OOP-R on Canvas2D

8 September 2019

khushalsagar@

OVERVIEW

Out of process raster (OOP-R) is an architectural change aimed at enabling the use of multiple graphic backends (Vulkan/Metal/D3D12) for rasterization in Chromium, particularly targeted at rasterization currently done using skia in client processes (see The GPU process), which is done by 2 clients : compositor (cc) and canvas2d. As of today, the compositor has been transitioned to use OOP raster[1]. This document is to summarize the remaining work for transitioning canvas2d to use OOP raster.

Prior to OOP raster, rasterization using Skia is performed using a GrContext provided by viz::RasterContextProvider. Under the hood, we set up the GL bindings for this GrContext to a cross-process GPU command buffer implementation. Instead of issuing GL commands issued by skia in the client process directly to the driver, the command buffer serializes them for execution in the service process (see GPU Process/Command Buffer).

With OOP raster, the serialization of GL commands is replaced with paint commands recorded in a PaintOpBuffer. These are higher level commands which can be executed on an SkCanvas that may be backed by any GPU backend (or software) implementation.

DESIGN DETAILS

Client Context APIs

The viz::RasterContextProvider encapsulates multiple APIs to provide access to different functionalities on the client side. The 3 high level interfaces that we need to understand here are:

1) GLES2Interface

   Allows direct access to GL APIs sent to the GPU service via GL command buffer. This is currently used by multiple components in the renderer but the eventual goal is to limit this

[1] Still being rolled out to users via finch on some platforms.

to WebGL, since by design this requires the context to support GL. All other use-cases should go through the RasterInterface below.

2) GrContext

   Allows use of skia on the client side for GPU rasterization, by binding it with the ContextProvider's GL command buffer.

3) RasterInterface

   Provides higher level APIs which can be mapped to any graphics API backend on the service side. The APIs relevant for rasterization using skia are BeginRaster/Raster/EndRasterCHROMIUM, used to serialize and send a PaintOpBuffer to the GPU service. However, these APIs are not supported on RasterImplementationGLES (used with GL decoder).

Each of these functionalities, which need to be supported for a context, should be specified at creation time (using ContextCreationAttribs). The important part here is that supporting OOP raster and GLInterface/GrContext for a RasterContextProvider is mutually exclusive. This is because supporting OOP raster requires the use of a RasterCommandBuffer/Decoder, which does not provide a GL API. This is a design choice to make execution of commands from the client, interleaved with work using skia, simpler and also because there should be no use-case where both these functionalities are required together.

Blink Context Providers

The renderer process has multiple ContextProviders, each backed by its own command buffer/command stream. For the work pertaining to canvas2d, the ContextProvider usage we need to understand/modify here is the one relevant for blink and renderer's main thread. The APIs detailed above are exposed to blink using the WebGraphicsContext3DProvider interface, which internally wraps a ContextProviderCommandBuffer. The important part here is that there is a single instance of this context per thread (other than WebGL). This means that for the main thread, the context used by canvas2d is also shared with the media/video stack (see SharedMainThreadContextProvider).

HIGH LEVEL PLAN

At a high level, the work involved with transitioning canvas2d to use OOP raster can be divided into 2 broad parts:

Eliminate GLInterface dependencies

Since supporting OOP raster for a context implies that it can not be used for GL execution, a prerequisite for this is to remove all GL dependencies from contexts used by canvas2d. And since canvas2d uses the SharedGpuContext, which is a per thread instance, this effectively means eliminating all GL access on this shared context (basically everything other than WebGL?). The details for this are a bit unknown at this point, and would need to be addressed on a case by case basis. My take here would be to look at all callers of WebGraphicsContext3DProvider::ContextGL and trace back whether the usage could be from a non-WebGL context to decide whether it needs to be updated. Our approach here has been to move all resource allocation to SharedImageInterface, which is designed to support cross graphics API synchronization of resources, and add APIs to RasterInterface for other functionality as needed.

The good part here is that we don't need to simultaneously maintain code using GLInterface and RasterInterface for the consumers. In order to allow an easy transition here, we currently support a RasterInterface implementation backed by a GL decoder. So the goal here would be to eliminate the use of GLES2Interface for each case incrementally until the shared context no longer needs to support it. One downside though is that we won't have coverage for the RasterDecoder path for these contexts until the step below is completed, but at least for scoped unit-tests we could run 2 versions of them using GL and RasterDecoder.

Another option, if we did not want to block the canvas2d work on the above, would be to create and use a different context for canvas rather than the shared one. This might have some gotchas though because code outside could assume that canvas2d is on the same shared context and lack synchronization across streams. I think eventually we do want to avoid using GL from any context other than WebGL, but it's worth evaluating whether a separate context in the short term is better to make progress with canvas2d first.

Switch GrContext use to RasterCHROMIUM

The second part is to switch uses of GrContext, which depend on running skia's GL implementation on the client side, to use RasterCHROMIUM commands on RasterInterface. The main code change for this would be in CanvasResourceProvider, which manages the rasterization backend used for canvas2d. The resource allocation here has already been moved to shared images, but we still import them into SkSurfaces on the client side which will need to

switch to using RasterCHROMIUM. A few sub-problems here that have already been identified are:

1) **Drawing accelerated images on canvas**

   The canvas context exposed to script supports multiple APIs for drawing accelerated resources (video, webgl canvas, 2d canvas, etc.) onto a canvas element. Since we currently draw them using an SkCanvas, this works by storing texture backed SkImages in paint recordings. In a lot of cases, we import shared image mailboxes into textures on the canvas context and then wrap them in SkImages for these draws.

   The ideal way for storing and serializing these accelerated resources in paint recordings would be as mailboxes. PaintImage is the class used to store images in paint recordings, and it supports different types of backing data, one of which could be a mailbox. When this PaintImage is deserialized on the GPU service, we can retrieve an SkImage for the mailbox which wraps the resource in whichever API is being used by skia. Note that we expect all resources used here to be created using shared images and thus have mailboxes, since the only context which can/should create GL textures on the client side is WebGL.

   In blink, these resources are passed around using StaticBitmapImage which already provides PaintImages for use with a PaintCanvas. The accelerated version of this would be AcceleratedStaticBitmapImage which can be created using mailboxes or SkImages wrapping textures. The Image base class provides an API (PaintImageForCurrentFrame) to get a Paint representation of the image for use in drawing with a PaintCanvas. The aim would be to have it return mailbox backed PaintImages instead of importing mailboxes to wrap in SkImages as is done today. This is somewhat tricky because a lot of the consumer code directly uses SkImage from PaintImage, mostly for operations like scaling, color conversion, readbacks, etc. So would need to be updated once PaintImage no longer provides an SkImage if its accelerated. One of the biggest such users is ImageBitmap.

   Another thing would be to get rid of the AcceleratedStaticBitmapImage::CreateFromSkImage factory method, so we always have images created using mailboxes only.

2) **Preserving transform/clip state on canvas**

   Canvas2D has APIs which allow script to save the transform/clip state on the canvas currently before modifying it using save/restore operations. This effectively allows the user to build a stack of this state that can be restored as needed. When calls are

executed on a canvas2d context from script, we store them in a PaintOpBuffer for deferred execution and clear this once they are drawn. For the common case, this is managed inside Canvas2DLayerBridge.

The transform/clip state is mirrored on the SkCanvas and is preserved across multiple draws through the use of a constant SkSurface, which allows us to only store paint operations executed by script since the last draw. But with OOP-R, we create transient SkSurfaces on the service side each time a resource is updated, so this transform/clip state is no longer preserved.

This means that either we need to retain the SkSurface on the service side (less likely) or need to ensure that each time a PaintOpBuffer is serialized for a draw using RasterCHROMIUM, we also serialize the transform/clip stack. This is already done in some cases where we do need to tear down the SkSurface (for instance if the page is hidden and we're purging all memory associated with the canvas element) here, so would just need to be hooked up properly for the OOP-R case.

3) **Eliminate canvas deferral disabling option**

Canvas2dLayerBridge used to support 2 modes for how execution of draw calls on script is managed with actually drawing them using skia. A deferred mode, where the operations are buffered in a PaintOpBuffer, and non-deferred mode where they are directly executed on an SkCanvas. The latter can no longer work with OOP raster, since skia runs in a different process. These were mostly workaround for issues, and it looks like all of them have been eliminated now. The only case I see now is Canvas2DLayerBridge::WritePixels, where we want to copy the pixels directly onto the canvas backbuffer and avoid an intermediate copy for deferred raster. This is not a concern when canvas is accelerated, we'd already need to do a copy to shared memory for sending those pixels to the GPU.

For the second part here, the change is significant enough that we'd need to support both code paths (GPU and OOP Raster) at the same time and roll this out on each platform via finch.