back on texture usage. At some point, there can be too many layers on screen using too many textures and content just disappears, leading to bad web experiences. Layers in general add CPU overhead which is unnecessary if raster performance is fast and caching is not necessary. (This is why squashing was embarked upon.) Additionally, Ganesh pays performance overhead for layers (framebuffer switching).

Composited recordings solution: move compositing decisions out of Blink into compositor, be able to create as few or as many layers as possible

Alternatives:

1. Teach Blink about whether Ganesh is on or not. Teach RenderLayerCompositor about how to record 3D transforms, canvases, videos into GraphicsContext so that all reasons for compositing become hints and can be squashed together if Ganesh is on.
2. Teach cc about squashing. (Essentially #1, but in the compositor instead.) In this world, Blink creates layers only for cc to uncreate them.
3. Teach Blink about memory limits. This is a lot of work. It's also ambiguous how layers map to memory usage, so this would be very handwavy. This would also require #1 in order to make it even possible to reduce memory costs.
4. Rewrite cc data structures to be more efficient with large numbers of layers. This is non-trivial as some of cc's most complicated code (LayerTreeHostCommon) is involved here.
5. Improve squashing by refactoring RenderLayerCompositor/CompositedLayerMapping so that it can squash in more circumstances. The current CompositedLayerMapping-owned-by-RenderLayer means that there are RenderLayer tree hierarchy limitations on what can be squashed into what even though final paint order is linear.

3. Less translation (reduce redundant geometry calculations)

Currently, Blink, cc, and SkPicture all have independent notions of where content is in the space of the page. Blink must calculate absolute positions for overlap in RenderLayerCompositor. cc must calculate positions to know which subrectangle of each layer is visible. SkPictures don't have any geometry information and must be traversed to figure out what images intersect which rectangles and need to be decoded, even though Blink also has this information.

Composited recordings solution: bake geometry/visibility information into recordings so it can be used both by Blink and cc

Alternatives:

1. Teach Blink about viewporting. Teach RenderLayerCompositor about what parts of layers are visible. Remove cc main thread CalcDrawProperties.
2. Add an API for Blink to inform cc about where images are rather than traversing SkPictures.
3. Add geometry information to SkPicture so that it can more quickly answer questions about what images intersect which rectangles. This would be something like moving from TileGrid to something better.

4. Eliminate compositing chicken-egg issues

https://code.google.com/p/chromium/issues/detail?id=282720

Composited recordings solution: if Blink doesn't know about compositing, then there can't be chicken-egg issues by definition (the nuclear option)

Alternatives:

1. Try to remove these issues incrementally, although these have been historically very tricky to fix.

5. Faster paint (invalidation rect friction)

Blink provides "invalidated rects" in the space of GraphicsLayers to cc. cc calls back and paints/records everything in that rect into a new SkPicture. Blink doesn't have a spatial bounding hierarchy, so record costs involve a lot of unnecessary walking of RenderObjects that aren't visible. This causes record costs to be proportional to the size of the RenderObject tree in the page rather than the size of the invalidation. Finally, the new SkPicture can't be merged with old SkPictures and so cc has to maintain a grid of inflated SkPictures to try to trade-off number of SkPictures required for rasterization with record costs.

Composited recordings solution: targeted updates of recordings, invalidations live at the render object level, compositor retargets them to the correct backing that it creates

Alternatives:

1. Make SkPicture mutable. Add more semantic information about invalidations that are just mutations so that pictures can be updated incrementally.
2. Hierarchical SkPictures in Blink. Maintain cached SkPictures hanging off the render tree so that rerecording doesn't need to happen for subtrees that haven't changed.
3. Add bounding box hierarchy to Blink to improve RenderObject tree walk.

. Faster raster

Blink is not in a good position to make good low level raster decisions. Because of impl-side painting, the final scale of content is not known at record time. There are some optimizations that can only be done at raster time where final scale is known, such as fixing background color bleed in rounded rectangles. Additionally, SkCanvas commands may be too low level; they may make sense for a software world but are expensive with Ganesh, e.g. rounded rect borders that were previously done with a clipPath before https://codereview.chromium.org/174243003/.

Composited recordings solution: move raster decisions later in the pipeline where these decisions that can made fully knowing the final scale and the choice of raster backend

Alternatives:

1. Continue adding higher level APIs to SkCanvas/GraphicsContext. drawRect is one step towards this, but something higher level like drawBorder might be even better.
2. Plumb a flag about when Ganesh is being used to Blink so that it can attempt to make better decisions about what to paint. Increases paint complexity in Blink.