Android WebView Rendering

What is Android WebView?

Web content in a box:

- Powers browsers such as AOSP Browser
- Displays almost all mobile banner ads
- Can be used to create portable HTML-based apps
- Often intermixed imperceptibly with native Views: for example, to layout a paragraph of text in a fancy way

The secret 6th Blink platform

- Same compile-time flags as Chrome for Android, but lots of runtime differences:
  - Single-process
  - externally managed, synchronous rendering model
  - provides hooks to override cookies, networking, inject Javascript, etc etc
  - a few “quirks” WebSettings for legacy compat

WebView version history

Android <= J: custom WebKit-based “classic” WebView

Android K: Chromium 30/33-based WebView

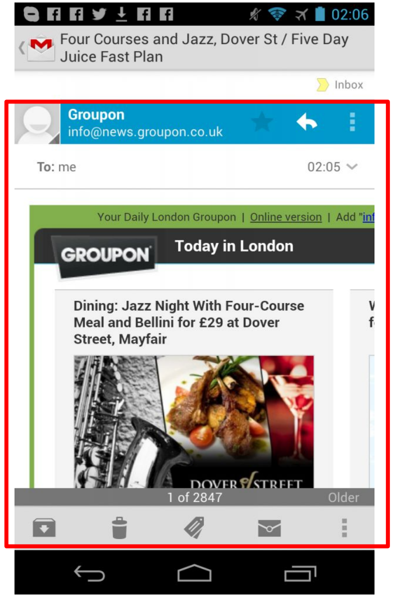
Android L: Unbundled evergreen WebView, autoupdated via Play Services

Use cases

When we started looking at how apps were actually using the WebView, we had some surprises...

Three Google email apps

GMail: All the emails in a thread are concatenated into a **viewport-sized** hardware WebView, with whitespace where the blue headers are. WebView handles scrolling and the app reads the scroll offset every frame to translate the headers.


02:06
Four Courses and Jazz, Dover St / Five Day Juice Fast Plan
Inbox
Groupon
info@news.groupon.co.uk
To: me
02:05
Your Daily London Groupon | Online version | Add "inf
GROUPON
Today in London
Dining: Jazz Night With Four-Course Meal and Bellini for £29 at Dover Street, Mayfair
1 of 2847
Older

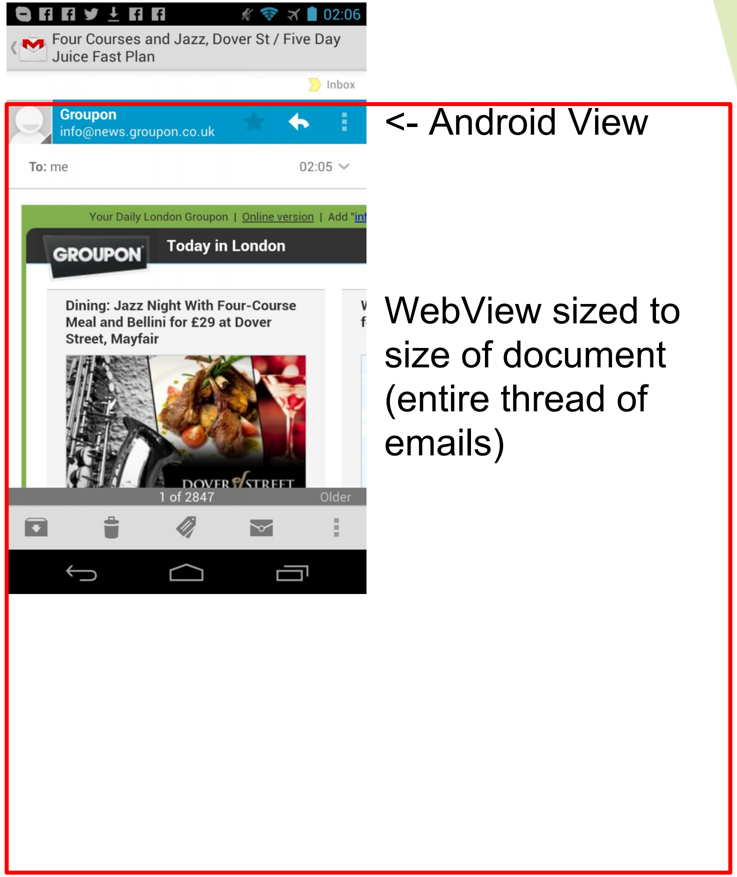
<- Android View

WebView sized to screen

<- Android View

Three Google email apps

“Email”:
document-sized hardware WebView. App handles scrolling by applying a hardware matrix to both WebView and blue headers.

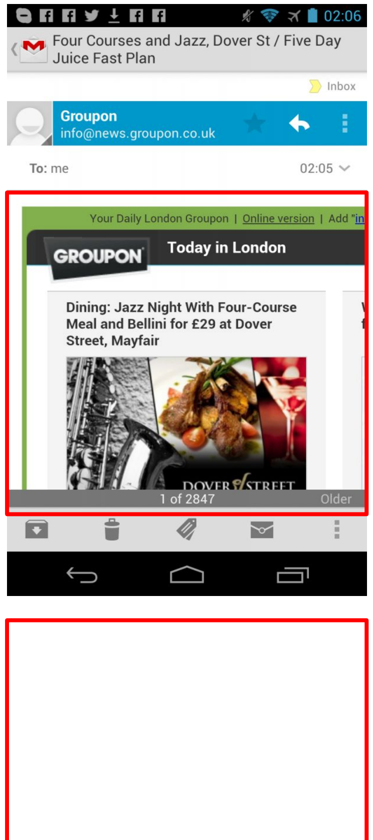

02:06
Four Courses and Jazz, Dover St / Five Day Juice Fast Plan
Inbox
Groupon
info@news.groupon.co.uk
To: me
02:05
Your Daily London Groupon | Online version | Add "inf
GROUPON
Today in London
Dining: Jazz Night With Four-Course Meal and Bellini for £29 at Dover Street, Mayfair
DOVER STREET
1 of 2847
Older

<- Android View

WebView sized to size of document (entire thread of emails)

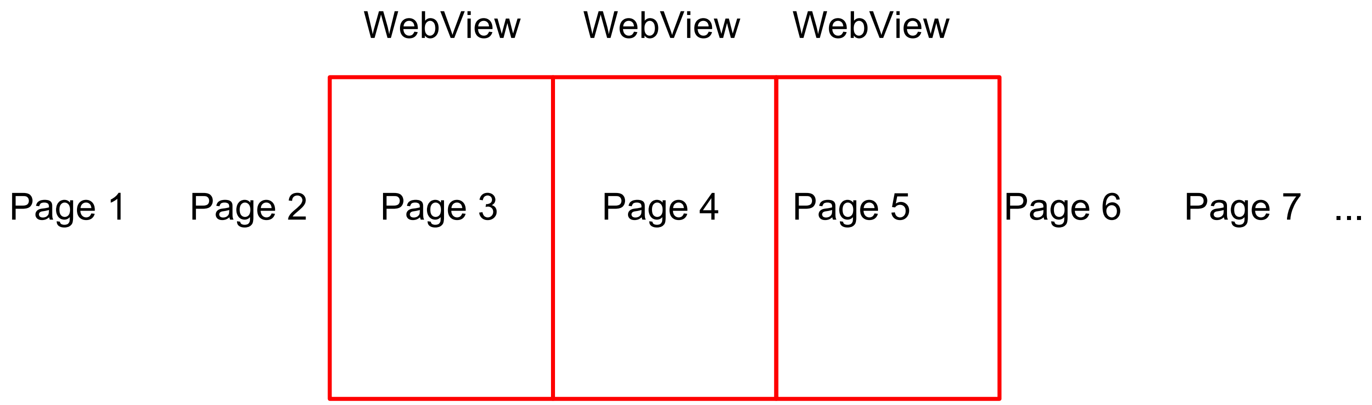
Three Google email apps

Inbox: One WebView **per email**. Custom RecycleView equivalent to avoid having more than ~3 active. No overlap between headers and WebView.


02:06
Four Courses and Jazz, Dover St / Five Day Juice Fast Plan
Inbox
Groupon
info@news.groupon.co.uk
To: me
02:05
Your Daily London Groupon | Online version | Add "in
GROUPON
Today in London
Dining: Jazz Night With Four-Course Meal and Bellini for £29 at Dover Street, Mayfair
DOVER STREET
1 of 2847
Older
WebView 1
WebView 2

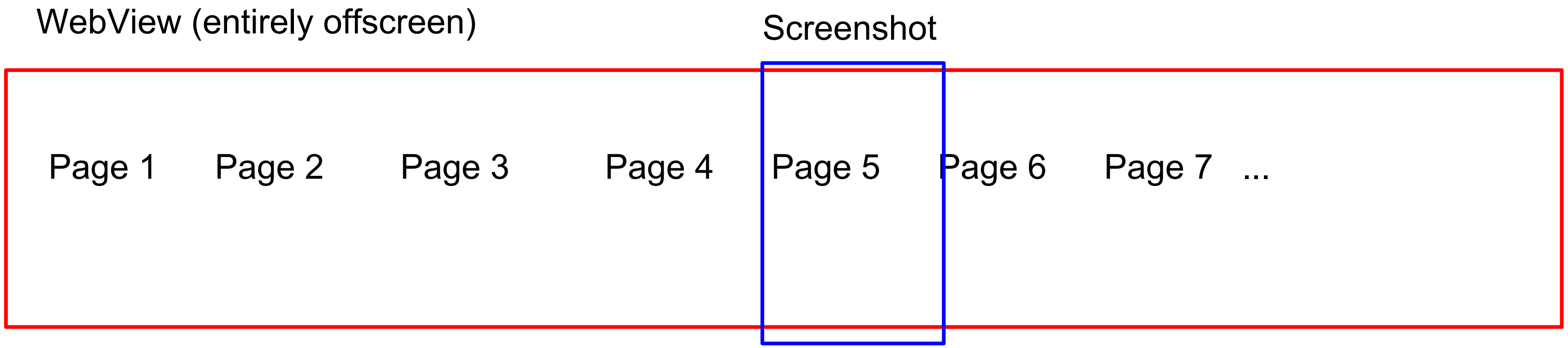
Pagination with three WebViews

Common pattern across news apps: one WebView per page, handles touch events in Java and applies View translation matrix to side-swipe. WebView itself mostly passive (may vertically scroll in “tab”-style use cases).



Pagination using software screenshots

Google Books: Each chapter is one giant WebView. Uses WebView.onDraw() with translation matrices to render individual pages into software bitmaps. Then uploads those into textures and does fancy page-turning using its own GL.



Fun legacy WebView APIs

WebView.capturePicture(): Return a “Picture” object (Java wrapper around SkPicture) for the entire document.

WebView.setPictureListener(): Call this callback, providing a Picture object for the entire document, every time it changes.

APIs in View base class

View.onDraw(Canvas): Synchronously draw the view into an Java-wrapped SkCanvas. SkCanvas usually backed by a software bitmap. Cannot checkerboard, and an arbitrary matrix can be applied.

View.get/setScrollX/Y(): Synchronously set or read scroll offset.

View matrix setters: Synchronously translate/scale.

Android assumes all Views are quick and easy to redraw from scratch

Other types of Android Views tend to simple enough to regenerate from scratch on a single thread at 60fps.

Web content is complex enough that we have an async pipeline generating a tile cache instead.

An Android View is a bit like a RenderObject in Blink

- Every frame, Android walks the View tree, performs layout if needed, and asks all Views to issue draw commands at the latest size and position
- It’s a “pull” rendering model rather than the “push” model that’s dominant in Chromium
- (To be fair, the average Android View is more heavyweight than the average RenderObject.)

Implication #1: Android decides our scroll offset and tells us at the last minute

- The scroll offset or matrix may change just a few instructions before the draw call.
- It may even suddenly teleport us to a distant part of the document. In software mode, correct rendering is still expected when this happens (cannot “checkerboard”).

Implication #2: Nothing can be async

If **anything** is async in the input or graphics pipeline, it would no longer work to write a Java method that synchronously:

1. Hands WebView a touch event
2. Reads the WebView’s scroll offset
3. Applies a matrix to another View relative to it

Implication #3: Always be ready for software draws

- **API says nothing about "hardware mode" or "software mode"**. Although hardware draws can only happen when the WebView is attached to a hardware-accelerated View tree, software draws to a side canvas may occur at **any** time.

The implementation

(TLDR: clone classic Android WebView architecture by reshuffling Chromium components)

Threading model summary

Chrome

Multi-process

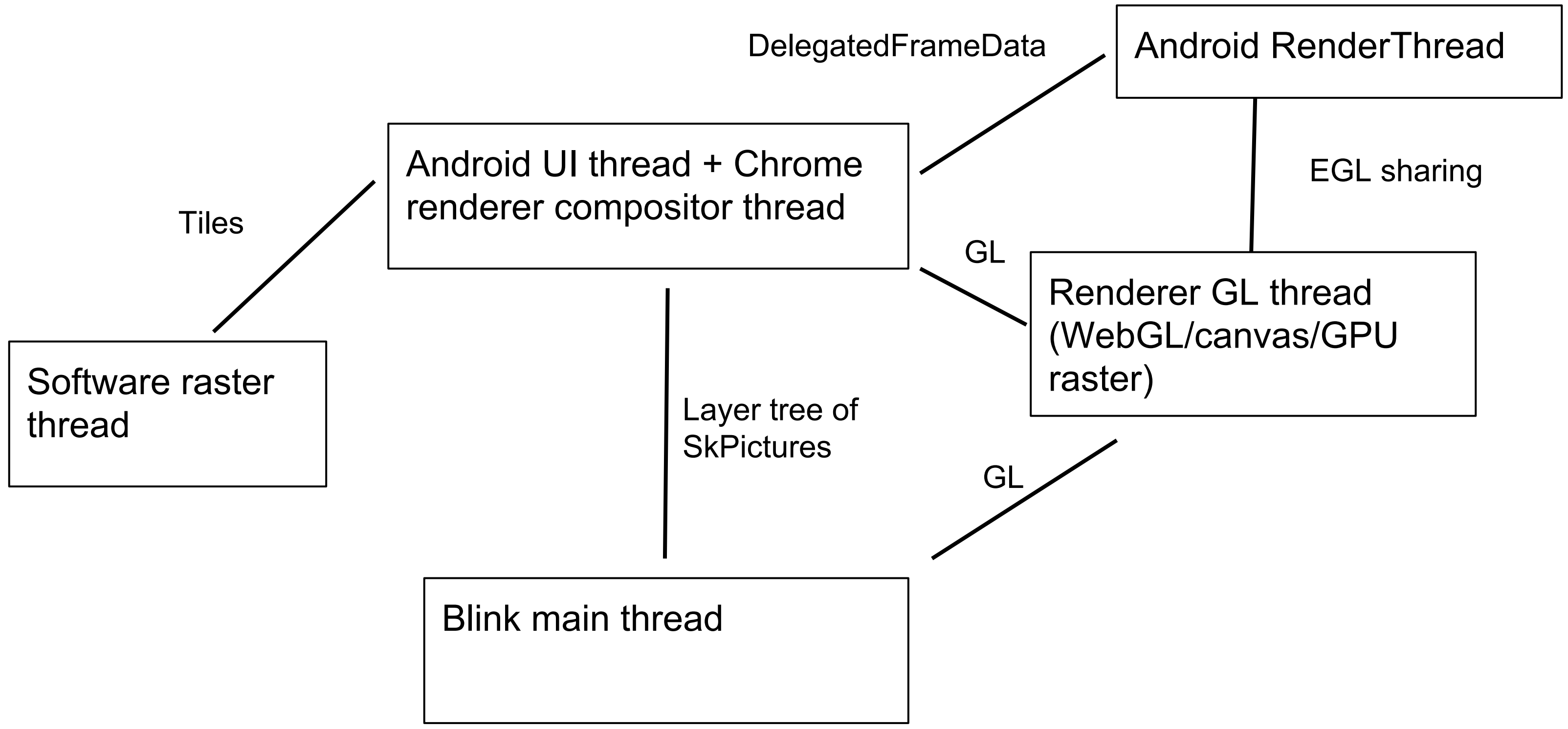
- UI thread
- GPU thread
- Texture upload thread
- Per renderer process:
  - Blink thread
  - CC impl thread
  - Raster thread

WebView

Single-process

- Combined UI + renderer CC thread
- Android RenderThread (+in-process GPU thread)
- Canvas/WebGL GPU thread
- Blink thread
- Raster thread

Threading model summary



System graphics components

Chrome

- SurfaceView hardware overlay
- Async uploads using EGLImage and glTexSubImage2D().

WebView

- “Draw functor”: inject draw calls into system GL context.

Merge CC impl thread and UI thread

- Classic Android WebView used the UI thread for compositing. That's why it had all those synchronicity properties.
- We considered keeping the renderer CC thread and using sync IPCs. But, just about everything would have ended up as a sync IPC.

Single-process

- Avoids shipping vast data for capturePicture.
- CC currently doesn’t support commit across processes.
- There were other non-graphics reasons to go this route, so this assumption was locked in early anyway.

Android RenderThread + GPU thread

- New as of L release.
- Android now pushes DisplayLists and runs all GL on this thread instead of on the UI thread. Benefit:
  - Can apply a matrix on Views for off-main-thread transition animations. (Same motivation as our Chrome's compositing thread, but doesn't handle input.)
- We're using DelegatingRenderer for this. Parent compositor (pushing to in-process GL commandbuffer) is on the RenderThread, child compositor on the UI thread.

Canvas/WebGL thread

- Provides asynchronicity and isolation. (Would be scary to expose system GL context to arbitrary GL.)
- Output surface shared with system context using EGLImage.
- In the future, we plan to move GPU tile raster work to this thread as well.

Blink and raster threads still separate

- Finally a place where not much weird happens :). They were separate in classic WebView as well.
- Note that raster thread is only used for hardware rendering. Software rendering is rastered synchronously on the UI thread.
- Optionally, SkPicture recordings can be infinite-sized to deal with the teleportation+software-draw problem. For newer targetSdkVersion we limit the size of the recording by default for better scalability.

“Draw functor”

- Private API. The WebView injects a callback onto the display list. When the painting reaches that step, the callback is called and the WebView runs arbitrary code to produce arbitrary GL, then returns.
- Implicit contract to save/restore GL state.
- Had to be used instead of public APIs “SurfaceView” and “TextureView”:
  - The public APIs push frames asynchronously

WebView software rendering

- Shares some code with desktop platform “software compositing” but very distinct in other ways.
- Also shares some code with GPU raster. Skia runs on the UI thread rastering all the SkPictures into the target canvas.
- Tiling and raster thread infrastructure not used at all. Reasons:
  - Maintaining software and hardware tiles at the same time would be huge headache.
  - Would have caused memory usage and invalidation time regression
- No support for RenderSurface-based features, and alpha blend ordering is incorrect.

Scheduling differences

- Still uses ThreadProxy between Blink and UI thread…
- But CC doesn't make its own scheduling decisions. It draws immediately whenever Android decides.
- When the CC state machine decides a draw is needed, it calls out to Android View.invalidate().
- Draws can also be forced for View-system-related reasons (for example, another small view is animating on top of us).
  - We have several bugs/hacks around unpredictable draw timing (differently sized draws that race CC invalidations, draws that don't come after CC "requested" them)

Where's the code

android_webview/ : Upstream Chromium directory with most of the WebView code.

frameworks/webview: Android tree with WebView "glue" code for private APIs like draw functor.

content/browser/android/in_process/synchronous_compositor_output_surface.h : main interface between CC and android_webview/

The future of WebView?

- Under active debate at the moment
- Opposing use cases:
  - WebView as a browser-in-a-box, VS
  - WebView as one tool in a hybrid app toolset
- Opposing update priorities:
  - App developers want the latest features and performance improvements
  - They also want stability to the point of bug-for-bug compatibility

Questions?

Contact the team at **android-webview-dev@chromium.org**

Extra slides

“Gralloc” zero-copy buffers

- Stopped using these in Android L WebView
- Motivation #1 was an early plan for hybrid software rendering that involved unified hardware/software tiles: relied on gralloc explicitly exposing unified memory model.
- Motivation #2 was the idea that gralloc is much better and WebView would be pioneering it for Chrome to later adopt.
  - But it turns out gralloc kind of sucks (memory fragmentation, filehandle limits…)

Hardware rendering initialization

- All WebViews start by supporting only software draws.
- When the WebView is attached to a View tree marked for hardware acceleration, *and* the first hardware draw occurs, CC’s GL infrastructure is synchronously initialized and a frame is drawn.
- When the View is detached from the View tree, the GL infrastructure is torn down and all resources are deallocated.
- Can still respond to software draws at any point in the lifetime. Not a mutually exclusive mode like in Chrome.

Viewport model differences

- Additional matrix to be applied above the CC root layer.
- Draw functor allows CC to clobber the entire screen. We must carefully restrict ourselves to the View bounds.
- WebView can be extremely tall (sized to document). In this case, fixed-pos layers are positioned according to WebView size, while tiling decisions must be based on actual visible size.
- Scissor for a single draw may be smaller than actual visible size too (“another small view animating on top of us” case)