Partial Texture Updates with Uber+Ganesh

danakj, enne, piman

Tranking bug: crbug.com/419394

Problem

Currently every texture shipped to the browser from the renderer is required to be at least double buffered. This is because the browser takes a reference on the textures, and may want to produce frames from the textures given to it while the renderer is preparing its next frame.

This means that heavy invalidations require up to twice as much GPU memory as no invalidations.

This also means that we must redraw the entire tile every time a pixel in it changes. The amount of pain this causes scales somewhat with tile size, and ganesh uses large tiles (maybe 4 per viewport, or even less). This makes us allocate and re-raster a huge part of the viewport for a blinking cursor.

Solution

We will allow returning of resources from the browser to the renderer along with the BeginFrame message. The renderer will be able to update its existing resources, since they will no longer be in use by the browser, and return them back to the browser before its deadline.

Dependencies

- Browser-driven deadline scheduling - This depends on the renderer's scheduler being driven by a BeginFrame message from the browser compositor, so that the renderer's frame generation happens after the resources are returned, and before the browser needs to draw.

Details

Here's a timeline of how a frame would look.

Scenario A: All's good in the hood.

Browser Compositor/Display	**Renderer Compositor**
BeginFrame Insert SyncPoint A Collect resources from last renderer frame and ship with BeginFrame IPC. ==========>	
	Receive BeginFrame Start main thread frame ……. Finish main thread frame (3**) Start compositor thread frame Wait on SyncPoint A (1*) Accept released resources Update resources in place (4**) Insert SyncPoint B Send CompositorFrame IPC <==========
Wait on SyncPoint B (2*) Draw and Swap frame	

1* The SyncPoint A ensures the browser is done using the resources when the renderer changes them.
2* The SyncPoint B ensures the renderer resources are ready when the browser uses them
3** The main thread may not finish its frame in time. This is covered by Scenario B.
4** The compositor thread could take arbitrarily long due to bugs or malicious intent. This is covered by Scenario C.

Scenario B: The main thread frame takes too long.

Browser Compositor/Display	**Renderer Compositor**
BeginFrame Insert SyncPoint A Collect resources from last renderer frame and ship with BeginFrame IPC. ==========>	

	Receive BeginFrame Start main thread frame ……. Compositor thread gives up Send CompositorFrame IPC sending all the resources given from the browser back again. (1*) <==============================
Draw and Swap frame	
	.... Finish main thread frame ....?
BeginFrame Insert SyncPoint A Collect resources from last renderer frame and ship with BeginFrame IPC. ==============================>	
	Start compositor thread frame Create new resources to double buffer Insert SyncPoint B Send CompositorFrame IPC <==============================
Wait on SyncPoint B Draw and Swap frame	

1* When the renderer gives up before the deadline, it sends the resources back to the browser. This leaves them in use, so if the renderer wants to produce a new frame on its own, it would require double buffering unless a new BeginFrame came from the browser.

Scenario C: Oh noes.

Browser Compositor/Display	**Renderer Compositor**
BeginFrame Insert SyncPoint A Collect resources from last renderer frame and ship with BeginFrame IPC. ==============================>	
	Receive BeginFrame Main thread frame Start compositor thread frame Wait on SyncPoint A Accept released resources

.... Time out occurs (suggest 0.5s) (1*) Draw and Swap frame (2*) Put renderer in **the penalty box**	Update resources in place .....

1* We suggest that the timeout is not too long that it completely destroys interactivity with the UI for the user, but is long enough if the renderer is just missing its deadline and hits the next deadline. So it should be long enough to not be racey with the next deadline. So a timeout of >= 3 deadlines seems reasonable, which is 50ms (for 60hz). We arbitrarily chose 500ms instead to make this race very very hard to hit by accident, especially during startup when the compositor thread may not be responsive, but it could be much less with experimentation/data collection.

This timeout is meant to tell a malicious or badly broken renderer and prevent it from blocking the UI forever. It's not meant to happen under normal circumstances. If thread descheduling is a problem then a longer timeout would also work well, as the UI thread should be equally unable to make progress as the renderer compositor thread.

2* Some choices of what to end up drawing in the browser in this situation: draw with whatever resources are currently available (risk of a non-atomic frame update), draw with a solid color, and/or kill the renderer.

Scenario D: The penalty Box

Once the renderer has misbehaved, it no longer gets to participate in this mechanism and must fall back to always double buffering. This is achieved by no longer sending resources to the renderer with the BeginFrame message, causing any resources to remain in use by the browser. This is the default behaviour today.

Browser Compositor/Display	**Renderer Compositor**
BeginFrame ==========>	Receive BeginFrame Start main thread frame ……. Finish main thread frame Start compositor thread frame Update resources double buffered Insert SyncPoint B Send CompositorFrame IPC

Wait on SyncPoint B (2*) Draw and Swap frame	<==========

Questions

In Scenario B, the renderer main thread misses its deadline, so the compositor thread just gives the resources back to the browser. We could have the renderer hold onto them longer, but this raises questions/points:

- The browser has already derefed them, so we need to tell it that it can still use them. This is the same as giving them back, almost.
- Once the renderer is ready to produce a frame, would it tell the browser that it's going to use the resources now? It would require a round trip to not be racey, which is almost the same as asking for a new BeginFrame.

Why only Ganesh? Why not software compositor (or software raster)?

In software compositing/raster, we have a pending tree in the compositor, since we expect updates to take more than one deadline to complete. We could guess when it will be less than that and try skip the pending tree, but its much less common than the ganesh case. So we conclude at this time it is not worth the complexity.

Worst-case scenario?

When there are multiple visible windows submitting frames and animating, they will effectively be rate-limited to the min(frame rate of all renderers).

Any time a renderer can't hit 60 fps with ganesh raster, we should consider it a bug and fix it. But while such bugs exist, one renderer can cause another's animations to become janky. One the one hand, this is what you get when you allow each renderer to draw directly to the backbuffer, or approach solutions such as that - which is what this is. On the other, maybe we will want some fancy penalty box heuristics to drop renderers out of the fast path while such bugs may exist (I'm not currently convinced that we do).