target using ganesh mode. In such a modeあらわな

Pro:

- Transparent LCD text where you can't actually bake the text to a texture
- For layers that change every frame, drawing to texture, then drawing the texture to screen is 2 passes. Drawing to screen is 1 pass. If the layer is big, this win can be significant.

Con:

- Sizing of the render target becomes tricky
- Difficult interop story with Ubercomp
    - Each render target produced by the render thread in übercomp mode needs to allocate a texture for the render target and put it in a mailbox, and push it across on some part of the FrameData.

Sticky Triggering

For all of these triggering options, an interesting possibility is to make triggering sticky: once you become gameshed, you would never go back to software rasterization.

Cases of interest:

- Layer starts off simple and becomes complex over time, or the reverse
- Static for a long time then starts animating

Other Triggering Options

In the phase 1 document, we picked a few explicit triggering modes to try out initially.

Part of this project is to figure out which layers should render with ganesh, and which should continue rendering with software.

1. **Implicit ng:** One option is to use heuristics on the layer's state, so that content authors don't have to do anything to get Ganesh enabled. This is simple, but it can be confusing to developers: code adlds more text to a div as it's moving, and it suddenly stops Ganeshing, and becomes unexpectedly janky, for instance.

2. **Explicit triggering:** Another option is use the presence of a specific style or flag on content, added by authors, to deterministically trigger rasterization using Ganesh. Since content creators are more aware of which layers would benefit from accelerated rendering, some think this idea merits consideration.

These don't have to be mutually exclusive. Ultimately this boils down to what the return value of PictureLayerImpl::PickDrawingApproach() should be. It returns one of the three modes, {SwMode, TiledGaneshMode, DirectGaneshMode}.

Implicit triggering options

- Op count on picture < 30; once we go Ganesh
- Invalidation history shows high invalidation rate, made feasible by our ongoing work on flattening the picture pile; see https://codereview.chromium.org/23698016/ and crbug.com/312861.
- Presence of transparent LCD text
- Mask layers are obvious candidates for acceleration
- Layers that are running unaccelerated animations

The nice property of these solutions is that they're very web-compatible. But, through accelerated canvases larger than 256x256, 3D CSS making layers, etc, we know that this dspace raises a ton of painful support problems. We also know it makes testing

- -blink-coherence: default | may-move-scale-or-fade | contents-may-change | static
-
    - good for prototyping, virtually impossible to prototype
    - WebLayer::debugName() will give you the ID of a div, we might be able to use strstr(layer::client_->debugName(), "ganeshable") for a quick hack
- ~~backface-visibility: visible;~~ // Many concerns have been raised.

Command Buffers

Explore the use of recorded GPU command buffers in lieu of cached SkPictures (ideas from epenner@ and danakj@). This could potentially result in less conversion overhead (SkPicture → command buffer → driver GL). The reason this isn't part of the initial effort is that we're not yet sure that this conversion overhead is more than noise, so we would

need to make sure it was carefully measured before we started this (substantial) rewrite.

Direct Rendering

Direct rendering requires our render targets to have stencil. We have this set up to work for Linux, OSX. But we do not have stencil on Android or Windows.

Uber Compositor

A number of implementations for ganesh+uber compositor came up.

1. Always bake to tiles ← we chose this for v1
    - Pro:
        - Simple.
        - May actually be a win over sw rast for many cases: 0 copy upload!
    - Cons:
        - Extra GPU work on animation-heavy workloads
        - May need to replay the same SkPicture on many tiles
            - This can be mitigated somewhat by using scratch-pads
                - One big scratch pad to skip per-tile costs in painted anims
                - ‘Row’/’Col’ scratch pad to amortize per-tile costs
                    - Ideal size for <16ms GPU time
                    - Followed by copy into row/col tiles
        - From epenner: One annoying quirk might be mixing uploads and FBO rendering into tiles. Internally this can trigger changes in texture format which can bring the GPU thread to a halt when this occurs. This may be avoidable these days or less of a penalty than it once was, but something to keep in mind. The type of SW uploads we do can also influence the texture format and make this better.

          We could also just segment tiles by format+usage-type. Slightly worse memory consumption but that would also solve this problem.

2. Sending SkPictures for direct-draw up to browser process, running Ganesh there
    - Pro:
        - Maximum GPU efficiency on cases that don’t benefit from baking
        - Easy for browser CC to apply transformations
    - Cons:
        - Additional attack surface and crash source in browser process

        - Need code to serialize SkPictures
            - Rather than serialize SkPicture, I'd dump it into another type of format. The browser compositor just needs to draw the thing with Ganesh. It doesn't need to do things like re-record portions of it, or deal with pixel refs or etc etc etc, right? So if I was to do this I'd consider dumping the SkPIcture thru something that can generate either a GL command buffer stream or a Ganesh stream of some sort that's less problematic maybe possibly for attack.

            - Discussing this with nduca he pointed out some interesting stuff since GL is stateful that would not make this too trivial to secure, but I'm also not sure it can't be done if it was useful enough..
        - Need a mechanism for sharing image data (mailboxes?)
        - Extra CPU work to ship SkPictures

3. Somehow run Ganesh directly in GPU process
    - Pro:
        - Save CPU by binding ganesh direct to GL driver, e.g. D3D, Android GLES
        - Save GPU on cases that don’t benefit from baking
    - Cons:
        - Still need to maintain all the GL command-buffer code for WebGL etc
        - Need to define new sharing/memory management model
        - Need to define way for parent compositor to apply transforms
        - Far away from our current architecture, a lot of work/subproblems

4. Allocate a resource in the render process for the current render target. Render-process side, direct-render using ganesha into that render target. Send the render target to the browser process using a mailbox.
    - Pro:
        - Plays well with hw overlays.
        - Nice properties wrt LCD text
    - Con:
        - Incompatible with subtrees with OOPIF/Site Isolation layers
            - Could switch dynamically between #1 and #4 based on the presence of isolation.
        - Lose the GPU efficiency benefits of uber comp (may need to resort to modal switching with #1 to preserve it in some cases)
        - May not play well with non-main render targets that change size frequently
        - What if GPU time for this work is >16ms? In scrolling case we may want #1 here to spread out the cost over several frames.