

```
---
title: 'The Linux Graphics Stack in a Nutshell'
author:
- Thomas Zimmermann <tzimmermann@suse.com>
abstract:
  This paper goes through the individual components of the Linux
  Graphics stack; describing their purpose, general way of
  working and how they interact with each other. It starts with
  video memory and kernel-side rendering, then looks at how an
  application draws content into its window and finally look at
  the compositor that puts everything onto the screen.
  The paper is for a beginner- to intermediate-level audience.
  You should have an understanding of the Linux system in general,
  but won't need expertise in graphics or graphics programming.
keywords:
- Linux
- kernel
- graphics
- driver
- DRM
- Wayland
- Mesa
...
```

Introduction

The Linux graphics stack is somewhat under-documented. While there exists documentation on the individual components involved in the stack and there is significant overlap among the individual communities, it is often hard to understand the bigger picture and how it all works together.

For most of the time, from the advent of Linux in the early 1990s to the late 2010s, graphics on Linux was implemented by the X Window System; X11 or X for short. The *X server* is a single monolithic component running in userspace where it provides input and output to its clients. *X clients* are applications that use the X server for interacting with the computer user. The user is in front of the X server's machine, while the X client applications are either running on the same host as the X server or on another host within the network. The former is common on single-user desktop systems, the latter is common on distributed systems or thin clients.

The latter use case is also representative of the environment in which X11 and it's predecessor were designed.¹ X was initially designed and implemented at the Massachusetts Institute of Technology during the mid-1980s. In the network-heavy and mainframe-centric computing environment of the university, having a

¹[TODO]

network-transparent is an almost natural conclusion. All interaction between X server and clients are packed into the X network protocol, distributed over the network (or at least localhost) and unpacked on the other end of the connections. This includes each X clients individual drawing commands that generate the application's output. Clients send drawing commands to the server, which then updates the screen accordingly.

This design of network-transparent drawing is slightly wasteful if client and server run on the same host machine. It is entirely incompatible with the requirements of fast, hardware-accelerated 3d graphics that became common during the 1990s. The amount of input data required for 3d graphics cannot be transferred over networks quickly enough to saturate hardware.

Linux solved this problem by introducing the *Direct Rendering Infrastructure (DRI)*, a set of components in the X client, X server and the Linux kernel. On systems with client and server on the same host, the client can draw directly with local hardware acceleration and handover the resulting buffer to the server for display.

Because of this and a number of other problems in X' design. X was slowly phased out on Linux and replaced by a streamlined graphics stack that fits the requirements of modern systems.

DRI's kernel component, *the Direct Rendering Manager (DRM)*, was extended to support memory management and modesetting. It now also enables accelerated 3d rendering. The X protocol has been replaced by Wayland, a smaller protocol that leaves drawing entirely to the client. The X server itself is still available for legacy applications, but it is implemented as an optional client within the Wayland-based graphics stack.

Video memory

The graphics stack's central resource is video memory, as all components of the graphics interact with it in some way. Memory management is therefore one of the primary tasks of Linux kernel's DRM framework and its drivers. DRM doesn't expose video memory directly. It provides buffer objects that represent a specific amount of memory.

Video memory comes in a variety of configurations and the exact implementation depends on the hardware. With discrete graphics adapters, video memory is installed on the device and connected to the PCI-like bus.² In System-on-a-Chip designs, video memory is usually part of the system memory. An in-between solution that is often used with integrated desktop GPUs, is stolen memory. The system BIOS sets aside a few Megabyte of system memory that acts like dedicated video memory, although it is not actually connected to the PCI bus.

²In this paper, the term *PCI-like bus* covers PCI itself and its successors, such as AGP or PCIe, in the various existing versions. While the electrical and logical details of these busses often differ widely, they are mostly equivalent to device drivers and userspace programs.

The stolen memory stores the displayed framebuffer, but is no longer available as system memory.

Every graphics drivers implements a memory manager according to the hardware's features and requirements. The kernel's DRM framework provides a number of helpers that implement common solutions

- Translation Table Manager (TTM): TTM drives discrete graphics cards on the PCI bus. It supports discrete video memory, GART and system memory. Each device's graphics driver announce the available memory areas to TTM and lets it handle the details. Graphics buffers can move among memory areas. Graphic cards usually require source and target buffers for scanout and rendering within the discrete video memory. But they can often fetch textures and object meshes from GART memory. If a buffer object is not required, TTM will move it to system memory.
- SHMEM: graphics buffers are located in system memory. Memory is fully cachable and swapable. Some chipsets can scanout directly from system memory (ARM Mail). Other chips use SHMEM-backed buffer objects as shadow buffers and perform `memcpy()` into their video memory as part of the pageflip operation. Uses kernel's `shmem` functions.
- DMA: graphics buffers are located in DMA-capable memory. The memory often has limited caching (write-combine). SoC often have areas of shared DMA memory where drivers can allocate from. SoC are usually capable of operating directly within the memory.

Those 3 helper libraries generally cover the needs of all supported hardware. Some drivers build additional functionality on top. Intel's i915 does it's own memory management for integrated GPUs, but has switched to TTM for the discrete cards.

GEM is the kernel's internal interface and a number of `ioctl()` calls to interact with buffer objects. The common data structure is `struct drm_gem_object`, which all memory managers inherit their own buffers from.

GEM interfaces allow to pin and unpin, `vmap` and `vunmap`, and `mmap` to userspace.

There's no GEM interface to actually create buffer objects. As each buffer object is subjects to features and requirements of the hardware, that's left to each driver. For each kernel driver there's a compatible userspace component that calls the correct `ioctl()`.

For very simple use cases, there's the dumb-buffer interface and `ioctl()`. It fills the gap left by the missing GEM create functionality. Userspace can call it to get a buffer object. It's unaccelerated and therefore only usable for software rendering. Useful for clients with minimal dependencies, such as Plymouth during system startup. Memory managers also implement the dumb-buffer

functionality. Drivers for hardware without accelerated rendering only provide dumb-buffer `ioctl()` for creating buffer objects.

Rendering

Rendering is highly dependent on hardware features. Most of DRM's supported hardware are simple framebuffers that don't support accelerated rendering at all. DRM client that wish to render on such a device can create a buffer object via the dumb-buffer `ioctl()` and implement all rendering in a software library.

Performance is a concern in this scenario. The system CPU has to perform all rendering operations. Although slow, this is usually manageable for today's user interfaces. Applications with sophisticated graphics are not usable, though.

Besides the rendering algorithm, the graphics hardware itself can be a bottleneck. Clients have to be careful to not read-back from the provided buffer object's memory. Although rendering is unaccelerated, the memory can still be located in discrete VRAM on the hardware. Such memory uses write-combine caching. Only small writes are cached. Every read operation will transfer data over the bus uncached. This can easily slow down rendering to single frames per second.

Accelerated rendering avoids these problems. However, there's no common interface or feature set among drivers. The client in userspace allocates the output buffers via driver-specific `ioctl`s. It sets the shader plus the required resources, such as textures or meshes, and invokes the shader. Shader and resources are stored as DRM GEM buffer objects.

Ideally, the DRM driver is a simple proxy between the client in userspace and the rendering hardware. To make this work, it must be possible to isolate the shader's address space from the rest of the system. In case of an error, the rendering hardware has to inform the driver about it. In this case, clients can provide the shader to the hardware. Only in case of an error, the hardware will involve the DRM driver.

One noteworthy exception is the `vc4` driver for Broadcom VC4 chips as found on the RaspberryPi. Being a SoC design, all components share the same physical memory. The GPU does not provide an I/O MMU that isolates the shader from the rest of the system. So a shader can potentially access any location in memory. To prevent this from happening, the `vc4` driver walks over the shader's instructions and verifies that they only refer to valid memory regions, such as bound textures.

Client rendering

With the kernel interfaces in place, we can now look at the client rendering. DRM clients generally run in userspace, with the only exception of the kernel's graphics console. We'll get to this in a later section.

Userspace clients interact with DRM via the device files in `/dev/dri`. Files named `render*` support only rendering, files named `card*` also support modesetting. Historically, `card*` files came first. Separate device files for rendering were introduced for privilege separation and for hardware without display output.

Clients do not render directly onto the screen, but into graphics buffers. As outlined before the GEM buffer object can be obtained via dumb-buffer ioctls or via driver-specific interfaces. The dumb-buffer object's memory can only be written via mmap-ing the memory pages and copying raw pixel data. Userspace libraries, such as `pixman`, can be used for 2d rendering. That's good enough for rendering 2d applications in software, but usually not sufficient for high-performance 3d graphics.

Buffer objects that have been allocated via driver-specific interfaces provide hardware-accelerated rendering. Such buffer objects also store graphics shaders, textures, object meshes and all kinds of graphics data.

The details of rendering are hidden from the client. The Mesa library implements a number of 3d rendering interfaces, most notably OpenGL, OpenGL ES and Vulkan, which clients employ. As these interfaces are standardized and hardware neutral, most clients don't have to care about the features of the hardware or DRM driver. If anything, a client tests if a certain version of a rendering interface is available.

Mesa is structured into the standardized front-end interfaces that clients employ and backend implementations. The backends are written for software rendering or the various DRM drivers. Both are connected by Gallium, a midlayer framework within Mesa. Gallium provides the individual pieces of a 3d rendering pipeline in a driver- and frontend-neutral way. The *state tracker*, the implementation of a particular programming interface, turns client rendering commands into Gallium3D commands for further processing. For shaders, NIR is an intermediate representation that sits between the frontend's shader language and the backend driver's hardware instructions.

TODO: paragraph about video

Independent from the interface or rendering software, the client usually contains a scenegraph with all the 2d or 3d graphics objects. After walking over the scenegraph and invoking rendering commands, the client will hold the results in the rendering buffer's GEM buffer object. If the buffer object backs a window of the user interface, it is now ready for compositing.

Compositing

Compositing is the process of taking a number input buffer objects and creating the final image that is being shown on the screen. In a typical user interface, this on-screen output is composed from a number of sources. In the background is a wallpaper image, background pattern or color. Drawn on top of that are the application windows of the active clients. The arrangement of those windows

depends on the compositor functionality and the preferences set by the user. Interacting with the application window usually means interacting with the application. In front of the application windows are the interface elements, such as the taskbar or application menu, where users can interact with the compositor itself. And finally the top-most layer is an indicator of what the user is currently interacting with; usually a mouse cursor on desktop systems.

This stack of windows and interface elements is nothing but another scene graph. The compositor is itself a DRM client that renders as described in the previous section. It is, however, a privileged one as it can modify the on-screen content and point it to its own output buffer.

Applications that take part in compositing are represented by one or more windows. Each window is backed by a buffer object that is being rendered to by the application. The rendering process has been outlined in the previous section. Once complete, the application transfers control of the rendered output's buffer object to the compositor. The application's output buffer now becomes an input buffer for the compositor. The easiest way to implement this in the compositor is to render a rectangle at the application window's on-screen location and use the provided buffer object as a texture image.

The compositor and its connected applications communicate by using the Wayland protocol. Wayland is a traditional client-server design, where the applications act as clients to the display service that the compositor provides. In the Wayland protocol, a surface represents an application window's on-screen area. Attached to it is a Wayland buffer that contains the displayable data plus color format and size information. Changing a surface's attached buffer eventually results in a Wayland protocol message from client to server that updates the on-screen content.

The Wayland protocol is transaction oriented in the sense that an application can make multiple changes to a surface before committing them to the compositor. On the compositor side, there's interest in moving to transactional processing as well. The compositor collects incoming changes to a surface and possibly its subsurfaces and applies them at once after the commit [gnome 44]. This avoids tearing or intermediate state during surface updates.

All protocol transfers happen on the compositor's host system via a local stream socket. [^Try `ls -l XDG_RUNTIME_DIR/WAYLAND_DISPLAY`]. The most significant advantage of local transfers is the ability to use shared memory for buffer transfers. With a network-transparent protocol, applications have to be encoded and possibly compress buffers into a network format before the transfer. Transferring full buffers over networks (or even localhost) has an inherent overhead. Afterwards the compositor has to decompress and decode the buffer into its native format.

With local stream sockets, the application creates a file descriptor that refers to shared memory and transfers it over the socket (at the size of an 'int') and let the compositor mmap the file descriptor's memory. Both now have a

established a channel where the application draws into the shared memory and the compositor renders from it. Wayland’s surface-damage messages can serve as a synchronization method of low overhead.

Shared memory (just) good enough for software rendering. For hardware rendering it is insufficient. The application would have to render in the graphics hardware, read back the result over the slow hardware bus and copy it into the shared memory area. To avoid this, compositors and applications can transfer graphics buffers via Linux’ dma-buf mechanism via Wayland. Applications render their window content via Mesa interfaces as outlined in a previous section. This ends with a swap-buffers command of the rendering interface, such as `eglSwapBuffers()` with OpenGL, which invokes the update to the Wayland surface’s buffer. Instead of transferring a reference to shared memory, applications send a reference to the GEM buffer object, which we have seen in previous chapters.

Note that the details of this are more complex than illustrated here. Accelerating rendering via graphics hardware is inherently asynchronous and therefore requires synchronization. After the application has sent the current frame’s final rendering command to Mesa and swapped back and front buffers, it is not guaranteed that the hardware has finished rendering. This is intentional and required for high performance. Having the compositor display the content of the GEM buffer object before the hardware has completed results in distorted output.

Instead, the hardware signals when it has completed rendering. This is called fencing and the associated data structure is called a *fence*. The fence is attached to the dma-buf information that the application transfers to the compositor. The compositor has to wait for the fence to signal completeness before it can use the resulting buffer for its own output.

As outlined in the previous section the applications’ output buffers become the compositor’s input buffers, from which it created the final on-screen image. The image is then displayed onto the screen, using DRM’s kernel interface for modesetting.

Scanout and Display Output

Graphics devices perform display output by reading pixel data from video memory, converting it into a display signal and sending it to a connected display device.

DRM provides a number of data structures and `ioctl()` interfaces to implement this. The pixel data is stored in GEM buffer objects that we’ve already seen for memory management and rendering. A DRM framebuffer gives structure to buffer objects by assigning a color format and a resolution. The number of object objects per framebuffer depends on the framebuffer’s color format. The most common case, 32-bit XRGB, and many other formats use only a single buffer object that stores all pixel components. For other formats, pixel components are stored in individual pixel planes, where each pixel planes is represented by it’s

own buffer object. That's common with YUV formats.

For display generation, the framebuffer needs to be placed on the screen. DRM planes represent the hardware components that fetch pixel data from the video memory and places it within the generated output. If multiple planes are active, they are layered on top of each other in a stack of planes. The stack's bottom plane is in the background and the stack's top is in the foreground.

DRM distinguishes between primary planes, cursor planes and overlay planes. Primary planes are at the bottom of the plane stack and provide the *background*, such as a graphical desktop environment. They usually cover the full screen and cannot be moved. Cursor planes are at the top of the planes stack. They are small and can be moved within the screen; just like the mouse cursor they are being used for. Overlay planes are anything else. Hardware often supports video output in a corresponding color format. They are located between primary plane and cursor.

The taxonomy is more for historical reasons. Display clients and programming interfaces used to distinguish between various types of planes, but DRM uses a *universal planes* interface in which all planes are alike. Every plane has the same properties that describe the possible sizes, location or color formats. Display clients still use the notion of primary plane, cursor and overlay to quickly test for features available.

Planes are closely related to the CRTC. The CRTC is the central data structure that represents the hardware components for compositing and display-mode generation. For each pixel on the screen, it takes each individual plane's framebuffer data, interprets it according to the plane and framebuffer configuration and puts it onto the screen. It then moves on to the next pixel. Each pixel might move through additional post-processing stages, such as brightness or colorspace adjustments, before going to the output.

For the output, DRM provides a number of structures, most importantly encoders and connectors. The encoder represents the hardware that creates the output signal and the connector represents the physical connector and the attached display. The connector also provides information about the display's characteristics; usually given as EDID data.

Together, these elements form the display pipeline, where pixel data flows from planes to CRTCs, encoders and connectors. The individual elements can often be connected in a variety of ways. For example, a CRTC might send to multiple encoders, or a framebuffer might be mirrored on multiple CRTCs. While this gives a lot of flexibility, it also create a problem. Not all combinations of pipeline elements might be supported. For example, with limited video memory, it might not be possible to store the framebuffers for the high-resolution primary plane and the cursor plane's pointer image at the same time. Switching modes (or even simple page flips) might fail.

DRM's atomic modesetting solves this problem. Atomic modesetting tracks the

complete state of all elements of the display pipeline in a single, compound data structure. DRM modesetting is atomic in the sense that it either applies the compound full state of all display pipeline's elements, or none of it. Clients that wish to program a display mode, set the state of all elements and apply them at once.

Modesetting involves two phases: an initial check of the new state, and a commit of the same. DRM state checking is independent from the hardware state. The DRM core, its helpers and the DRM driver test the state against the limitations and constraints of the available display hardware. For example, a plane has to verify that the attached framebuffer is of compatible size or the CRTC has to verify that the given display resolution is supported by the hardware.

If checking succeeds, the new atomic state can be programmed to hardware during the commit phase. If state checking fails for one or more of the elements, DRM stops the modesetting operations and returns an error to the client. It is also possible for the client to only verify atomic states without committing them. This allows DRM clients to build a list of supported configurations beforehand.

DRM provides a number of ioctls for modesetting. The ATOMIC ioctl loads an atomic state for checking and possibly committing into the kernel. This covers all elements of the display pipeline. From pre-atomic times, there are still ioctls that cover individual functionality, such as SETCRTC to program the display mode or PAGE_FLIP to change the primary plane's framebuffer. By themselves, they can only guarantee a successful update of their own state, but not a successful update of the entire state of the display pipeline. So immediately after receiving them, DRM maps these older ioctls to the atomic check and commit primitives to give them atomic semantics.

Conclusion