Native OpenGL ES 3.x in ChromeOS

martin krastev

What is ChromeOS?

Google's take at 'a thin web client' – an OS for runnig Chrome

- Launched in 2011; based on Gentoo Linux with emphasis on security (SELinux) and performance; architecture-agnostic (x86, ARM, ports to MIPS)
- Lergely open-source – ChromiumOS == ChromeOS sans the closed parts
- Open platform – anybody can make a certified ChromeOS device if it complies with Google's requirements
- A true consumer desktop Linux according to Linus Torvalds [1]
- Initial support for Android apps since 2014
- Full support for Android Play Store since 2016

Android's native 3D API (since 2014/Android 5.0) is OpenGL ES 3.x. Hmm..

How do Android apps run on ChromeOS?

ARC++ paper from XDC2016 [2]

> *“App Runtime for Chrome (ARC), was launched September 2014 with a few curated applications.”*
>
> *“[current] ARC++ is Android running alongside ChromeOS and providing ChromeOS users with Play-Store Android apps without developer action.”*

Essentially, ARC++ is an Android full-compatibility layer in ChromeOS.

How to do out-of-browser GLES work in ChromeOS

- Android IDE apps – self-hosted Android development? Could be an option..

How to do out-of-browser GLES work in ChromeOS

- Android IDE apps – self-hosted Android development? Could be an option..
- Debian via KVM (since R70) – full Linux toolchain but no VGPU just yet : /

How to do out-of-browser GLES work in ChromeOS

- Android IDE apps – self-hosted Android development? Could be an option..
- Debian via KVM (since R70) – full Linux toolchain but no VGPU just yet : /
- ChromeOS ‘developer mode’ – relaxed security – full* FS access
    - Crouton – chroot environment that enables Linux userspaces on top of ChromeOS; as desktop distros don’t support the GPUs in non-x86 chromebooks, this comes of little help to us : /

How to do out-of-browser GLES work in ChromeOS

- Android IDE apps – self-hosted Android development? Could be an option..
- Debian via KVM (since R70) – full Linux toolchain but no VGPU just yet : /
- ChromeOS ‘developer mode’ – relaxed security – full* FS access
  - Crouton – chroot environment that enables Linux userspaces on top of ChromeOS; as desktop distros don’t support the GPUs in non-x86 chromebooks, this comes of little help to us : /
  - Native – full Linux toolchain via Chromebrew – an open-source package manager for ChromeOS; as contemporary ChromeOS mandates GLES 3.x support, a user-space GLES stack comes factory-provided on every chromebook :)

How do we access that stack?

First thing first: what does such a stack comprise of?

- Kernel-side GPU buffers management – DRM (Direct Rendering Manager) client-side library: /usr/lib/libdrm.so ✓
- Native provider of GLES contexts – EGL API: /usr/lib/libEGL.so ✓
- User-space GLES API stack for ES 2.0 and above: /usr/lib/libGLESv2.so ✓
- On-screen visualisation of GPU-originating DRM buffers – ?

Screen-posting of GPU content shall remain an unknown for now..

Phase one: pinging the GPU

An essential prerequisite:

DRM must support render nodes – non-exclusive access to the GPU DRM device(s). ARC++ paper from XDC2016, page 9:

> *“**Direct Rendering Manager (DRM)***
> *Used for rendering and buffer allocation on both Android and ChromeOS*
> *Allows for efficient sharing of graphics resources (DMA Buffers) between the two platforms*
> *Chrome is DRM-Master and can program a display-controller*
> *Android doesn’t need mode-setting capabilities in ARC++*
> *Render-node access to the GPU is sufficient”*

that’s exactly what we need ^ ^

Phase one: pinging the GPU, cont’d

Try to create a PBuffer-based GLES2 context and see what comes out of it

1. Establish an EGL connection to the default display
2. Enumerate all PBuffer-capable EGL configs
3. Pick one of desired RGBA format and create a PBuffer surface from it
4. Create a GLES 2 context with both draw *and* read surfaces set to that PBuffer
5. Draw something via that context
6. Read back via glReadPixels and save to file

Stare profoundly at results..

Phase one: pinging the GPU, cont’d

Single-light-source Phong with tangent-space normal-mapping over a polar sphere

*Actual grab from one of the first ChromeOS runs, or thereabout.

Phase two: on-screen at any cost!

How do we show framebuffers on screen? Once again, ARC++ to the rescue!

- ARC++ provides Wayland protocol into the underlying ChromeOS compositor
- Chromebrew provides libwayland – contains the necessary client-side libraries

Unfortunately the path that works on most other platforms – wl_egl_window_create, is not supported on ARC++ – oh well.

Roll up sleeves for manual labor – copying pixels from the framebuffer into a SHM-based wl_surface – works every time..

Phase two: on-screen at any cost, cont’d

..But at a high price : /

On a low-power chromebook the CPU-carried glReadPixels can use as much as 30-40% of one core for a modest surface of 1Kx1K RGBA8888 (no pixel conversion) displayed at 60fps. Even on a high-end chromebook that can use as much as 10-20% of a core – yikes!

Phase three: doing it the right way

ARC++ paper from XDC2016, page 17:

> *“**Wayland Interfaces***
> *In addition to the core wayland protocol, Chromium implements the following interfaces*
>
> *wp_viewporter*
> *xdg_shell_v5 (zxdg_shell_v6 experimental)*
> *wl_drm*
> *zwp_linux_dmabuf_v1”*

I wonder what that last one does ^ ^

Phase three: doing it the right way, cont’d

A picture starts to slowly clear up..

- PBuffer-based surfaces are *the sole* type of renderable surfaces EGL provides in ChromeOS
- We need a GLES context where we set our render target (via FBO) to some special image type
- That image type has to come from a linux_dmabuf kernel object

Easy-peasy.. Right?

Phase three: doing it the right way, cont'd

The curious case of the surfaceless context (EGL_KHR_surfaceless_context)

> *"These extensions allows* [sic] *an application to make a context current by passing EGL_NO_SURFACE for the write and read surface in the call to eglMakeCurrent. The motivation is that applications that only want to render to client API targets (such as OpenGL framebuffer objects)* [like we do] *should not need to create a throw-away EGL surface just to get a current context."*

Sure, I'll take one of those low-calories contexts, thank you!

Phase three: doing it the right way, cont’d

‘dmabuf’ images and where to find them

1. Figure out the DRM device for the sole GPU: /dev/dri/vgem
2. Get a ‘dumb’ buffer object: ioctl(DRM_IOCTL_MODE_CREATE_DUMB)
3. Promote that to a ‘prime’ FD: ioctl(DRM_IOCTL_PRIME_HANDLE_TO_FD)
4. Turn that into an EGLImage: eglCreateImage(EGL_LINUX_DMA_BUF)
5. Use image as renderbuffer storage: glEGLImageTargetRenderbufferStorage

Voila – we have an dmabuf-based renderbuffer.

Phase three: doing it the right way, cont’d

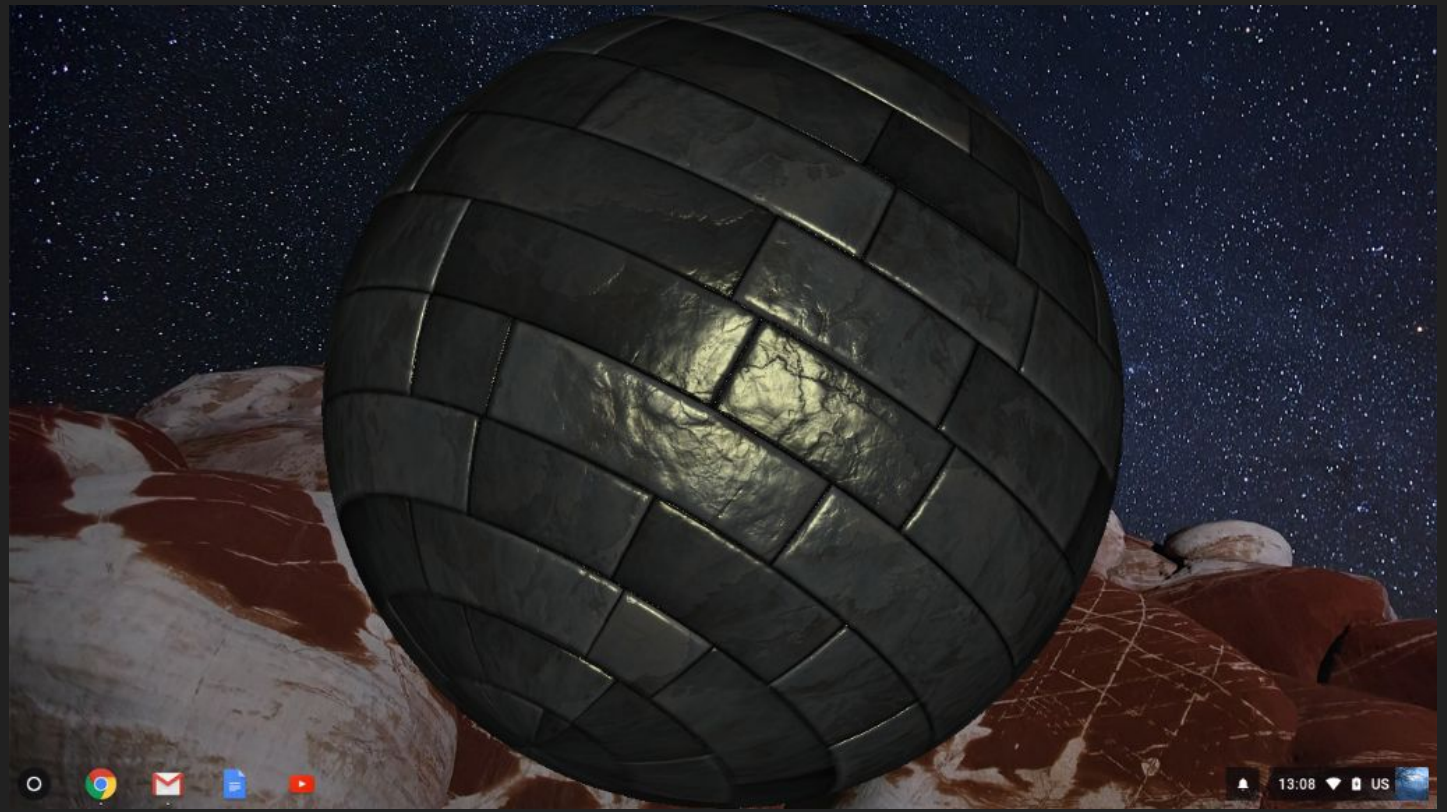
Now do the Wayland wiring via that zwp_linux_dmabuf_v1 protocol extension

1. Procure a functional zwp_linux_dmabuf_v1 interface from Wayland
2. Use the above to create a wl_buffer from the same ‘prime’ FD we got earlier
3. Use that wl_buffer as backing storage for a wl_surface

Now we can render to that EGL image and it magically appears in that wl_buffer! The rest is standard Wayland compositor redraw logic.

Demo code

https://github.com/blu/hello-chromeos-gles2



Conclusions

Google may not have meant for Chromebooks to be used that way, but we *can* use them in bare-bone dev mode for native GLES3.x development!

Pros

- Entry-level chromebooks sell for sub-$300, and ARM models provide 8-12h of work on a single charge
- Chromebrew – familiar Linux toolchains, fresh package versions
- ChromeOS *is* ‘consumer-grade’ Linux – superb web experience, media codecs ‘just work’, external displays & extended desktops, active community and long-term Google support

Conclusions

Google may not have meant for Chromebooks to be used that way, but we *can* use them in bare-bone dev mode for native GLES3.x development!

Cons

- Mind the updates! Google may (and does) break functionality as they see fit – some Chromebrew packages need rebulding every couple of major ChromeOS revisions
- As we use ARC++ APIs (i.e. Exosphere) not everything might be fully documented, but ARC++ is open-source, so we can track down issues and file bug tickets with Google, which they even fix!

That’s all, folks! Questions?

External references

[1] Linus Torvalds in Conversation with Swapnil Bhartiya, https://www.youtube.com/watch?v=pQWj2Fgxdrc
[2] David Reveman, ARC++ Graphics: Rendering, Compositing and Window Management, https://www.x.org/wiki/Events/XDC2016/Program/Arcpp_Graphics.pdf
[3] Crouton project, https://github.com/dnschneid/crouton
[4] Chromebrew project, https://github.com/skycocker/chromebrew
[5] Hello-chromeos-gles2 project, https://github.com/blu/hello-chromeos-gles2