An Overview of the Linux and Userspace Graphics Stack

Paul Kocialkowski
paul@bootlin.com

Paul Kocialkowski

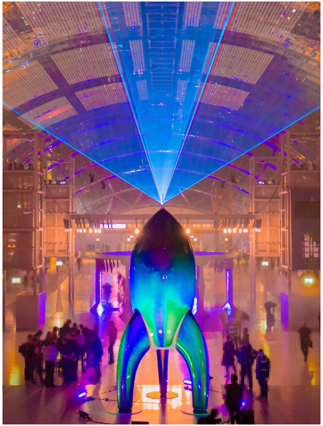
- Embedded Linux engineer at Bootlin
  - Embedded Linux **expertise**
  - **Development**, consulting and training
  - Strong open-source focus
- Open-source contributor
  - Co-maintainer of the **cedrus** VPU driver in V4L2
  - Contributor to the **sun4i-drm** DRM driver
  - Developed the **displaying and rendering graphics with Linux** training
  - Contributing **Allwinner MIPI CSI-2** support in V4L2
- Living in **Toulouse**, south-west of France

Introduction

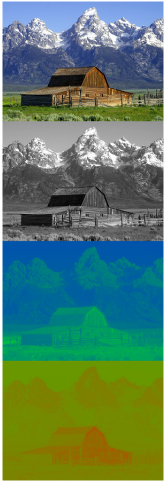
What All the Fuss is About with Graphics

What do we mean by graphics?

- Graphics deals with digital representation of **light**
- Taking the form of **pictures** or **frames**
- Light in the physical world is continuous
- Digital pictures are **discrete** or **quantized**
- Discrete picture elements are **pixels**
- Using a **color model** and **color space**



All About Pictures and Pixels



- Pictures have **dimensions** (width and height) in pixels
- **Aspect ratio** is the width:height fraction
- **Resolution** links pixels to length units (px/in)
- Specified **scan order** in memory
- Pixels have a specific **format**:
  - Color channels in a color space
  - Alpha (transparency) channel
  - Depth and bits per pixel (bpp)
  - Organization in memory as planes
  - Sub-sampling

All the Things Dealing with Pixels

Graphics:

- **Displaying**: producing light from a digital picture
  - Monitors, panels, projectors
- **Rendering**: generating digital pictures from primitives
  - 3D rendering, 2D shape drawing, font rendering and more
- **Processing**: transforming digital pictures
  - Filtering, scaling, converting, compositing and more

Media:

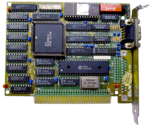
- **Decoding/encoding**: (un)compressing pictures
  - Picture codecs (JPEG, PNG, etc), Video codecs (H.264, VP8, etc)
- **Capturing/outputting**: receiving or sending pictures from/to the outside
  - Cameras, DVB and more

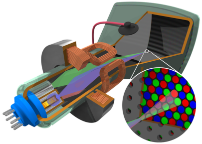
Graphics Hardware Components

Display Hardware (Source)

Display output is implemented through many components:

- **Framebuffer**: pixel memory
- **Display engine**: hardware compositor (from planes)
- **Timings controller**: CRT-compatible timings
- **Display protocol controller**: protocol logic
- **Display interface PHY**: protocol physical layer
- **Connector and cable**: link to the display device (monitor or panel)





Rendering and Processing Hardware

A few types of implementations are used for **rendering** and **processing**:

- **GPUs** (Graphics Processing Units):
  - Highly specialized architectures and ISAs
  - Designed for 3D rendering, can also do 2D and processing
  - Loaded with small programs (shaders)
  - Configured with a command stream
- **DSPs** (Digital Signal Processors):
  - Dedicated processors with a specialized ISA
  - Run with a dedicated firmware or RTOS
- **Fixed-function ISPs** (Image Signal Processors):
  - Hardware implementations for specific tasks
- **CPU-based** implementations
  - All done in software (often use SIMD)

Generic Concepts for Software

Display Software Concepts

- Display access must be **exclusive** to a single program
- A **display server** manages the display framebuffer(s):
  - Provides a protocol for clients
  - Gathers buffers and updates from clients
  - Handles input events
  - Forwards relevant input events to clients
  - In charge of security and isolation concerns
- A **compositor** produces the final image from client sources
- A **window manager** defines policy between clients

Render Software Concepts

- Unlike display, GPUs can run different **jobs in parallel**
- GPU rendering is based on **primitives**:
  - Vertices, lines and triangles
  - Given positions in 3D
- **Textures and maps** can be involved as well
- Shaders (programs) define the result:
  - **Vertex shaders** for transformations and lighting
  - **Fragment shaders** for color and textures
  - More advanced shaders also exist
  - Shaders are **compiled on-the-fly** from source or intermediate forms (IRs)
  - Binary shaders are attached to the GPU **command stream**
- Multiple **rendering passes** can be used

Linux and Userspace Graphics Stack

Displaying Stack: Kernel

- Historic and legacy subsystem for display: **fbdev**
  - Very limited: static setup, no pipeline, pre-allocated buffers, sync issues
  - Still used by fbcon for on-display console
  - Available through `/dev/fb0` (please stop using it)
  - Source at `drivers/video/fbdev` in Linux
- Current and relevant interface: **DRM KMS**
  - Exposes each display pipeline element for configuration
  - Generic uAPI with a property-based system
  - Dynamic framebuffer allocation
  - Atomic API for synchronizing changes
  - Legacy compatibility layer with fbdev
  - Source at `drivers/gpu/drm` in Linux
- The **TTY** subsystem is also involved:
  - Graphics mode switch to detach fbcon
  - Virtual Terminal (VT) switching

Displaying Stack: Userspace Protocols and Servers

- **X** is the historical display server project used with Linux:
  - **X11** (X protocol version 11) is the protocol
  - Completed with many (many) **protocol extensions**:
    e.g. `XrandR`, `XSHM`, `Xinput2`, `Composite`
  - **Xorg** is the reference implementation
  - Using hardware-specific **drivers** for display and input:
    e.g. `xf86-input-libinput`, `xf86-video-modesetting`, `xf86-video-fbdev`
  - X provides server-side rendering (not used a lot nowadays)
  - Comes with various **security issues** and **limitations**
  - No longer adapted to modern-day computers, nor embedded
  - Most of its work is delegated through extensions

Displaying Stack: Userspace Protocols and Servers

- **Wayland** is a modern display server:
  - Designed from scratch to avoid common limitations and issues in X
  - Wayland is a **protocol specification**, not an implementation coming as a core protocol and optional extensions (e.g. `XDG-Shell`)
  - Server implementations are called **Wayland compositors**
  - **Weston** is the Wayland compositor reference implementation
  - Other implementations: `sway` (wlroots), `mutter` (GNOME), `Kwin` (KDE)
  - Improved security and isolation between clients
  - Compatible with X applications via **XWayland**
- Other display servers also exist:
  - **Mir**: Canonical's display server, more or less abandoned
  - **SurfaceFlinger**: Android's display server
- **Display managers** are commonly used at startup:
  - Serve as login screens at startup
  - Launch display servers and environments for users

Displaying Stack: Userspace Libraries

- Libraries implement low-level display server protocols:
  - **Wayland**: `libwayland-client`, `libwayland-server`
  - **X11**: `XCB`, `Xlib`
- Graphics toolkits abstract display server protocols:
  - **GTK** (C): Widget-based UI toolkit for X and Wayland
  - **Qt** (C++): Widget-based UI toolkit for X and Wayland
  - **EFL** (C): Lightweight UI and application library
  - **SDL** (C): Drawing-oriented graphics library (used in games)
- **Desktop environments** are based on a given toolkit:
  - Provide a desktop UI and a set of base applications
  - Implement a window manager/compositor
- Calls to the **DRM uAPI** are wrapped by `libdrm`
  - Used by every single program that supports DRM

Rendering Stack for 3D: Kernel

- The DRM subsystem is also in charge of **managing GPUs**
- Unlike DRM KMS, no generic interface but **driver-specific uAPIs** supported with thin helpers in `libdrm`
- Handles various **low-level aspects**:
  - **Memory buffers** (BO) management with `GEM`
  - **Command stream** validation and submission
  - **Tasks** scheduling with the DRM scheduler
- Most of the heavy lifting is left to userspace (only I/O in kernel)
- **Proprietary implementations** use their custom interfaces found in downstream kernels or out-of-tree drivers

Rendering Stack for 3D: Userspace APIs

Generic APIs are used for programs to leverage the GPU:

- **OpenGL** for rendering with desktop GPUs:
  - Compromise between complexity and control
  - Stateful and context-based programming model
  - Using GLSL (GL shading language) for shader sources
- **OpenGL ES** for rendering with embedded GPUs
- **EGL** for interfacing OpenGL with the display stack
  - Provides scanout buffers and sync
  - Supports X11, Wayland, Android and more
  - Replaces the legacy GLX for X11
- **Vulkan** for advanced GPU usage:
  - Low-level API with direct programming and memory management
  - Uses its own display stack integration: Vulkan WSI
  - Supports more than rendering (e.g. compute)
  - Uses a pre-built shader format: SPIR-V

Rendering Stack for 3D: Userspace Implementations

- **Mesa 3D** is the reference free software rendering library:
  - Supports OpenGL, OpenGL ES and Vulkan APIs (also Direct 3D 9)
  - Supports GPUs that have a DRM render driver:
    `radeon`, `amdgpu`, `nouveau`, `etnaviv`, `vc4/v3d`, `lima`, `panfrost`
  - Implements software rendering fallbacks:
    `softpipe`, `swr`, `llvmpipe`, `lavapipe`
  - Implements shader compilation with intermediate representations (IRs)
  - Also supports GPU video decoding through `VDPAU`, `VAAPI` or `OMX`
  - Also supports compute via OpenCL (`clover` driver)
- **Proprietary** libraries have their own secret implementations

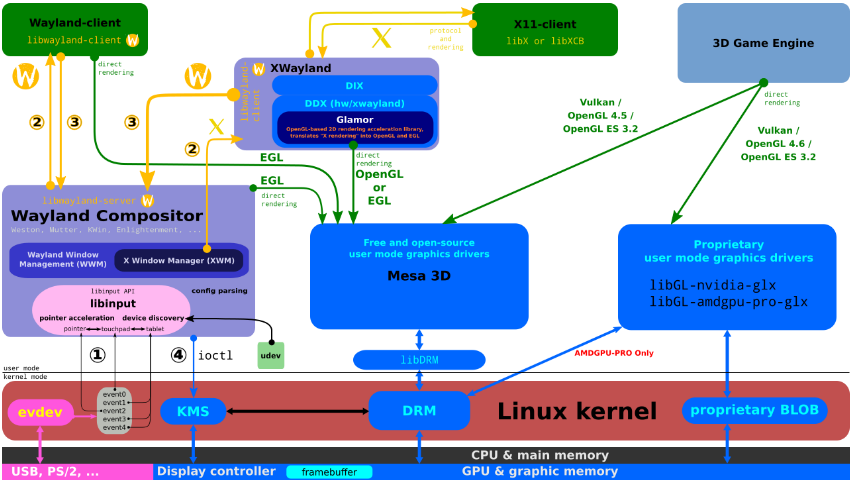
Rendering Stack for 2D: Libraries

- General **drawing/rasterization**:
  - **cairo**: widely-used drawing library
  - **Skia**: Google's drawing library
- **Font** rendering:
  - **FreeType**: historical vector font rendering library
  - **HarfBuzz**: recent vector font rendering library
- **User interface** rendering:
  - Full widget toolkits: **GTK**, **Qt**, **EFL** and more
  - Immediate-mode GUIs: **Dear ImGui**, **nuklear**
  - Animations: **Clutter**
- Mostly **CPU-based** implementations
- Sometimes leverage **GPU rendering** through 3D APIs and shaders

Processing Stack: Libraries

- Processing can be implemented:
  - Using optimized **CPU-based** algorithms
  - Using specific **SIMD** CPU instructions (NEON, SSE, AVX)
  - Using GPU rendering through 3D APIs and shaders
- Various libraries exist:
  - FFmpeg's **libswscale** for pixel format conversion and scaling
  - **Pixman** for various pixel operations
  - ARM's **Ne10** for NEON-accelerated pixel operations
  - **FFTW** for fast Fourier transforms
  - **G'MIC** image processing framework

Graphics Stack Overview



Advanced Topics: Memory Sharing and Fences

- Copying buffers between (hardware) components is a **major bottleneck**
- Specific APIs are used to share references (file descriptors) between applications:
  - **Shared memory** (SHMem) for system memory pages
  - **DMA-BUF memory** for device-allocated memory
- Synchronization between hardware devices is possible with **fences**:
  - A graphics pipeline is configured with fence references (file descriptors)
  - Fences are **signaled** when a device is done
  - The next device in the chain is then **triggered** by the kernel
  - No userspace roundtrip is necessary

Questions? Suggestions? Comments?

Paul Kocialkowski

paul@bootlin.com