Demystifying synchronisation

Elaborate communication rituals to bring pixels to your retinas

Daniel Stone

Graphics Lead, Collabora

daniels@collabora.com

Open First

What are we covering?

- A very basic 15-year old GPU model
- Sharing between contexts & processes
- Implementing this in the kernel
- Our strange ~~future~~ present
- Practical presentation pipelines

In the beginning, there was DRM

- Let's render a triangle and read it back on the CPU
- A simple example from a simpler time
- One device, one FIFO command queue
- DRM provides an interface (of sorts) for userspace to use GPU hardware

DRM: it’s a state of mind

- DRM has *almost no* generic ioctls
- All device access is through device-specific ioctls, forcing device-specific userspace
- Mesa is the canonical example for gfx
- This is why we’re insistent about open userspace: over half the driver is there!

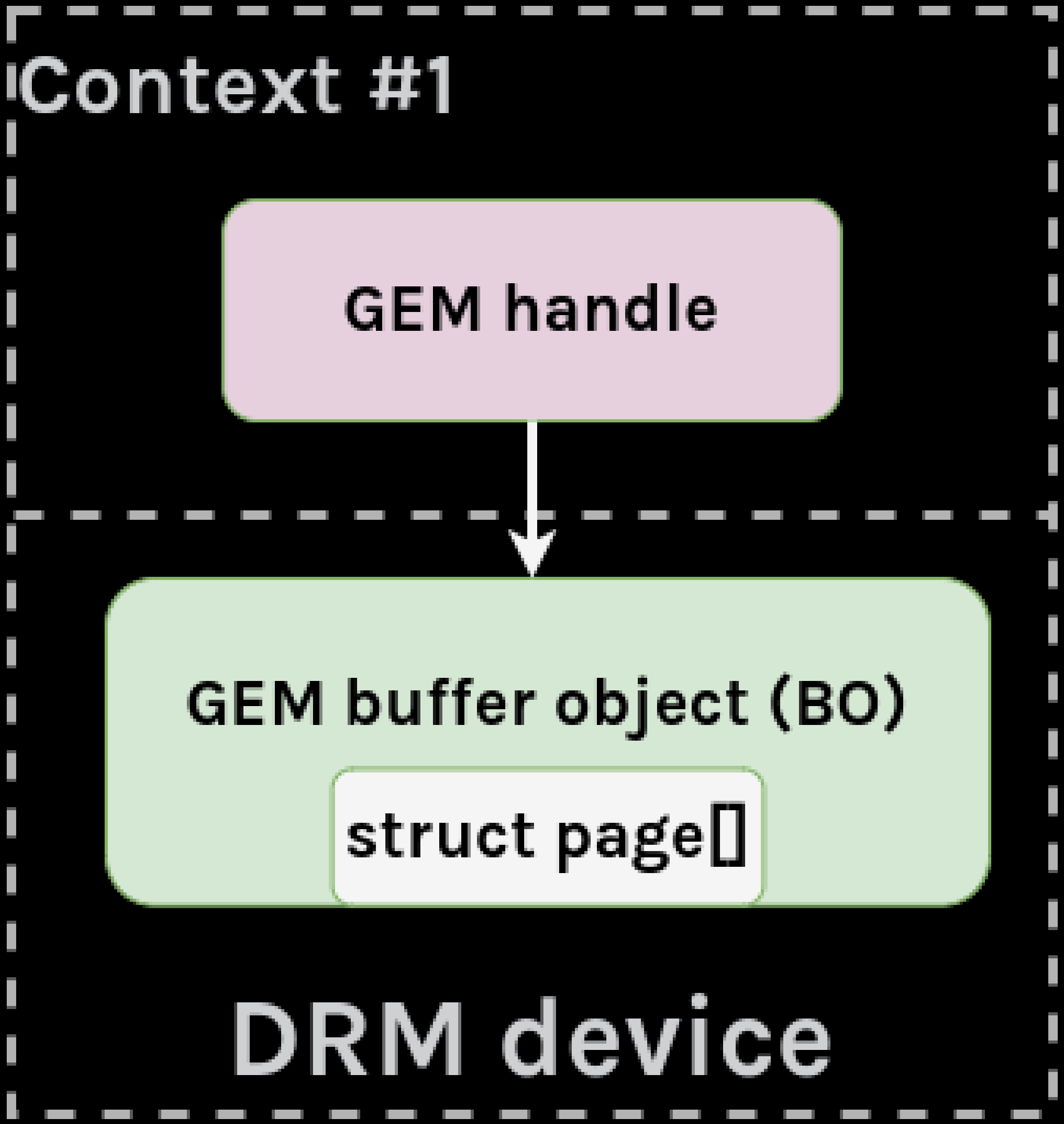
Step 1: memory access

- First device-specific ioctl: allocate mem
- GPU memory is special, not malloc
- Memory mostly untyped + byte length
- ‘BO’ is a buffer object: pointer & size
- Not just pixel data: also state, programs

Step 1: memory access (internals)

- Memory is allocated in system or VRAM
- Allocation is recorded globally to device
- Exposed to context via integer handle
- Internally usually an array of struct page[]

GEM BOs


Context #1
GEM handle
GEM buffer object (BO)
struct page[]
DRM device

Step 2: send the GPU into action

- Second device-specific ioctl: command submission
- CS ioctl will take input/output buffer list + auxiliary buffers (state, code)
- CS ioctl will append commands to queue

Step 3: getting to the CPU

- Third device-specific ioctl: buffer access
- Similar to DMA API, will take buf + area + access mode
- Map the buffer memory into CPU-visible
- A triangle! In CPU memory!
- [pause for applause]

But aren’t GPUs asynchronous?

- Oh … yeah.
- But remember how we passed the BO list into the command submit ioctl?
- That wasn’t just for fun.
- Knowing which commands touch which BOs lets synchronise against them

Implicit synchronisation

- Implicit synchronisation creates the illusion of synchronous/FIFO work
- CS ioctl takes list of {BO, access mode}
- Mapping buffer into CPU address space stalls in driver-specific ioctl for all commands to complete

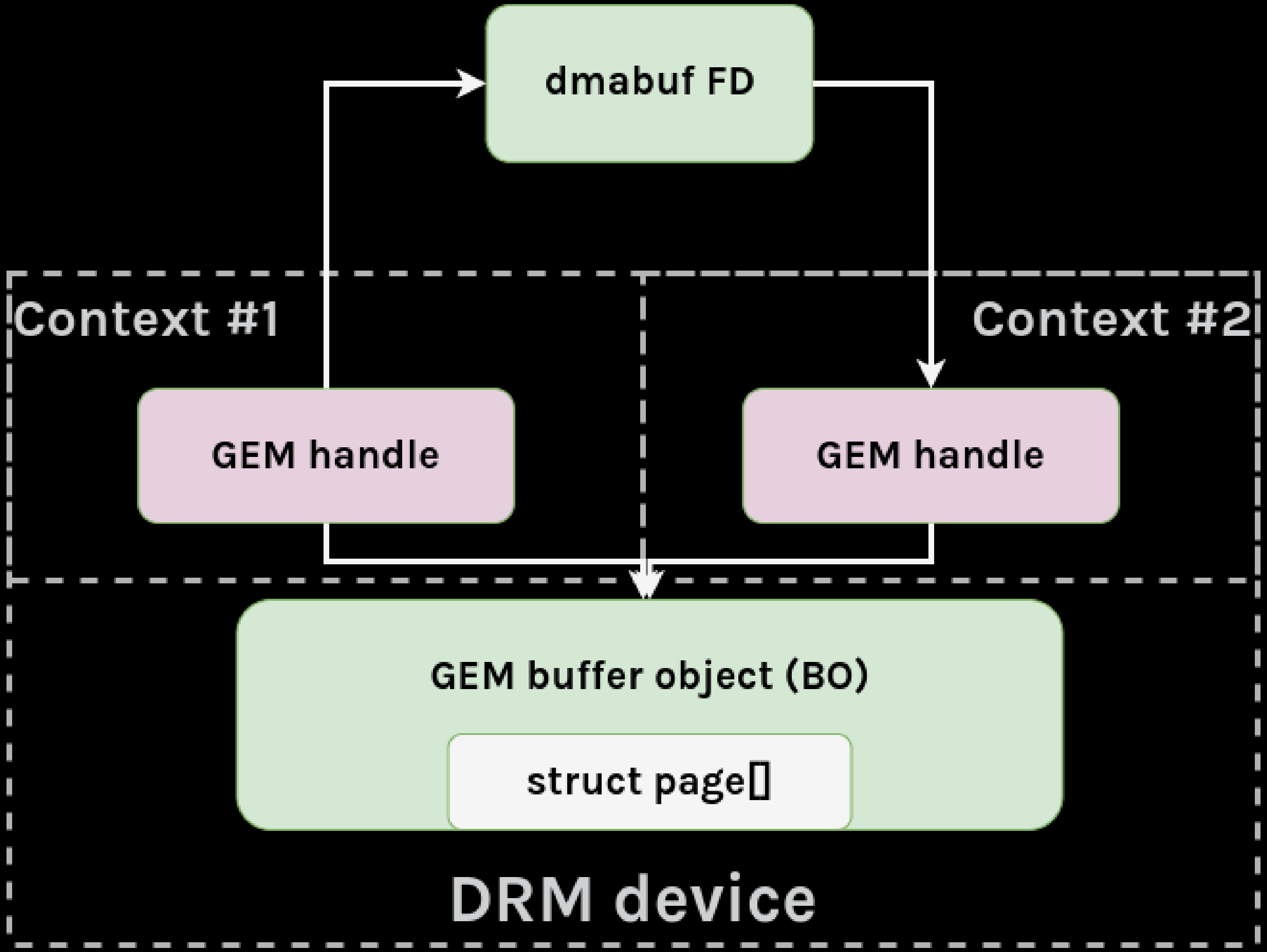
Implicit everywhere!

- Not just for CPU vs. GPU access …
- Implicit sync lets us share between processes/contexts
- Driver records ‘breadcrumb’ of hardware sequence number
- Avoid WAR/RAW hazards via stalls

Sharing between processes?

- dmabuf is a FD wrapper around BOs
- Allocations happen in device, BO handles in context, dmabuf system-wide
- Allows BO references to be passed between contexts/processes

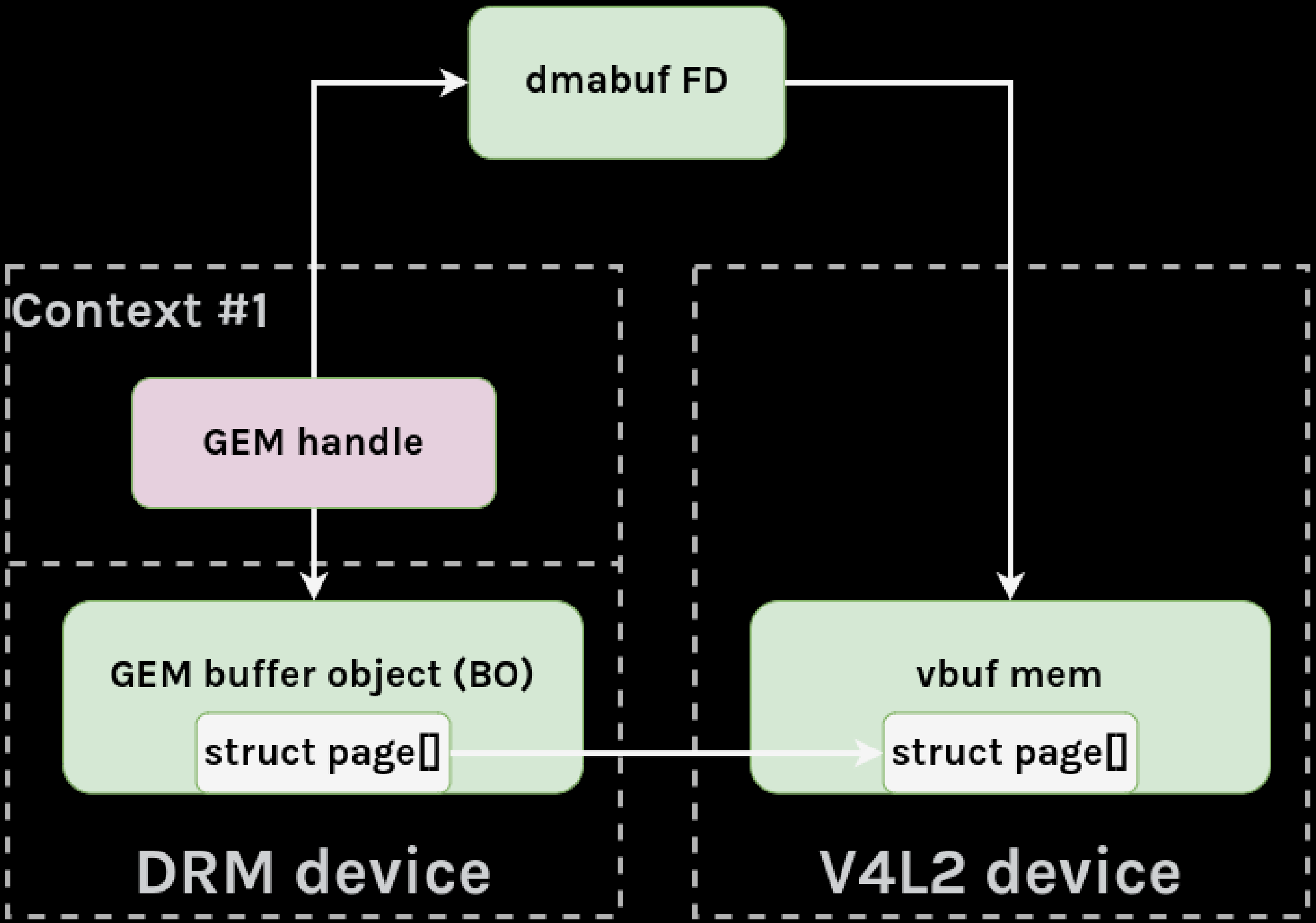
GEM & dmabuf


dmabuf FD
Context #1
Context #2
GEM handle
GEM handle
GEM buffer object (BO)
struct page[]
DRM device

Sharing between devices?!

- dmabuf is generic kernel API allowing sharing between devices/subsystems
- GPU memory isn’t *that* special, mostly just pages
- Each subsystem has its own import/export API for dmabuf

GEM & V4L2 & dmabuf



What can’t dmabuf do?!

- Well, a lot
- dmabuf is not an allocator
- dmabuf is not a constraint solver
- It’s just a handle to pages, and a semi-complete cache-coherence interface

Oh, that’s disappointing

- It’s a start, at least …
- Lingua franca for buffer sharing
- Kernel dmabuf users: DRM, V4L2, others
- Userspace: Wayland, X11, EGL, Vulkan, GStreamer, PipeWire, VA-API, everything

Great! Where to now?

- Back to the talk topic maybe?
- Now we’ve exposed buffers, let’s expose sync operations as well
- And surprisingly, as a FD …
- CS ioctl returns dma-fence FD to signal completion of GPU-side work

What is dma-fence?

- Also a FD, also cross-device/subsystem
- Also importable by all kernel+userspace
- FD materialised when work is queued
- Signals once when work is completed
- Guaranteed to signal in ‘reasonable’ time

dma-fence in the kernel

- Same-device as efficient before: device can do internal sync operations against own fences
- enable_signalling() callback forces CPU notification of work completion
- Userspace can poll on FD, other devices can get callbacks to schedule own work

Even more levels of illusion

- dma-resv ties a dmabuf to dma-fences
- Allows implicit synchronisation across contexts/processes
- Before you schedule any work against a BO, check the dmabuf dma-resv for others’ fences
- Place a dma-fence on the dma-resv as you do schedule work, for others to sync against

Easy as you like

- But why do we have dma-resv when all userspace supports dma-fence?
- Partly we have to forever, because X11
- But mostly because it’s not actually sufficient for what we need …

How hard can it possibly be?

- So we just need to turn binary to integer, right? drm-syncobj does this, right?
- syncobj gives us the mechanism to contain multiple fences in a single container, which are roughly the semantics we want

But …

- Remember how I said fences complete in guaranteed time?
- Timeline semaphores don’t: they allow wait-before-signal
- This makes a mockery of our dependency scheduler

But …

- Hard to schedule jobs with WBS
- Painful interactions with memory management: swap, reclaim, etc
- Can’t provide the same interop with implicit sync because it might never fire
- Pain.

The tip of the iceberg

- Hardware has fully isolated contexts
- New APIs need user-controlled VMAs
- Full autonomous scheduling for rings
- Parallels with io_uring/RDMA: kernel not adding any value, just overhead

Anything else?

- Games truly need huge throughput
- Heavily pipelined, speculative, asynchronous operations and paging
- So we have to let them use the fancy user contexts & rings to meet performance demands

So we’re solving for games

- Also GPGPU/compute
- Long-running workloads, gigantic data sets
- GPU-demand page faults coming any year now …
- Different demands, same hardware + APIs

What’s your brilliant solution then?

- Well, we’re not quite sure
- … but neither are the hardware people
- Current thinking is probably a hybrid
- Userspace gets its own happy little world most of the time, at full performance

What’s your brilliant solution then?

- But as soon as interop is required, degenerate to lower-performance mode
- Let userspace run ahead until it needs to interact with the outside world
- Interfere only at the margins, enforce only what we need to

Sounds easy?

- Not entirely, no
- Violent kernel & userspace API change required
- Hardware designs aren’t finalised
- A lot of open questions, not loads of time

Is display at least easy?

- Well, pretty straightforward
- Once client has finished draw and handed off, no need to track thousands of draw calls – just a couple
- Only one or two fences needed
- No CPU overhead concerns with this

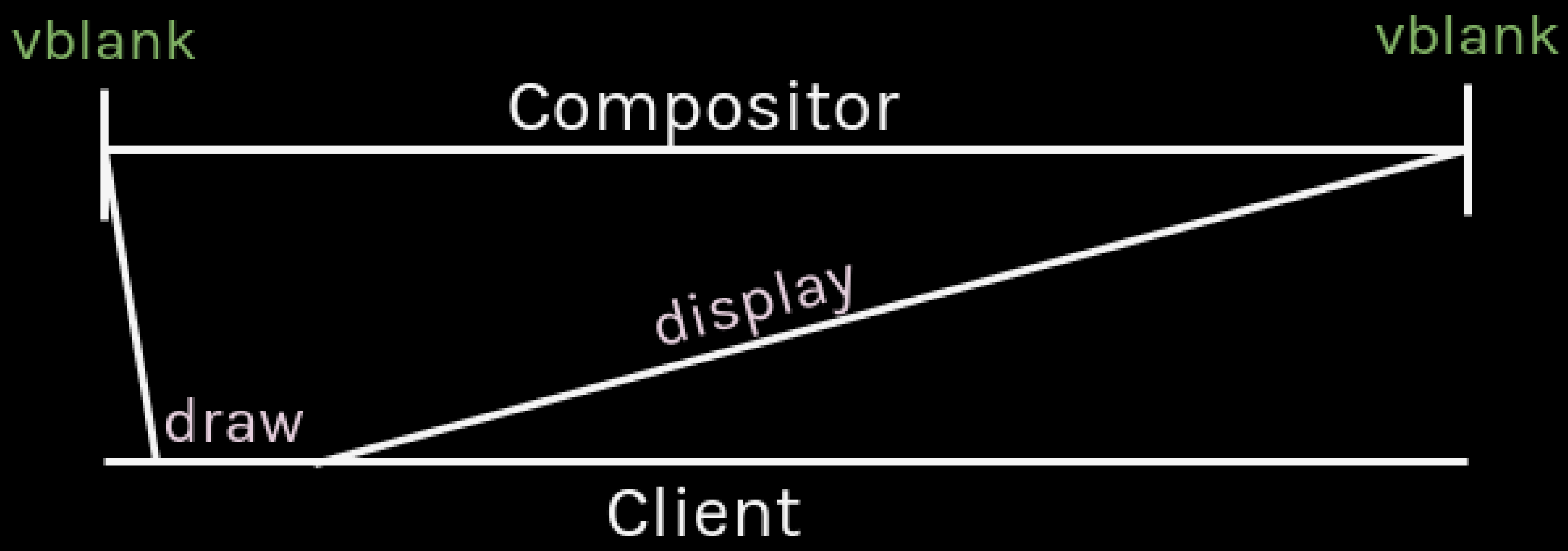
Oh, that’s good then

- Display is also backwards though …
- Data flows forwards: client paints content, hands to window system, window system hands to display
- But timing flows *backwards* from display to window system to client

Why?

- The display clock is (sort of) fixed
- We know exactly what deadline we need to hit to get something on screen
- Compositor works backwards to find latest time to prepare content and hand off to display

Ah, vblank!



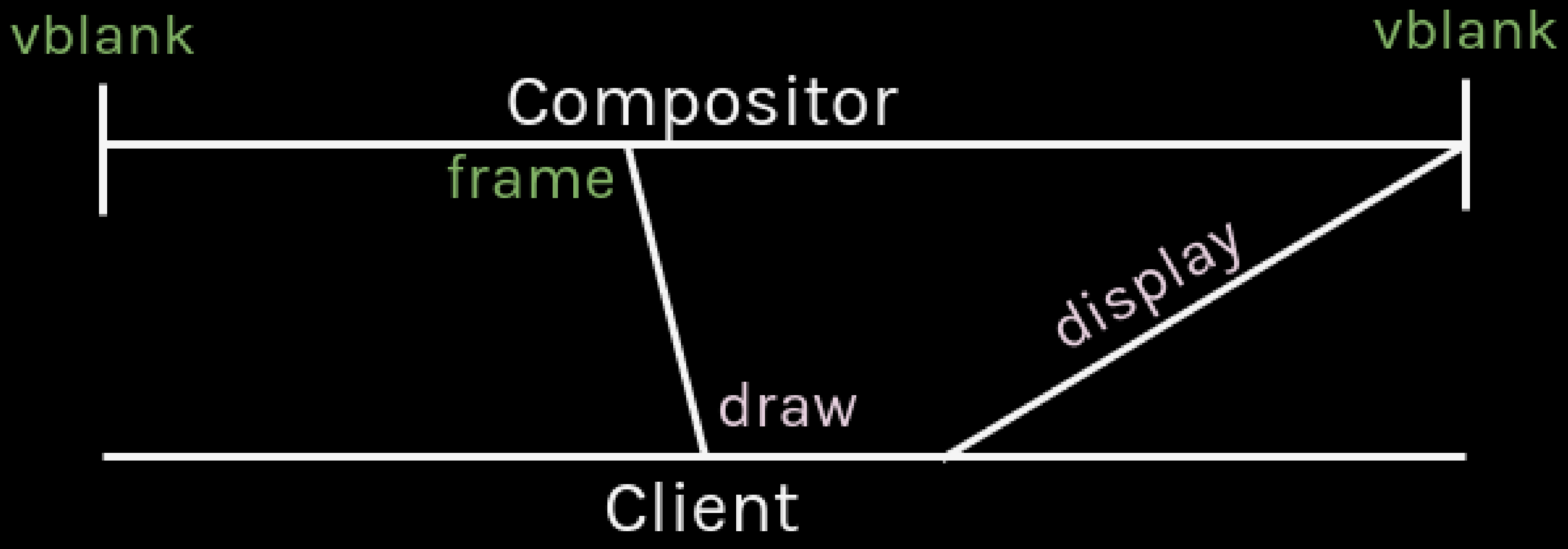
vblank! Easy.

- Working from vblank is great if you can produce content slightly faster than the display runs
- Working from vblank is great if you don’t mind 16ms latency
- Neither of these things are true

vblank strikes back

- Reduce latency by introducing a separate ‘paint now’ signal
- Also allows adaptation to slow clients by pacing them to lower multiple or overlap
- See Michel Dänzer’s Mutter work

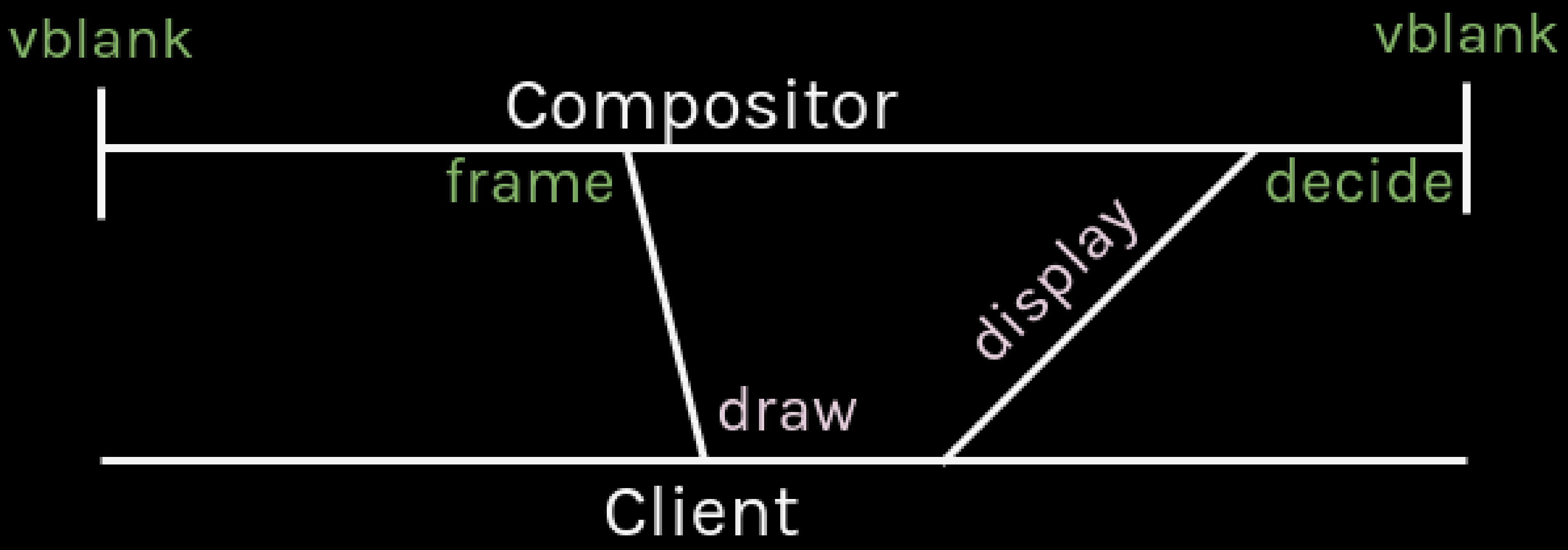
vblank + frame



Compositor complexity

- But what about wait-before-signal clients?
- Need to introduce late-binding decision point in compositors
- Just before paint, check for readiness and decide between old & new

vblank + frame + WBS



What have we learned?

- Graphics is surprisingly difficult
- Games wants console latency & perf, desktop unknowability, laptop hardware
- Compute wants HPC functionality and throughput on fixed workload/hardware
- Both of them share the same APIs …

Sounds difficult

- Well, it’s got all of our attention at least
- Many vendors & community members focusing hard on all these topics
- It’ll keep us busy for a while
- … and in the meantime, no-one can buy GPUs anyway

Thank you

```
Message {
  config {
    priority: "high"
    body: "Collabora is hiring"   // Many open
positions
    recipient: "you"              // Please
join us
    calltoaction: "http://col.la/join"
  }
}
```