Chemnitzer Linux-Tage 2014

KMS UXA DRM OMG WTF BBQ?

Linux Graphics Demystified

Martin Fiedler
Dream Chip Technologies GmbH

Agenda

- Console and Frame Buffer
- X Window System
- OpenGL, Mesa and Gallium3D
- DRI – Direct Rendering Infrastructure
- KMS – Kernel Mode Setting
- Compositing
- Driver Overview
- Other Graphics Systems – Android, Wayland and Mir
- Video Acceleration
- Hybrid Graphics

Console and Frame Buffer

A long, long time ago …

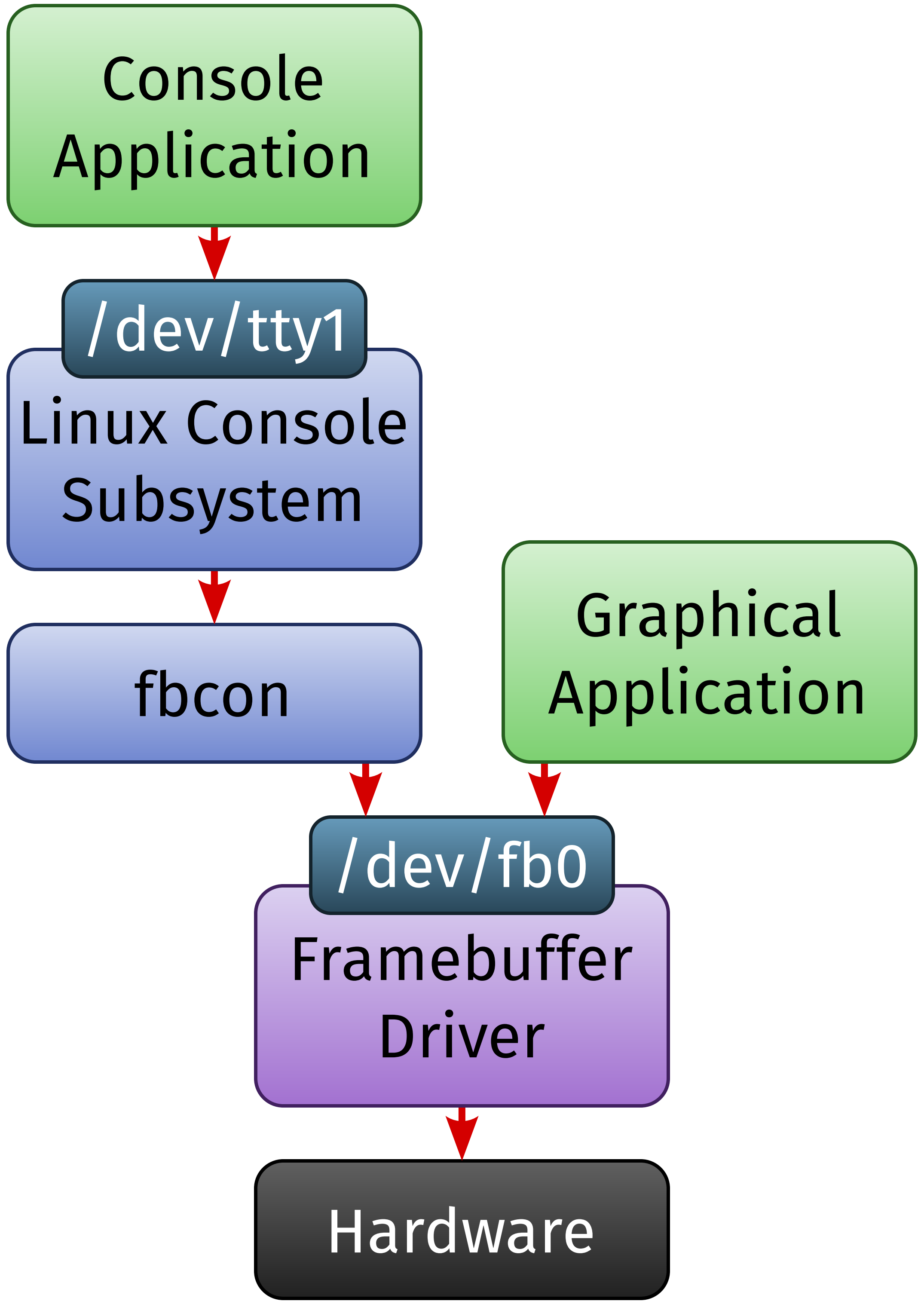
When Linux was first made:

- Linux console used VGA hardware directly
  - … in text mode, of course ☺
- first graphical applications brought their own drivers
- first graphics libraries appeared, e.g. **SVGALib**
- applications are responsible for sustaining the graphics hardware state
  - at start: graphics hardware state is saved
  - at exit: graphics hardware state is restored
  - still valid for the X Server today

Framebuffer Devices

First in-kernel graphics framework: **Framebuffer Devices** (»fbdev«)

- required for porting:
  many platforms don't have a text mode
- hardware-specific kernel drivers
  with common API
  - z.B. `intelfb`, `atifb`
  - `vesafb`: hardware independent,
    uses the VESA BIOS of the graphics card
  - `efifb`: same, but for UEFI
- accessible from userspace: `/dev/fbX`
- very simple API
- **fbcon**: text console emulation
  with bitmapped fonts (and penguins ☺)
  - done in the kernel, not userspace

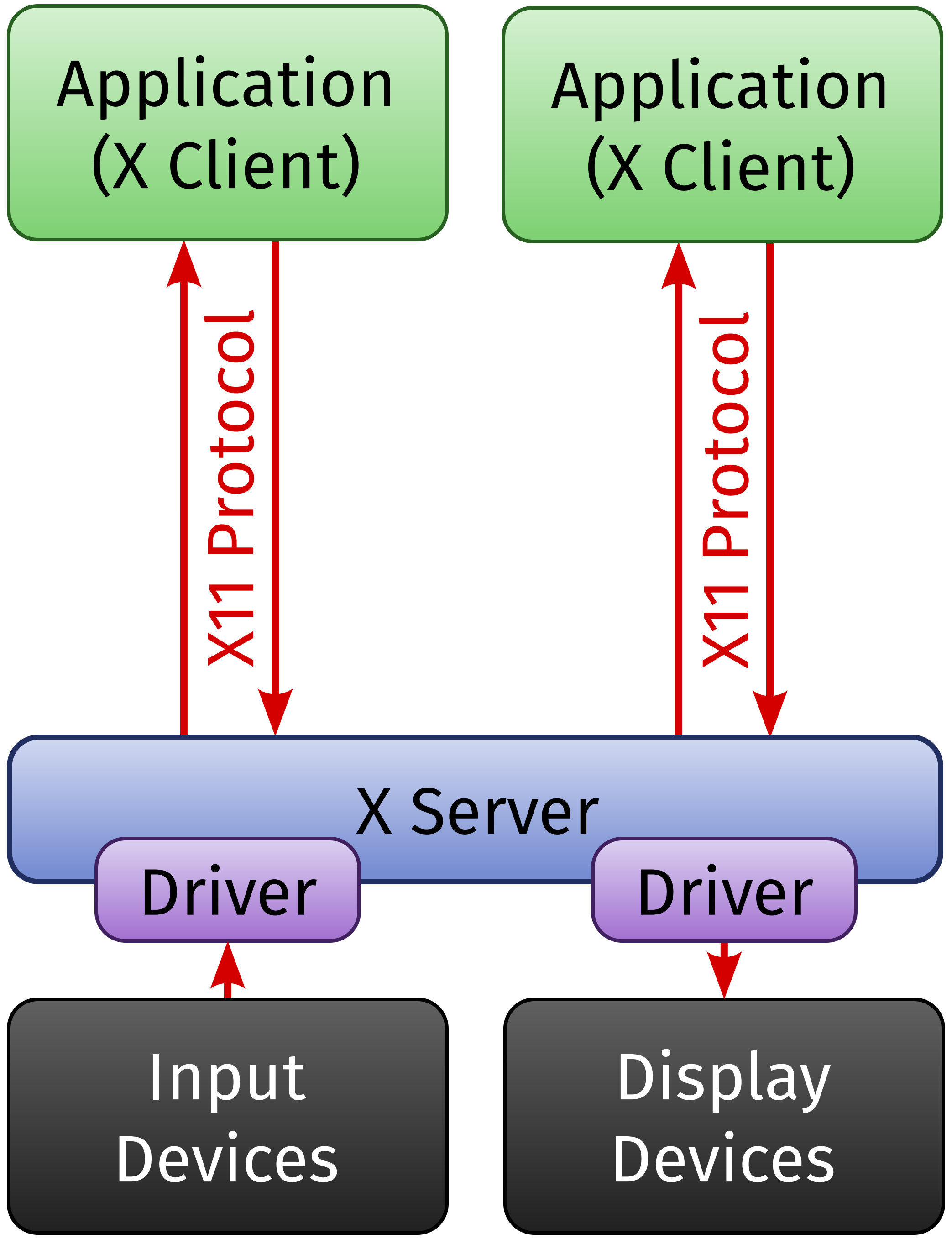


X Window System

X Window System

Most commonly used graphics system on Linux: The **X Window System** (»X11«, »X«)

- popular on all Unix-like systems
- client/server architecture
  - *client* = application
  - *server* manages input and output
- *network transparent*: client and server not required to run on the same machine
  - communication via TCP/IP
  - or locally via Unix Domain Sockets
- **X11** is the name of the *protocol*
- X Server manages a window hierarchy
  - *root window* = desktop wallpaper
  - *top-level windows* = application windows
  - *subwindows* = controls (buttons etc.)



X Clients and Servers

- X Clients don’t implement the X11 protocol directly, but use libraries:
  - traditionally **Xlib**
  - newer, leaner alternative: **XCB** (»X11 C Bindings«)
  - toolkits (Motif, Gtk, Qt, …) internally use Xlib or XCB, too
- *Window Manager*: special X Client that manages the positions of the top-level windows and draws window frames (»decorations«)
- X Server manages input (keyboard, mouse, …) and output (graphics only)
  - generic part: **DIX** (»Device Independent X«)
  - hardware-specific part: **DDX** (»Device Dependent X«)
    - contains drivers for input and output devices
- most popular X Server implementation: **XFree86**, today **X.Org**
  - DDX part is modular: drivers are stand-alone modules
  - DDX interface may change with each version of the server

X Extensions

The X Protocol can be extended with new functionality via **Extensions**. Examples:

- **XSHM** (»X Shared Memory«) – faster local display of bitmap graphics
- **Xv** (»X Video«) – hardware-accelerated video display
- **GLX** – OpenGL on X
- **Xinerama** – multi-monitor support
- **XRandR** (»X Resize and Rotate«) – graphics mode setting without restarting the X Server
- **XRender** – modern anti-aliased, alpha-blended 2D graphics
  - today used for (almost) every 2D graphics application

2D Acceleration in X

Multiple approaches to hardware-accelerated 2D graphics in XFree86 / X.Org:

- **XAA** (»XFree86 Acceleration Architecture«, 1996)
  - simple acceleration of line drawing and fill operations
- **EXA** (2005) – derived from **KAA** (»Kdrive Acceleration Architecture«, 2004)
  - dedicated to XRender acceleration
- **UXA** (»Unified Memory Acceleration Architecture«, 2008)
  - developed by Intel, designated successor to EXA
  - not adopted by non-Intel drivers
- **SNA** (»Sandy Bridge New Acceleration«, 2011)
  - very Intel specific, but also quite fast
- **Glamor** (2011)
  - implements all 2D acceleration via OpenGL
  - result: hardware independent

OpenGL

OpenGL

OpenGL (»Open Graphics Language«) is *the* standard API for 3D graphics.

- industry standard, governed by the »Khronos Group« consortium
- functionality: hardware-accelerated drawing of textured triangles

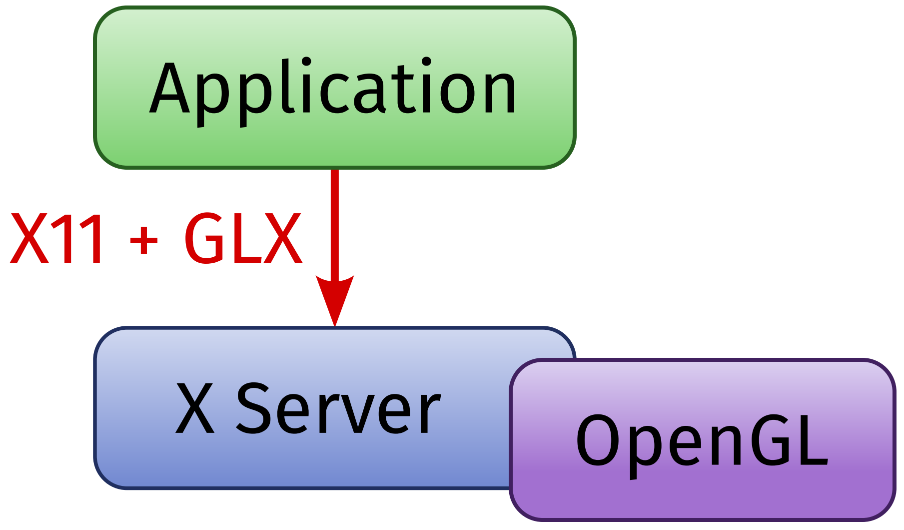
- OpenGL ES = »OpenGL for Embedded Systems«
  - (mostly) a subset of OpenGL, ~90% compatible
- OpenGL (ES) 2.0 and newer feature programmable **shaders**
  - C-like language **GLSL** (»OpenGL Shading Language«)
- extension mechanism (similar to X11)

- additional API required as »glue« to the windowing system:
  - **GLX** for the X Window System
  - WGL (Windows), AGL (Mac OS X)
  - **EGL** for OpenGL ES (Embedded Linux, Android, iOS, ...)
    - available on all systems, will eventually supersede GLX etc.

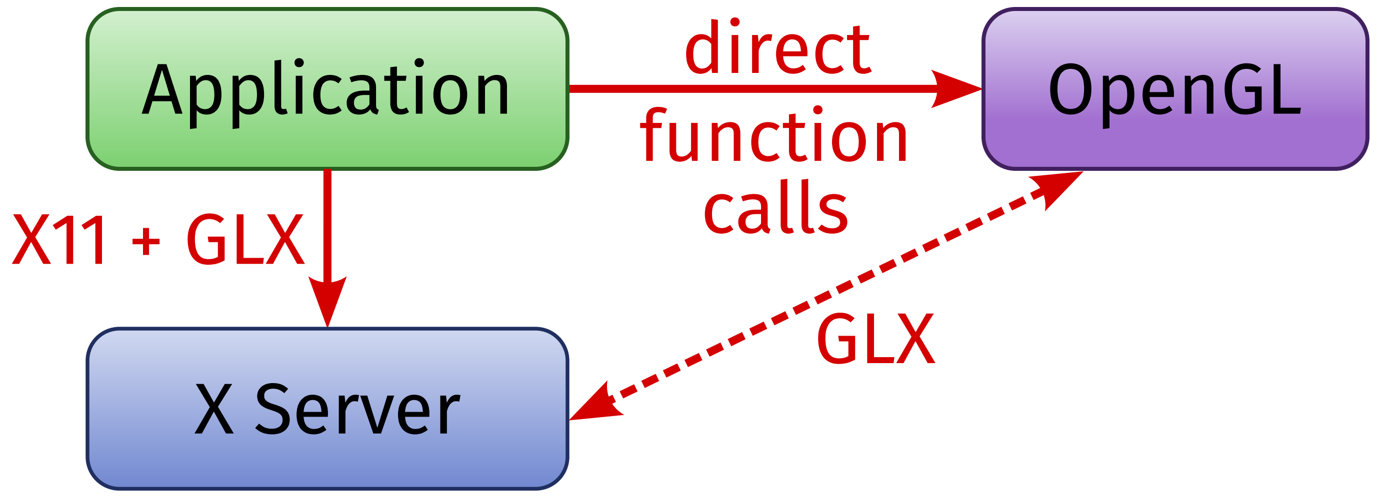
Indirect vs. Direct Rendering

What does OpenGL on Linux with X.Org look like in practice?

- GLX = part of the X protocol
- *Indirect Rendering*
  - OpenGL commands are transferred via the GLX protocol
  - some time ago, this didn't allow for hardware acceleration



- *Direct Rendering*
  - local only (not networked)
  - client liks against `libGL.so` and uses that directly
  - `libGL.so` contains a (possibly hardware-specific) OpenGL implementation



Mesa

There are two kinds of OpenGL implementations on Linux:

- the *proprietary* drivers by nVidia and AMD
- or **Mesa**

Mesa is an open source OpenGL implementation

- … including GLX, EGL and OpenGL ES
- initially only software-rendered
- today it’s the bases for all open source 3D drivers

Gallium3D

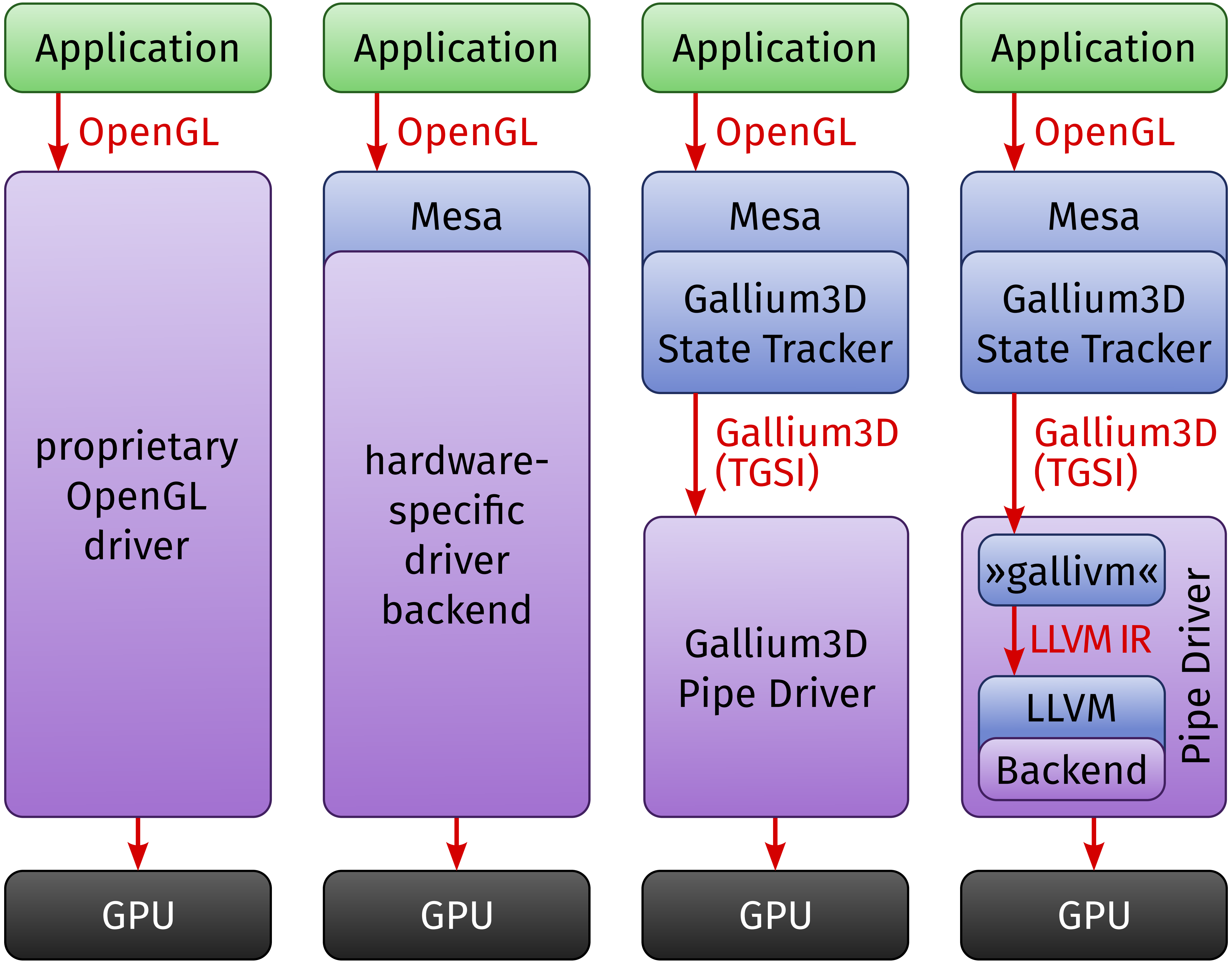
Gallium3D is a framework for implementing GPU drivers in an operating system independent manner.

- partially dependent on Mesa
- not just 3D graphics – also does GPU compute and hardware video decoding
- three basic parts:
  - **State Tracker**: implementation of a client API
    - e.g. OpenGL (via Mesa), OpenCL for compute, VDPAU and OpenMAX for video
  - **WinSys Driver**: implementation of the GLX or EGL layer
  - **Pipe Driver**: backend for a specific GPU
    - e.g. `llvmpipe` (a comparatively fast software renderer)
    - `nv30`, `nv50`, `nvc0`, `nve0` (nVidia GPUs); `r300`, `r600`, `radeonsi` (AMD GPUs)
- uses shader representation **TGSI** (»Tungsten Graphics Shader Infrastructure«)
  - some backends also use LLVM internally

OpenGL Driver Stacks

In total, there are four possible driver stacks for OpenGL:

- proprietary driver
  - replaces `libGL.so`
- »Mesa Classic«
  - generic `libGL.so`
  - hardware-specific backend in Mesa
- Mesa + Gallium3D
  - Mesa as State Tracker
  - Gallium3D backend (TGSI)
- Mesa + Gallium3D + LLVM
  - Mesa as State Tracker
  - Gallium3D backend (LLVM)



OpenCL

- Current GPUs are not just good for graphics
  - contain dozens to thousands of fast floating point compute units
  - **GPGPU** (»General Purpose GPU«) or **Compute** applications
- Standard API for compute: **OpenCL** (»Open Compute Language«)
  - also governed by Khronos Group
  - Linux support works in a similar way to OpenGL:
    - closed source drivers bring their own implementation
    - Gallium3D: state tracker »**Clover**«
    - **Beignet** for Intel GPUs
- other popular compute API: **CUDA**
  - proprietary, nVidia only, only available in closed source drivers

Direct Rendering Infrastructure

DRI & DRM

- OpenGL driver runs in userspace as part of the application process
- access to the graphics hardware is governed by a kernel driver
  - also manages concurrent access from multiple parallel processes
- proprietary graphics drivers have their own proprietary kernel driver APIs
- for open source drivers, there's a common framework:
  the **Direct Rendering Infrastructure** (**DRI**)

- multiple layers:
  - hardware-independent userspace library (`libdrm.so`)
  - hardware- and driver-dependent userspace library (e.g. `libdrm_intel.so`)
  - the kernel module itself: the **Direct Rendering Manager** (**DRM**)
- DRM exports device nodes `/dev/dri/card`*X*
  - but: interface between `libdrm_`*XXX*`.so` and DRM is partially driver-dependent

DRI Versions

There are three major generations of the DRI:

- DRI 1 (1998)
  - first, limited implementation
  - rather inefficient if more than one application wanted to use the 3D hardware
- DRI 2 (2007)
  - solves the most serious problems of DRI 1
  - the current, most widely deployed version
- DRI 3 (2014?)
  - many detail improvements
  - currently in development

If not mentioned otherwise, the following slides refer to DRI 2.

DRM Master and Render Nodes

DRM clients are not equal – there is a »**DRM Master**«

- typically the X Server
- runs as root
- manages the GPU alone
  - there's always just one DRM Master per GPU
- can authorize other processes to use the GPU

- Problem: can't use the GPU without an X Server
  - annoying for compute applications
- Solution: **Render Nodes** in DRI 3
  - `/dev/dri/renderDXX`
  - limited functionality – no graphics output
  - no authorization by the DRM Master required

Memory Management and Buffer Sharing

A major task of the DRI is managing graphics memory.

- Intel drivers use **GEM** (»Graphics Execution Manager«) for this
- most other drivers use the GEM API, but a different implementation beneath: **TTM** (»Translation Table Manager«)
- most important feature: passing and sharing graphics buffers across process boundaries
  - essential for compositing (»3D desktops« like Compiz)
- with GEM: **flink** API
  - global numerical IDs for shared buffers
  - security issue: IDs are easily guessable
- newer, more secure sharing API since Linux 3.3: **DMA-Buf**
  - buffers are identified by file descriptors
  - file descriptors can be transferred in a secure way via Unix Domain Sockets

Kernel Mode Setting

Issues with User Mode Setting

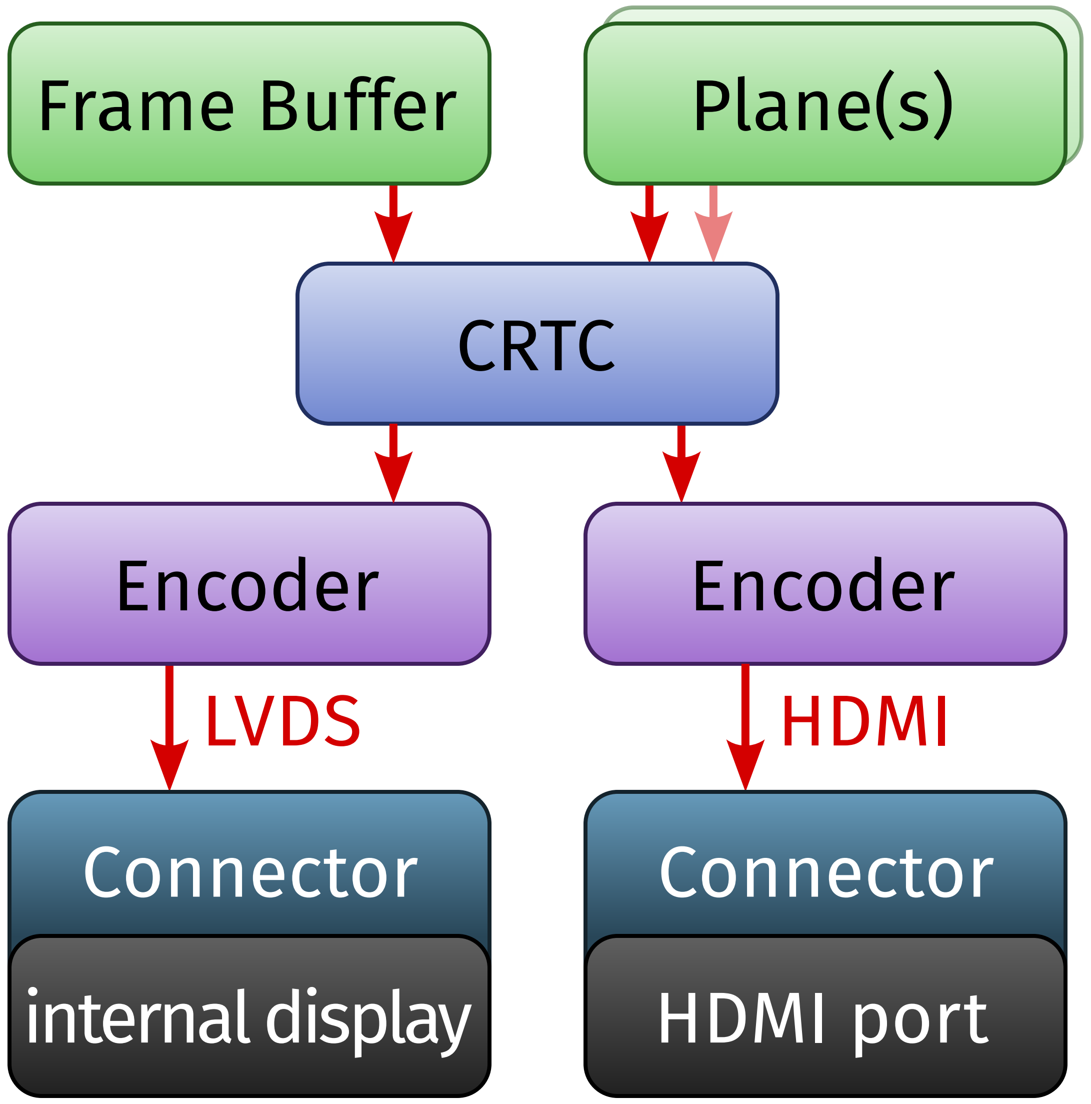
Classic graphics mode setting (»User Mode-Setting«) is problematic:

- hardware is being initialized multiple times
  - first by the BIOS for its boot messages …
  - … then by the framebuffer driver for the boot console …
  - … and finally by the X Server
- flickers during boot
- flickers when changing between virtual consoles and X Server instances
- duplicated driver code
  - framebuffer driver and DDX mostly do the same things
- issues with suspend and resume
- VESA framebuffer driver can’t reliably detect the display resolution
  - uses some arbitrary default resolution
  - result: boot messages look blurry ☹

Kernel Mode Setting

Solution: **Kernel Mode Setting** (**KMS**)

- a *single* driver in the kernel, used by the framebuffer *and* the X Server
- subsystem of the DRI
  - no new device nodes
- flexible display concepts, leverages the possibilities of modern display controllers:
  - *Frame Buffer*
  - *Plane* = overlay
  - *CRTC* = display controller
  - *Encoder*, e.g. HDMI transmitter
  - *Connector* = physical port or display
- Frame Buffers and Planes are DRI buffers



(Example)

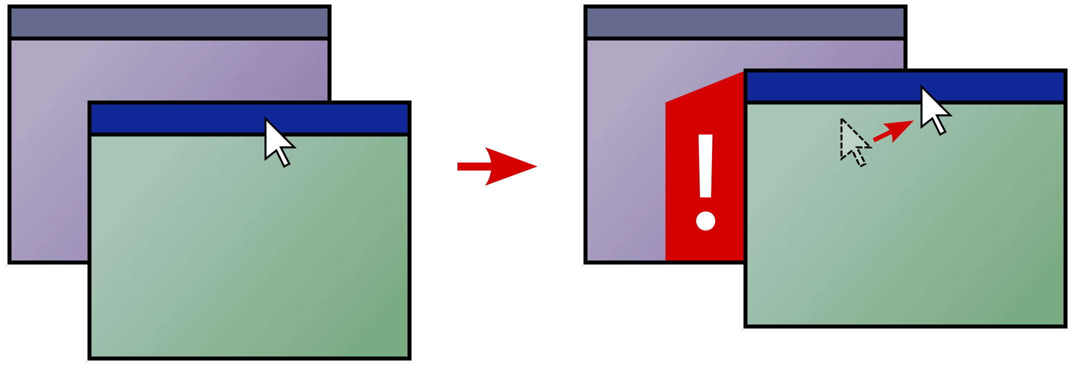
KMS: Outlook

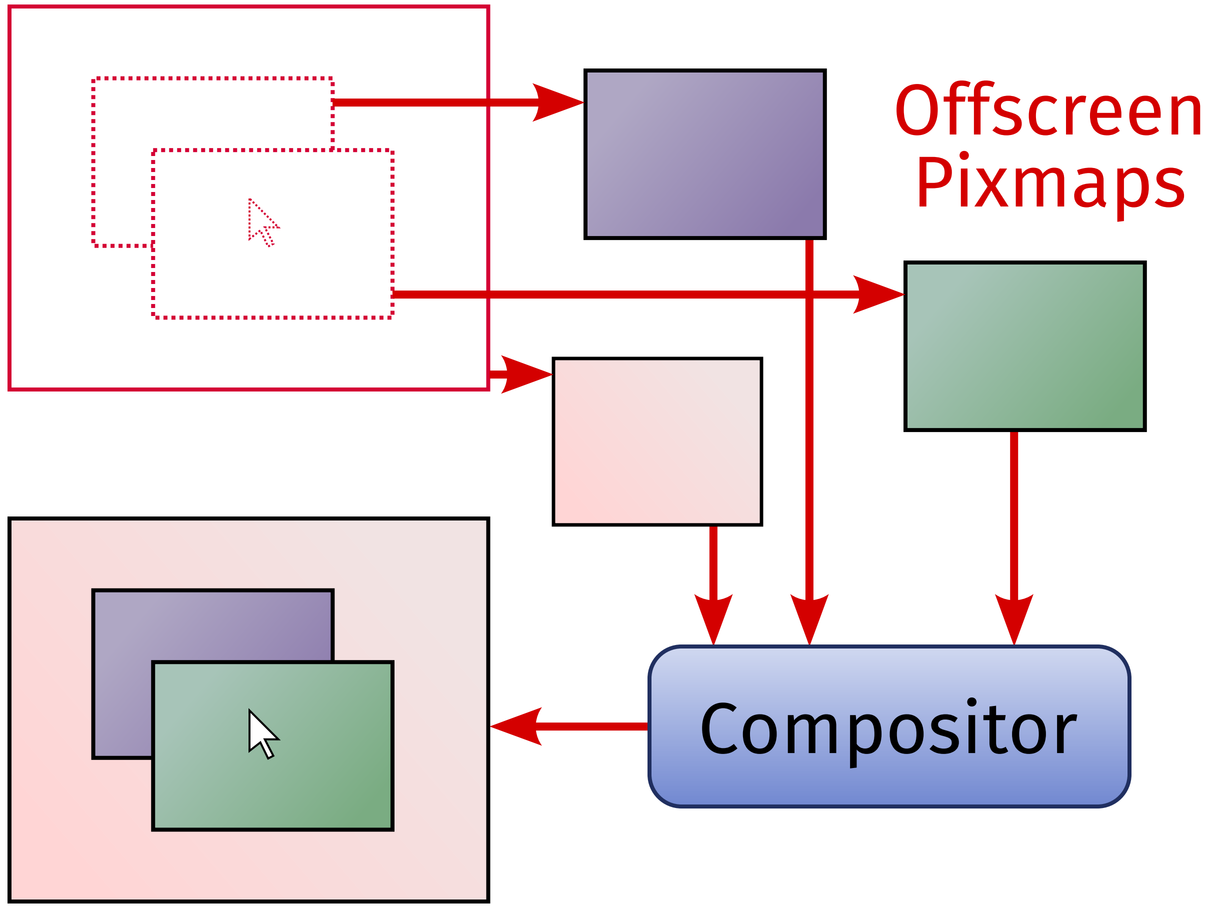
- `xf86-video-modesetting`: *hardware-independent* DDX driver for X.Org, based on KMS and Glamor
- **KMSCON**: replacement of the Linux kernel's framebuffer console layer with a proper, fully featured terminal emulation in userspace
  - hardware acceleration, multiple monitors, full Unicode support, anti-aliasing, …
- Further development of KMS: **ADF** (»Atomic Display Framework«)
  - useful for hardware with multiple overlay planes
    - standard feature on embedded and mobile devices
  - settings of all overlays can be modified synchronously (»atomically«)
    - prevents flickering and tearing

Compositing

Compositing

- normal X11 windows are »lossy«
  - have to be redrawn if areas that have been occluded by other windows are exposed
- alternative: **redirection**
  - window isn't drawn directly to the screen, but »off-screen« into a so-called **pixmap**
  - input handling continues to work as usual (i.e. as if the window was drawn on-screen)
- **compositor** finally draws the off-screen pixmaps at the correct locations
  - only one »real« window without redirection: the *Compositor Root Window*
- compositor commonly integrated into the window manager
- **unredirection** = suspension of redirection for full-screen windows

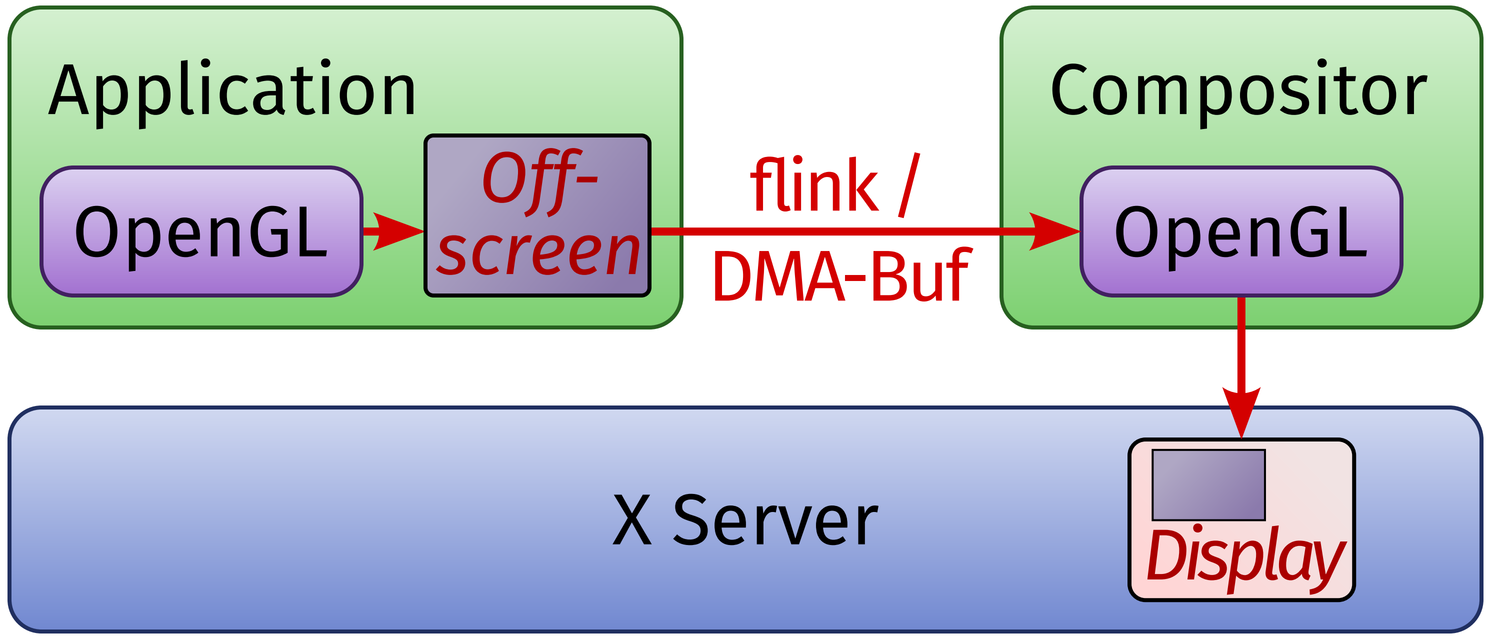




Compositing and OpenGL

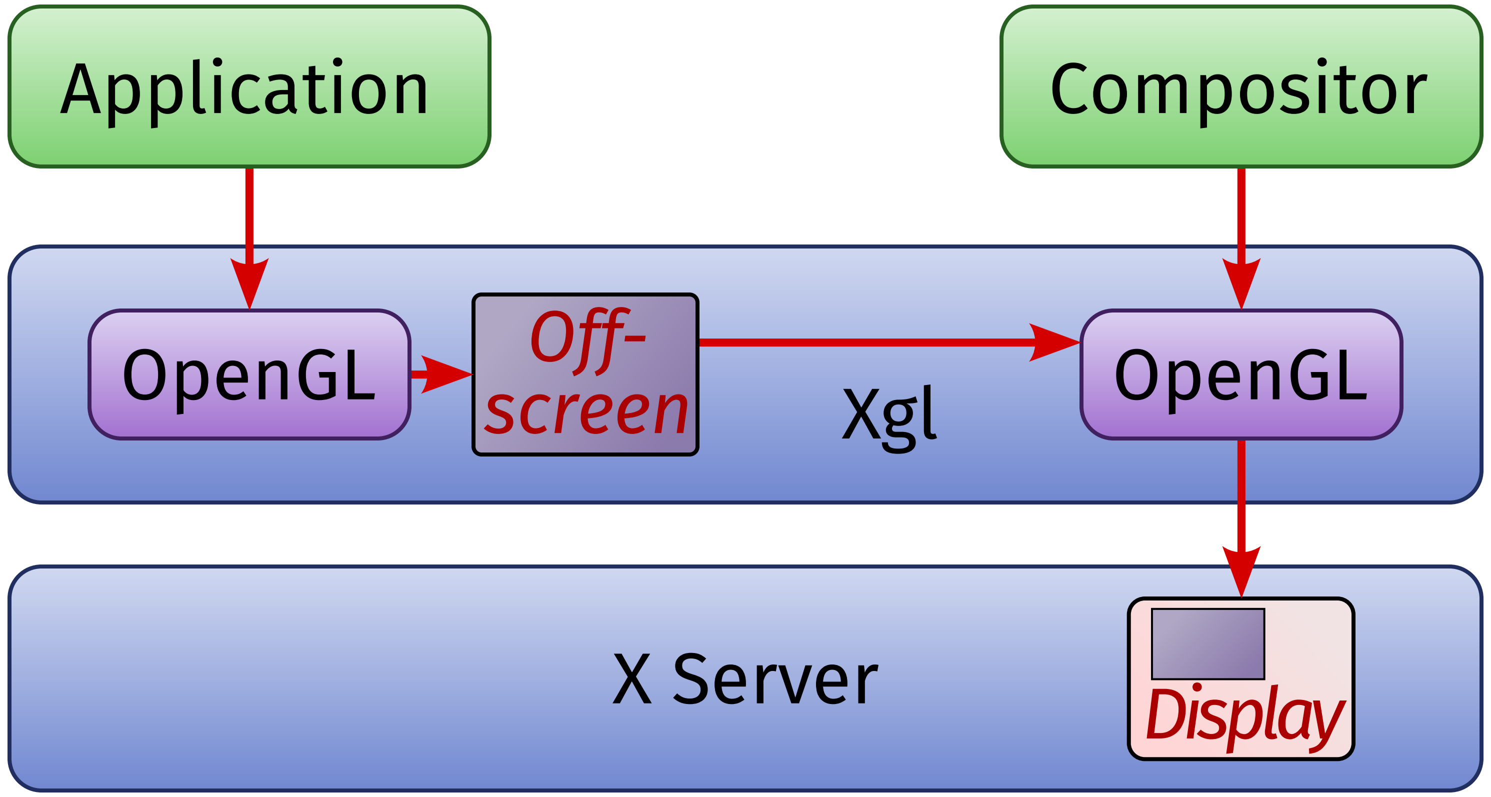
Compositing is particularly interesting in combination with OpenGL for »3D desktops« like Compiz.

- but: OpenGL »doesn't know« X11 pixmaps, just its own textures and framebuffers
- *Problem 1:* compositor has to access pixmaps as OpenGL textures for drawing
  - Solution: extension `GLX_EXT_texture_from_pixmap`
- *Problem 2:* compositor requires access to framebuffers of other processes' OpenGL contexts
  - today, that's easy to do with DRI buffer sharing
    - every OpenGL framebuffer is a DRI buffer
    - compositor uses these DRI buffers as OpenGL textures



Early solution attempt for the OpenGL compositing problem: **Xgl**

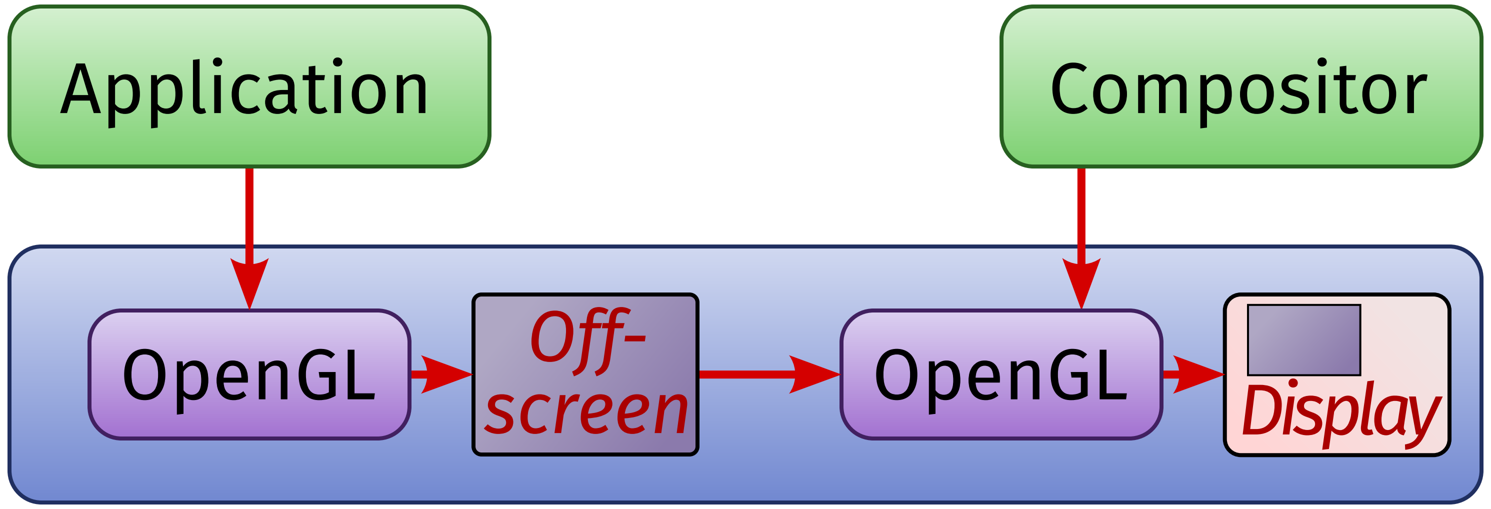
- Xgl = special »virtual« X Server
- draws everything with OpenGL
  - for classic X applications: using the **glitz** library (a predecessor of Glamor)
  - for OpenGL applications: by enforcing indirect rendering
    - all OpenGL commands go through the Xgl server
    - ... who redirects the output into OpenGL Frame Buffer Objects
  - this way, the server can give the compositor access to all windows' contents
- Xgl itself runs on another, »real« X Server



AIGLX

Other early approach to the OpenGL compositing problem: **AIGLX** (»Accelerated Indirect GLX«)

- enables hardware accelerated indirect rendering for OpenGL
- actually, it *enforces* indirect rendering:
  - all real OpenGL rendering happens in the X Server
  - output is redirected into OpenGL Frame Buffer Objects
- this way, the server can give the compositor access to all windows' contents



Driver Overview

Drivers for PC Graphics Hardware

- Drivers for DRI, X.Org (DDX), Mesa and Gallium3D often have different names
- »mix-and-match« possible in some cases

- for unsupported hardware
  - using the VESA BIOS or UEFI firmware for mode setting
  - software-rendered OpenGL
    - earlier: Mesa's software renderer – extremely slow
    - today: Gallium3D `llvmpipe` – generates machine code, considerably faster

- **Intel** integrated graphics
  - excellent driver support, exclusively open source
  - old-fashioned – no Gallium3D
    - experimental Gallium3D pipe driver »ILO«
    - official drivers are »Classic Mesa«

Drivers for PC Graphics Hardware

- **ATI** / **AMD** GPUs (integrated or dedicated)
  - proprietary closed source driver: **fglrx**
  - AMD publicly documents their hardware → good open source driver support
  - **radeon** driver family: Mesa for Radeon 7000 – 9250, Gallium3D from Radeon 9500
  - **radeonhd** driver family: Mesa for Radeon X1000 – HD4000, not developed further
- **nVidia** GPUs
  - proprietary closed source driver: **nvidia**
  - no hardware documentation → open source drivers rely on reverse engineering
  - **nv** driver: very old open source 2D driver for Riva 128 and older GeForces
  - **nouveau** driver family: Gallium3D, from GeForce FX upwards
  - **nouveau_vieux** driver family: Mesa, Riva TNT to GeForce 4

Typical Driver Stacks on the PC

Driver	Fallback	Intel	AMD		nVidia	
Framebuffer	`vesafb` / `efifb`	KMS	`vesafb`	KMS	`vesafb`	KMS
DRM/Kernel	—	`i915`	`fglrx`	`radeon`	`nvidia`	`nouveau`
X.Org DDX	`fbdev` / `vesa`	`intel`	`fglrx`	`radeon`	`nvidia`	`nouveau`
2D Accel.	—	UXA / SNA	propri-etary	EXA / Glamor	propri-etary	EXA
OpenGL	Mesa	Mesa	fglrx	Mesa	nVidia	Mesa
Mesa	Gallium3D	`i915` / `i965`	—	Gallium3D	—	Gallium3D
Gallium3D	`llvmpipe`	—	—	`r300` / `r600` / `radeonsi`	—	`nv30` / `nv50` / `nvc0` / `nve0`
OpenCL	Gallium3D	Beignet	fglrx	Gallium3D	nVidia	Gallium3D

Drivers for Embedded GPUs

The driver situation for GPUs in smartphones, tablets etc. is much more complicated.

- GPU-, SoC- and device manufacturers deliver closed source drivers only
  - usually appalling quality, lots of bugs
  - sometimes not even the kernel drivers are available as source code
  - sometimes even distribution of the binary blob is forbidden
- exception: Broadcom VideoCore IV (e.g. Raspberry Pi)
  - documentation and driver source code published in February 2014

Several approaches to develop open source drivers via reverse engineering:

- Qualcomm Adreno – **Freedreno**
- ARM Mali – **Lima**
- Vivante – **Etna_viv**
- nVidia Tegra – **Grate**
- Imagination Technologies PowerVR – ???

Other Graphics Systems

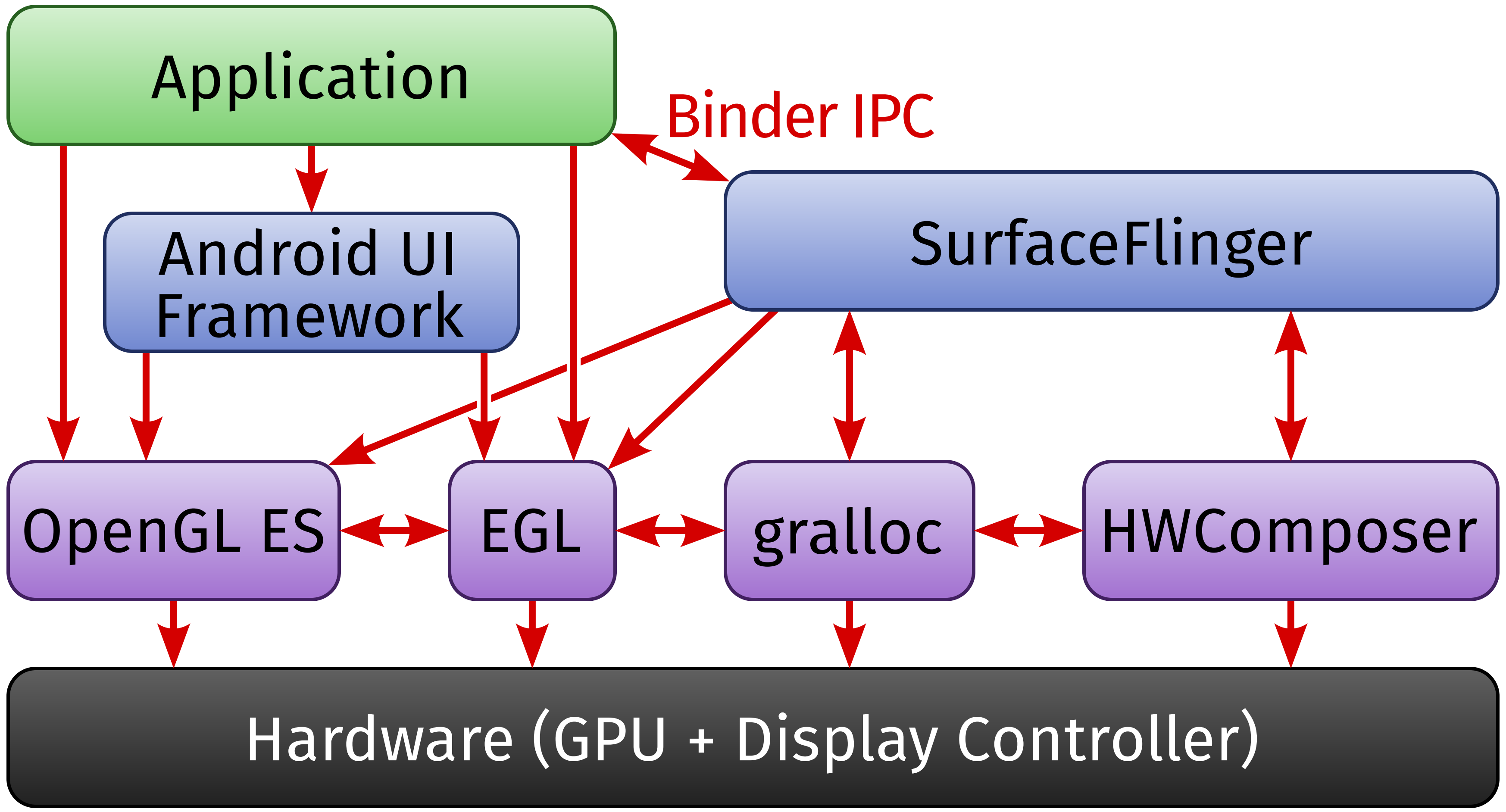
What else is out there?

Until now, we've been talking about the X Window System only,
but there are other graphics systems.

- the basic *concepts* are always similar, though
- Example: **DirectFB**
  - developed for embedded systems (set top boxes) in 1997
    - goal: graphics system with lower resource footprint than X
  - based on Linux' framebuffer devices
    - additional hardware drivers for acceleration
  - central library: `libdirectfb`
    - manages graphics and sound output as well as input
  - own window manager, ports of common toolkits, X compatibility (using a special X Server), …
  - nevertheless: not relevant for normal »desktop« systems

Android

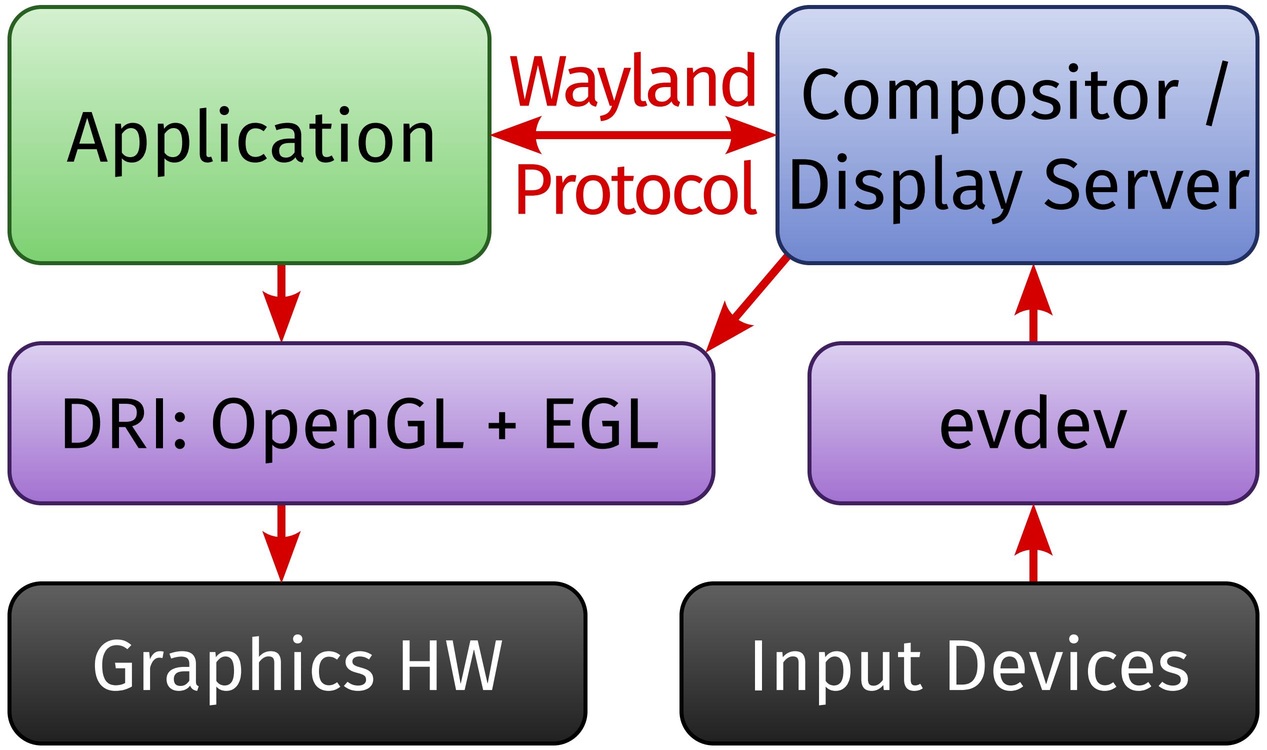
- Android uses the Linux kernel, but not much more
  - no GNU userland, no X
  - custom C library: **Bionic**
  - custom IPC mechanism: **Binder**
- graphics based on OpenGL ES and EGL
  - no DRI (mostly proprietary drivers)
- hardware-specific **HWComposer** library as rough equivalent of KMS
- **gralloc** for graphics memory management
  - part of HWComposer in newer versions
- compositor and display server: **SurfaceFlinger**
- SurfaceFlinger also allocates graphics buffers for applications



Wayland

So far the most promising candidate for replacing the X Window System: **Wayland**

- goal: radical simplification of X's concepts
- technically, it's a *protocol*
  - using Unix Domain Sockets
  - *not* network transparent
- server part is not a program of its own, but a library
  - used by the compositor
    → the compositor *is* the display server
  - reference implementation: **Weston**
- based on EGL and DRI
- buffer allocation and drawing completely done in the clients
- input devices are used via the kernel's event device framework



XWayland und Hybris

How can X applications be run on a Wayland system?

- **XWayland** = modified »rootless« X.Org Server
  that turns all top-level X windows into Wayland clients
- still requires hardware-specific DDX drivers, exceptions:
  - `xf86-video-wlshm` (hardware-independent, but not accelerated)
  - `xf86-video-wlglamor` (with 2D acceleration via Glamor)

Wayland can work on Android graphics drivers using **libhybris**:

- libhybris »mediates« between the GNU libc world and the Bionic world
  - libc applications can use Bionic libraries
  - in particular, they can use `libGLESv2.so`, the OpenGL ES driver
- also adapts some other Android peculiarities (e.g. gralloc, EGL differences)

Mir

Competition for Wayland: **Mir** by Canonical

- graphics system for upcoming Ubuntu versions
  - not yet in 14.04, but maybe in 14.10
- conceptually very closely related to Wayland,
  but a totally different and incompatible implementation
- uses more parts of Android, e.g. the input subsystem
- more focus on data exchange between applications
- graphics buffers are allocated in the server, but drawn in the client
- **XMir** = XWayland for Mir
- also employs libhybris for Android graphics driver support
- much resistance in the community
  - it's doubtfull whether another system is really necessary

Video Acceleration

Video Acceleration

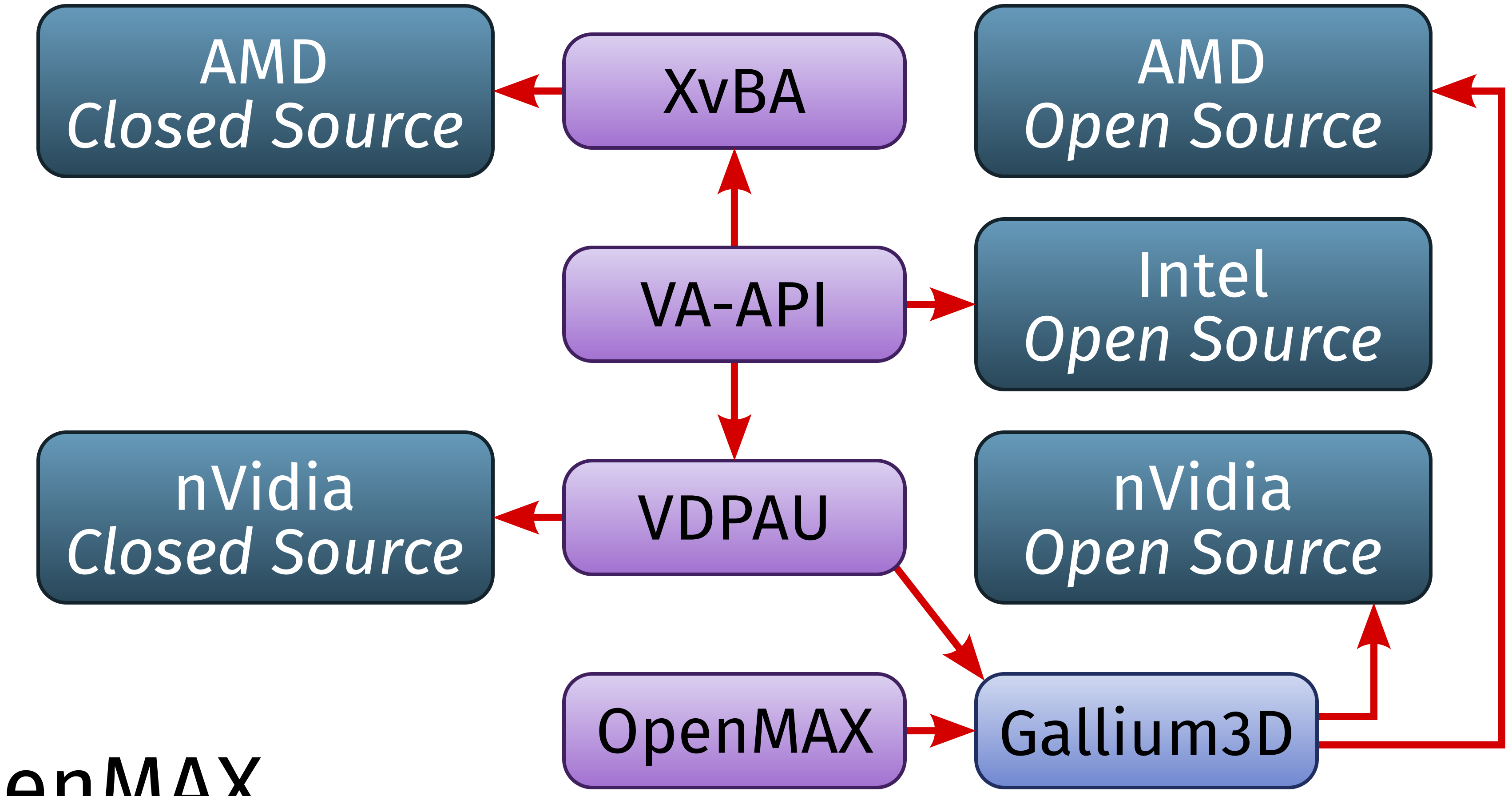
There are multiple approaches for hardware-accelerated video on X:

- **Xv** (X extension, 1991)
  - only for video *output*, not decoding
  - functionality: scaling, color space conversion
  - two typical kinds of implementation (can be mixed):
    - **Overlay**: directly overlays the video into the display output
    - **Textured Video**: draws the video into the framebuffer using the 3D hardware
- **XvMC** (X extension, 2000)
  - accelerates two specific aspects of MPEG-2 decoding:
    Motion Compensation (»MC«) and *IDCT* (8×8 block transform)
  - obsolete
    - specific to MPEG-2, never adapted to newer standards
    - supported by very few drivers only

Hardware Decoding

Current GPUs contain *hardware decoders* for the common standards (e.g. H.264).

- multiple incompatible APIs:
  - nVidia proprietary: **VDPAU** (»Video Decode and Presentation API for Unix«)
    - full-featured: decoding, display, deinterlacing, …
  - AMD proprietary: **XvBA** (»Xv Bitstream Acceleration«)
    - decoding only, display via OpenGL
  - Intel: **VA-API** (»Video Acceleration API«)
    - decoding into DRI buffers
  - embedded playforms: **OpenMAX**
    - industry standard for de- and encoding
- Situation improves slowly:
  - VA-API backends for VDPAU and XvBA
  - Gallium3D State Tracker for VDPAU and OpenMAX
  - Gallium3D backends for nVidia's und AMD's hardware decoders



Hybrid Graphics

Hybrid Graphics

- Many current notebooks have *two* GPUs:
  - processor-integrated graphics – slow, but saves power
  - additional (»dedicated«) nVidia or AMD GPU – fast, but inefficient
- **vga_switcheroo**: deactivates one of the GPUs
  - switching GPUs requires restarting the X Server
  - only works on systems with »Video Mux«
    where both GPUs can drive all displays
  - Problem: newer models are usually »muxless«
- by now, proprietary drivers by AMD and nVidia have their own switchers
  - based on XRandR 1.4 (`xrandr --setprovideroutputsource`)
  - work on »muxless« systems too
  - but: the dedicated GPU's output is copied over to the integrated GPU
    - not saving power (quite the contrary – both GPUs are active!)

Bumblebee and PRIME

For nVidia-based hybrid systems (»Optimus«) with the proprietery driver, there is a »real« hybrid graphics solution: **Bumblebee**

- initially, only the integrated GPU runs
- if an application is run using a special wrapper (`optirun`):
  - the dedicated GPU is activated
  - a second (invisible) X Server running on the nVidia driver is started
  - all OpenGL drawing commands are redirected to that second X Server via **primus**
  - after every frame, the final image is copied back to the integrated GPU's X Server

open source solution: **PRIME**

- currently in development
- extends the DMA-Buf APIs for cross-GPU buffer sharing
- fully dynamic »offloading« of rendering operations
- activated with `xrandr --setprovideroffloadsink`

Thank You!