cc::Surfaces for Video

(enne@, lethalantidote@, June 14th, 2017)

Overview

This document proposes a way to send video updates to the display compositor via surfaces instead of through heavy renderer compositor updates. This will allow the media system to submit video updates asynchronously with respect to the renderer compositor commit.

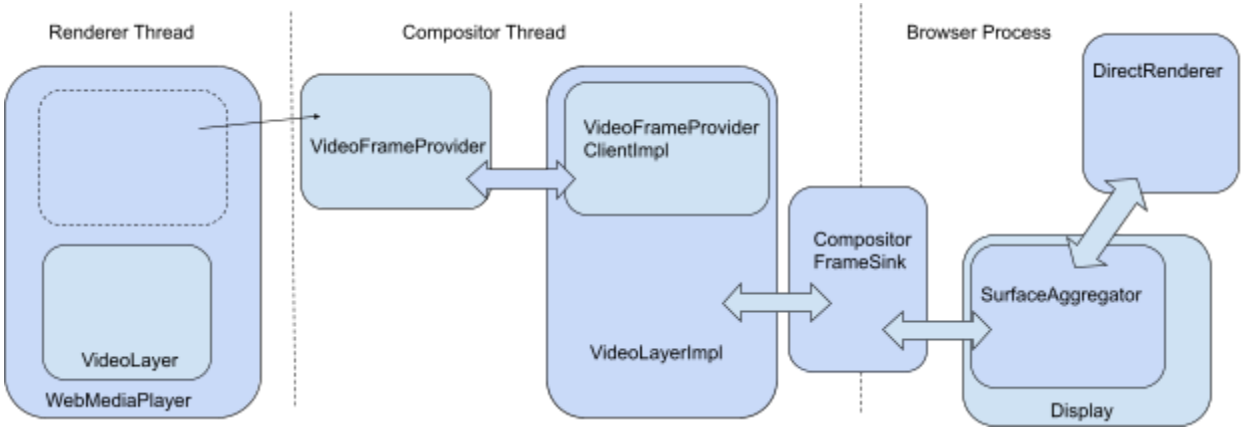
Goals

1. Enable the feature of having picture in picture video in the browser.
2. Improve power usage by not going through a full compositor update to update video.

Related docs / Background Reading

- Surfaces overview
    - Also, Surface Aggregation [internal video]
    - Also, Compositor Stack With Surfaces
- Video Playback and the Compositor
- Compositor Thread Architecture
- There's no offscreen canvas doc overview, but there is one for resizing :C

A brief stroll through the current video compositing path



Renderer process

WebMediaPlayer creates a VideoLayer with a VideoFrameProvider on the renderer main thread. This provider is passed along to a VideoLayerImpl on the compositor thread and wrapped in a VideoFrameProviderClientImpl. When the video layer goes away, the next compositor commit will generate a tree without the layer, and this will eventually destroy the VideoLayerImpl. The VideoFrameProviderClientImpl is stopped directly before the VideoLayer goes away.

VideoFrameProviderClientImpl is a bridge between the VideoFrameProvider and the layer interface. It handles locks, informing the layer when there's a new frame, and making sure the layer gets redrawn when necessary.

VideoLayerImpl uses VideoResourceUpdater, which turns a media::VideoFrame into resources that the compositor can consume. There's a number of resource conversion cases. Software video -> gpu compositor involves an upload. Gpu video -> gpu compositor wraps the video resource in a mailbox (with an optional copy). Software video -> software compositor copies a bitmap. Gpu video -> software compositor involves reading back from the gpu. There's also yuv -> bitmap conversion logic. Any uploads and copies are done using the compositor context.

In the end, VideoLayerImpl::WillDraw generates the resources needed for the frame and VideoLayerImpl::AppendQuads generates the quads for the video itself, including rotation and stretching calculations. VideoLayerImpl produces TextureDrawQuads (for software and gpu resources), StreamDrawQuads (for streaming texture resources), and YUVDrawQuads

Browser process

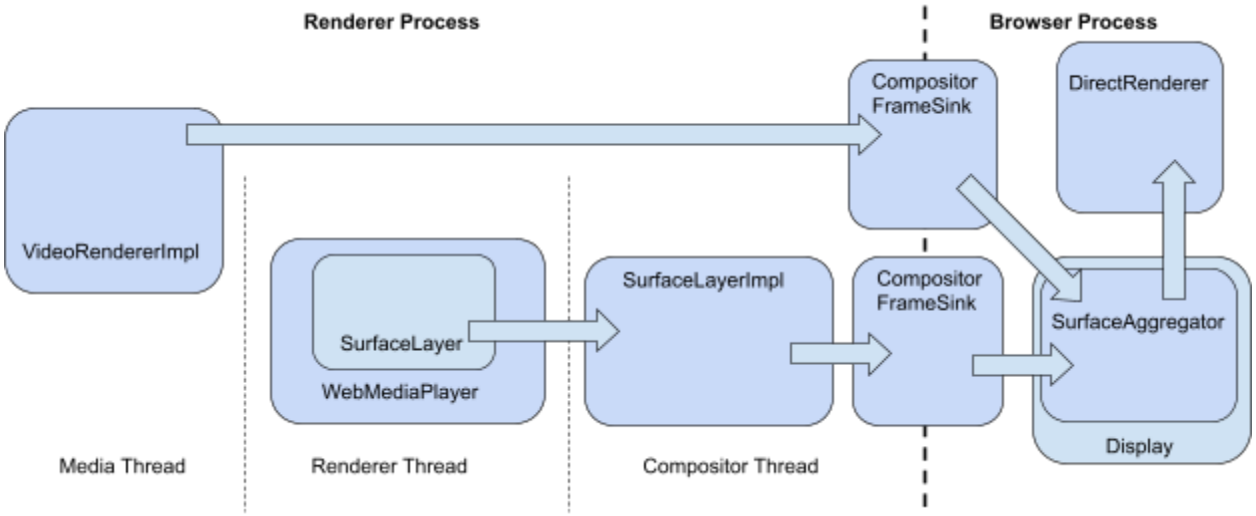
This compositor frame is sent to the browser via a CompositorFrameSink. Eventually, this frame ends up being aggregated in SurfaceAggregator as a part of the DisplayCompositor to produce the final frame to put on screen. This aggregated compositor frame is handed to the DirectRenderer (which can be either a GLRenderer or a SoftwareRenderer).

DirectRenderer::DrawFrame has the opportunity to analyze the frame and figure out if some quads should be promoted to overlays before drawing it.

Rough implementation details

High level overview

1. Add a browser-side provider for creating the mojo pipes for creating surfaces and frame sinks that the renderer can use to deliver CompositorFrames to the FrameSinkManager.
2. Renderer process main thread media system generates SurfaceLayers instead of VideoLayers. The SurfaceLayer will not need any knowledge of a VideoFrame; it will simply embed whatever compositor output we send to it.
3. Instead of depending on the compositor to upload videos, the media system needs to use its own ContextProvider/ContextGL/Context3D to upload/copy textures as needed.
4. Media system submits CompositorFrames containing the equivalent DrawQuads that VideoLayerImpl would have produced, on the media thread. This layer will not know the final display size of the video, nor anything else that comes from blink, unlike VideoLayerImpl.
5. Remove VideoLayer, VideoLayerImpl, video tests from cc/.



(1) Browser-side provider

The browser side of the implementation will look a lot like OffscreenCanvasProviderImpl. This provider is owned by the RenderProcessHostImpl, which is the bridge between the browser process and one renderer process.

I think the best option is to repurpose OffscreenCanvasProviderImpl and OffscreenCanvasSurfaceImpl directly. Their purpose is really to make it possible for the renderer to create surfaces and submit CompositorFrames. Probably there needs to be some bikeshedding about renaming it something like EmbeddedFrameSinkProvider or some such, but this is less important.

OffscreenCanvasProviderImpl explainer

For completeness, OffscreenCanvasProviderImpl provides two interfaces to the renderer via mojo bindings. One is CreateOffscreenCanvasSurface and the other is CreateCompositorFrameSink.

CreateCompositorFrameSink can only be called once and creates a CompositorFrameSink that the renderer can use to submit CompositorFrames. For offscreen canvas, the browser side of this hookup is done in OffscreenCanvasSurfaceImpl::CreateCompositorFrameSink and the renderer side that uses the created CompositorFrameSink to submit CompositorFrames is done in OffscreenCanvasFrameDispatcher.

CreateOffscreenCanvasSurface provides an interface for the renderer to add surface references and sequences. In general, I don't think video needs anything special here and can crib from the offscreen canvas implementation with respect to surface references. This is used in the renderer in CanvasSurfaceLayerBridge.

In order to interact with the FrameSinkManager, CreateOffscreenCanvasSurface also creates OffscreenCanvasSurfaceImpl objects in a map. These objects represent the lifetime of the communication, register the surface id with the FrameSinkManager, and pass along various callbacks.

All of these things are things that offscreen canvas and video will use.

(2) Generate SurfaceLayers in WebMediaPlayer / id plumbing

WebMediaPlayer currently creates a VideoLayer with a VideoFrameProvider. The change here is to make it create SurfaceLayers instead. In offscreen canvas land, this work is done by CanvasSurfaceLayerBridge. This could be done by the WebMediaPlayer directly or in some helper class. SurfaceLayers have a SetPrimarySurfaceInfo function on them that sets SurfaceInfo for the surface that it is embedding. The SurfaceInfo to use comes from submitting a compositor frame to the CompositorFrameSink and receiving it via the OnSurfaceCreated callback.

There will also need to be some equivalent to OffscreenCanvasFrameDispatcher. Let's call this new class VideoFrameDispatcher. VideoFrameDispatcher will live on the media thread. It will create and own the CompositorFrameSink. It will also generate local surface ids (which are half of SurfaceId, with the other half being the frame sink id).

Resize: why do videos need to resize anyway?

Videos have an intrinsic size and then get stretched to fit the bounds of their embedding container. It seems a little unnecessary to even have a complicated resize path when resizing the SurfaceLayer containing the video doesn't really have any effect on the video decode process or resources itself.

The stretching work could be done at aggregation time and the video frame submitting code can be agnostic about the size of the video on the page.

SurfaceLayer itself has a setting called SetStretchContentsToFillBounds that stretches anything that it has embedded. Setting this for any SurfaceLayer that's embedding video should make it possible to not have to inform the code submitting compositor frames for video about what size they should be at. The SurfaceAggregator will figure out the correct scale when aggregating frames and the display compositor will scale the video textures when drawing.

(3) Upload textures using media context

The summary of this task is to move the work that cc::VideoResourceUpdater does on the compositor thread using the compositor context to the media thread using the media offscreen

context. VideoResourceUpdater takes a media::VideoFrame and turns it into cc::VideoFrameExternalResources which is then consumed by cc::VideoLayerImpl.

The media system in the renderer uses a separate offscreen context for gpu work. This offscreen context is created by RenderTheadImpl and stored in a media::GpuVideoAcceleratorFactories object. This is used by GpuMemoryBufferVideoFramePool and VideoOverlayFactory to interact with command buffers.

Currently, the compositor context is used by VideoResourceUpdater to do various gpu work. It seems best to have the media system do this directly on the media thread whenever a video frame is ready via its own offscreen context.

The biggest issue in moving this work is that the compositor has a cc::ResourceProvider that it uses to manage resources and this is a level of indirection that is likely not necessary for the media system. For an initial prototype, it might be worth creating one just for the code reuse out of VideoResourceUpdater.

A cc::ResourceProvider is mostly just a wrapper around a context provider / gpu memory buffer provider / shared bitmap manager with some helper functions. It also has helpers for updating and waiting on sync tokens. Offscreen canvas just implements its own sync token handling locally. It's not that much code, so it's not clear to me whether using cc::ResourceProvider is worth introducing a potentially needless abstraction. This code is a lot of what OffscreenCanvasFrameDispatcherImpl is doing.

Sync tokens

One minor difference about using the media offscreen context instead of having VideoResourceUpdater run on the compositor context is that some code using copied gl textures now needs sync tokens for that copy.

(4) Generate compositor frame

The summary of this task is to move the work that cc::VideoLayerImpl does to generate a CompositorFrame from VideoFrameExternalResources onto the media thread. This frame can then be submitted to the CompositorFrameSink.

This is fairly straightforward. Once resources have been uploaded or copied as necessary, then a CompositorFrame should be generated and then SubmitCompositorFrame on CompositorFrameSink should be called.

Once the display compositor is no longer using resources from a CompositorFrame, it will return them to the CompositorFrameSinkClient via ReclaimResources. These resources will need to be

processed (see sync token section) and possibly returned to any resource pools that the media system is using.

Rate limiting

CompositorFrameSinkClients (e.g. OffscreenCanvasFrameDispatcherImpl) are ticked by the display compositor via OnBeginFrame and are notified that it's time to submit another frame. Asynchronously, once that frame has been drawn the display compositor will send that client a CompositorFrameAck. Offscreen canvas supports two pending compositor frames, after which it starts dropping OnBeginFrame messages. It seems probable that video will want to behave similarly.

(5) Remove VideoLayer entirely

VideoLayer should be fairly simple to remove. cc has pixel tests, but they all test yuv draw quads and such and not VideoLayer directly. video_layer_impl_unittest.cc should probably move to the content/renderer/media to test the work of generating quads. The rest of the cc unittests and the VideoLayer and VideoLayerImpl code can probably just be removed directly after that.

Appendix

Updated cc::Surface for Video Diagram