

The Mine of Information (Nuggets of Programming and Linux)



Linux Graphics Stack Overview

First published on: July 22, 2015

Categories: [Linux](#)

About

Welcome

Recent Posts

[Hyperconvergence](#)

[OSNews Various](#)

[Learning Python](#)

[Thoughts on the Equifax Data Breach](#)

[Data Warehousing](#)

[The Bitwarden Password Manager](#)

[Yubikey, FIDO2 and Backups](#)

[More databases - MemSQL and RocksDB](#)

Categories

[BigData](#)

[Cloud](#)

[Cryptography](#)

[Git](#)

[Infrastructure](#)

[Java](#)

[Links](#)

[Linux](#)

[Management](#)

[OSGi](#)

[Off-topic](#)

[OpenWRT](#)

[Programming](#)

Intro

This article gives a brief overview of how graphics are generated on a Linux system.

If you're trying to figure out how to configure your X server - you've found the wrong article. This is really only of interest to software developers (or maybe even interesting to nobody other than myself).

As with any attempt to summarize, there are areas below that are only *approximately* correct - this isn't meant to be textbook length. However if you see any fundamental errors, please let me know!

Towards the end of this article is a picture showing how the parts fit together; you might want to jump forward to look at that briefly before reading through the following text. However not everything on that drawing will make sense until *after* the descriptions below.

Note that I use the expression "graphics card" regularly below; however most of the time the information applies equally to embedded graphics systems (ie those on the motherboard or even on the same package as the CPU) as well as physical PCI/PCIe-based cards. You may find it useful to [read about the PCI bus](#) first.

Graphics Hardware Overview

Frame Buffers

In the end, all graphics cards generate a "frame buffer", ie a block of memory with 3 bytes per pixel, being the RGB color to show at that pixel on the screen. One frame-buffer known to the graphics card is the "scan-out buffer", ie the one that will be displayed.

For analogue outputs such as composite-video or component-video, card circuitry periodically reads each pixel of the scan-out buffer in turn and generates an appropriate voltage output at the appropriate time. For digital outputs such as HDMI, the graphics card simply dumps that data to the output cable (with some minor transformations), and lets the display device map that data to appropriate voltages to drive the physical output.

Very simple graphics cards do no more than support a "frame buffer" and associated output circuitry, and let the operating system write RGB values into that buffer. Libraries in the operating system then provide more sophisticated APIs for 2-D graphics or 3D graphics, and use software algorithms running on the CPU to compute the color of pixels to be written to the framebuffer.

More sophisticated cards provide various ways for an operating-system to offload the work of mapping graphics operations into pixel-colors from the CPU to the graphics card - but in the end the result is still a buffer full of pixel-values.

Of course modern cards provide multiple output options (eg component, HDMI, multiple monitors). Cards also usually support different *resolutions* (ie change the size of the scan-out buffer and the way its contents are mapped to pixels on the screen). Some cards support more than one "scan-out" buffer for a single display (the buffer contents are "layered" on top of each other). Selecting the output-device, resolution, and scan-out buffers is called "modesetting".

Graphics acceleration with modern cards

Modern graphics cards are basically a co-processor, ie perform "offloading" of CPU work. The general process for rendering graphics is:

- Allocate a buffer in which to write the output;
- Allocate one or more buffers and fill them with input parameters, eg tuples of (x,y,z,normal,color);
- Allocate one or more buffers and fill them with CU instructions for the Compute Units of the GPU to execute;
- Fill a "command stream" buffer with CS instructions for configuring the GPU;
- Pass the buffers to the GPU for processing.

Modern GPUs support two completely different instruction-sets: (a) the "command stream" used for configuring the card, and initiating rendering, and (b) the SIMD "compute stream" used for calculating pixel values. Early graphics cards with basic 2D support had only the "command stream" instruction-set; these instructions can configure the card output, map memory appropriately, and include explicit operations for drawing operations such as bitblits, drawing lines, etc. Later 3D cards added various fixed-purpose computation units that the command-stream could configure. Eventually GPUS added generic SIMD processing (primarily for "programmable shaders") and defined a corresponding instruction-set; over time this SIMD processing became so powerful that most of the original 2D

drawing operations and 3D “fixed function” units are now obsolete (using SIMD instructions is faster). The original “command stream” instructions are still used for “setup” tasks.

When execution is complete, the output buffer contains a 2D pixel array of RGBA values, suitable for copying/composing to an appropriate area of the framebuffer. Asynchronously to this process, the card’s output circuitry is repeatedly turning the “scan-out buffer” contents into appropriate signals to the display device. Actually, some cards have multiple sets of output circuitry, each of which can be using a different scan-out buffer - and potentially have a different resolution (“multi-head”).

Cards can have internal memory, or can share the CPU’s main memory. When the RAM is on card, then loading the program, vertex graph and textures from the CPU into the card is slow, but computing the results and writing to the frame buffer is fast. When RAM is “system memory”, then the reverse is true. Possibly more importantly, separate graphics RAM allows the CPU to perform memory accesses while graphics rendering is in progress, without contention for the memory bus.

Modern DRM-based kernel graphics drivers provide access to the GEM/TTM modules that allow user-space programs to allocate and access the buffers mentioned above: for “window backing buffers”, vertex graphs, textures, shader programs, and configuration commands. Apart from actually allocating/managing these buffers, preparation for rendering (mapping from graphics API calls such as OpenGL or GTK+ into the appropriate data and GPU instructions for the specific installed video card) is done via card-specific *userspace* libraries (eg Mesa-3D + gallium).

Graphics with X

The standard way of running multiple graphical applications on a single machine is by using the X (aka X11) display system. I’ll therefore talk about that primarily, with occasional comments on alternatives.

Display Managers, Startup and Login

The most direct way of using X is for a user to *log in* via a *console* (ie a physically-attached keyboard and screen), resulting in a text-only session. The user can then start an X server instance (resulting in a graphical display), and a “window manager” application which connects to the X server as a client. The user can then interact with the window manager to launch further graphical applications. Exactly which *client* applications are started when the X server instance is started is controlled by file “\$HOME/.xinitrc”; when that script terminates the X server is also terminated. Note that starting the X server is done via a SUID script, so that the X server instance runs as the root user.

Alternatively, the user first interacts with a [display manager](#) via a graphical *greeter* dialog. On a normal desktop system, a single display-manager process is started on system bootup. This then starts an X server instance attached to the system console, and displays a graphical “greeter” dialog-window. The code to display the greeter may be a separate application that the display-manager starts, or may be built in to the display-manager itself; in either case the process displaying the greeter window connects to the X server as a *client*. When the user enters credentials into the greeter, and the login succeeds, then the current X server instance is terminated and a new one is started for the user (though it still usually runs as root). On user logout, that X server is terminated and a new one is created for use by the greeter. The display manager manages the start/termination of the relevant processes. There are many different display manager implementations including [XDM](#), [GDM](#) (Gnome Display Manager), [SDDM](#) (the Simple Desktop Display Manager), etc. Although some of these display-managers are developed as part of a specific desktop environment project (eg Gnome), they are all capable of launching different desktop environments. There are a few greeter implementations which are not X clients but instead use *framebuffer* APIs to render graphics; in this case no X server is needed to display the greeter. Most modern greeter implementations also support the *Wayland display server* in addition to X.

In the last few years, effort has been put into supporting “rootless X”, ie running the X server as a non-root user. In this case, the X server running when nobody is logged-in uses a non-root “system” account. When a user logs in, that X server instance is terminated and a new one running as the logged-in-user is started. On logout, the current X server instance is again terminated and a new one started (to support the greeter program). An X server cannot run as non-root with “userspace DDX drivers”, and instead must rely on kernel-modesetting aka KMS (see later).

TODO: how is graphical “fast user switching” supported?

See Also:

- [Debian Wiki: Display Manager](#)
- [GDM Overview](#)

X server instances

An X server instance “takes control” of a local graphics card (and some local input devices, though that is not relevant here), initialises the card (using the settings in an X config file or specified on the commandline), then waits for other processes to connect to it and send it graphics commands over a socket (which may be a TCP socket or a local socket).

A more recent approach to graphics is Wayland. As with X, a server is started when a “desktop” user logs in, and client applications connect to it. However wayland requires clients to use local sockets only, ie client applications must be on the same server. Clients do not send “drawing commands” but instead pass references to memory-buffers containing pre-rendered graphics; the server then simply *composites* the images provided by all clients onto the scan-out buffer. Buffers (and other resources) are passed efficiently over the socket as *filehandles* or *buffer ids*, thus the requirement for local sockets only. Remote clients can be supported by running some “proxy” which is a normal client of the wayland

server but also accepts data over the network in some form; one option is for the proxy to accept standard X11-format graphics commands.

Note that an “X server” is a process running on a computer with a graphics card; it accepts commands over a network from multiple clients and writes to the graphics card. When a user has a weak desktop and a powerful remote computer available, then it is possible to run *cpu-intensive* applications on the remote computer while having the graphical interface for those applications appear on the desktop. The terminology gets a little tricky in this case; the remote computer is often referred to as a “server” - but the applications running on it are “clients” of the “X server” running on the local desktop. Under Wayland, the terminology would be the same: the “wayland server” is on the desktop, the “wayland client apps” run on the *remote (server) computer*.

An X server is a complicated thing; a recent count of source code lines showed the xorg X11 server to have significantly more lines of code than the Linux kernel.

X Network Protocols and Client Libraries

The core network protocol for X is basically a generic remote-procedure-call protocol.

An X server must support a “core” set of functions. It may then support any number of “extensions” that add more functions. An X client can query an X server to see which extensions it supports. Over the years, a large number of X extensions have been defined, and many of these are now *effectively* mandatory in that a large number of X client applications simply won’t work without them.

When writing an X client, there are two direct APIs for X: `xlib` and `xcb`. Both use the same wire protocol (ie network packets) to communicate with an X server. XCB is the modern, better, API; in particular `xlib` provides many functions that *block* until the server returns a result, while `xcb` is based around callbacks or queues of events. Of course many applications don’t use the X APIs directly, but instead use GTK+ or Qt or similar libraries which have “back ends” that use `xlib` or `xcb` to talk to an X server.

X Client/Server 2D APIs

X offers a set of functions (an API) for performing traditional 2D drawing operations (chars, rectangles, lines, etc). An X client application can call these functions directly (using `xlib` or `xcb`), or via a wrapper such as GTK+, but in either case network packets are sent from the X client to the X server to invoke the X 2D drawing functions.

Exactly how the X server handles such requests from clients is an implementation-detail. The xorg server implementation performs a mixture of internal operations and calls to the DDX api; see section on “X Userspace drivers” later.

Actually, much of the “core” set of X functions are no longer needed; they were designed in a quite different era and the drawing operations (line, rectangle, circle) and text-display (server-side fonts) they support were simply becoming obsolete. The XRender extension defines a large set of graphics operations that basically replace all the originals with better-designed alternatives - new text (glyph) operations, sophisticated shapes defined as lists of tessellated triangles and *composition* of such shapes together with support for alpha-channels etc. XRender doesn’t quite define a “scene graph” to be rendered but it comes fairly close. Data for XRender is still transferred using the normal X network protocols; the DDX API (see later) was presumably updated to efficiently accelerate rendering of such structures - 2D graphics cards are very good at exactly these sorts of things.

X Client/Server 3D APIs

The [Khronos group](#) publish several graphics-related specifications, including the OpenGL specification which defines an API for drawing 3-dimensional graphics.

An X client uses the EGL, GLX, or GLUT API to obtain an OpenGL context object. It then invokes functions on this context. EGL/GLX/GLUT are all “bootstrap” libraries for OpenGL which glue OpenGL to some specific system (eg X11+DRI or Wayland or Mir). GLX is x-specific while EGL and GLUT are cross-platform. GLX can optionally be configured to perform “indirect rendering” while EGL supports “direct rendering” only.

In “indirect rendering”, creating the OpenGL context sends a request to the X server to invoke a function from the GLX extension, which reserves/creates necessary resources. Further calls on the client to the OpenGL context are batched on the client side and sent to the X server when appropriate. On the server, this data is passed to a local OpenGL library which generates GPU instructions for the local graphics card and passes them on.

In “direct rendering”, creating the OpenGL context immediately sends a request to the X server to invoke a function from the DRI extension. The server returns the ID of a *local device node* (`/dev/dri/card*`) which the client then opens (or in DRI3, an already-open filehandle is returned). The client then uses `libdrm` to perform operations on this device-node in order to allocate graphics buffers and map them into the client’s memory-space. The client then uses OpenGL functions to generate appropriate sequences of GPU instructions into these buffers and submit them to the GPU directly. The device-driver underlying the opened node validates the submitted GPU instructions to ensure that only a “safe” subset of commands are issued, that buffer transfers do not overwrite resources for other clients, etc - ie the X client has *direct but limited* access to the actual graphics hardware. Finally, after the GPU has rendered the resources submitted by the client into a client-specified buffer, the client uses another DRI extension function to submit the buffer to the X server for *compositing* onto the scan-out buffer.

Obviously, DRI is only supported when the X client and X server are on the same host. The kernel-drivers behind the `/dev/dri/card*` nodes (one node per graphics card in the system) are card-specific, and are called DRM drivers;

userspace uses ioctls to perform buffer-management and (for modern DRM drivers) modesetting. There is more detailed information on DRI and DRM later in this article.

In “indirect” mode, the X server needs access to a card-specific OpenGL library in order to generate appropriate GPU instructions; in “direct” mode it is the X client which needs access to an appropriate OpenGL library. Further information on OpenGL is presented later.

Note that the library which creates the original OpenGL context object (mesa?) needs to know about GLX and DRI, in order to create a context object that behaves appropriately.

As noted earlier, an X server (or Wayland server) also controls *input devices* such as keyboards and mice, forwarding events to the relevant application. However this article is about graphics not input-devices so this will be ignored in the rest of the article.

libdrm is maintained by the xorg project, ie the same project that implements the standard X server.

X Userspace drivers

The Xorg implementation of an X11 server splits the code into a “front end” (Device Independent X aka DIX) and “back end” (Device Dependent X aka DDX). DDX back-ends include:

- xwin - backend for running xorg on Windows
- xquartz - backend for running xorg on OS/X
- xnest - backend for running xorg as a client of another xorg instance
- xwayland (since v1.16) - backend for running xorg on top of Wayland
- xfree86 - the “normal” backend for linux (though xwayland may become more popular in the near future)

Note that the name “xfree86” has been retained even after maintenance of the xfree86 sourcecode was transferred (back) to *xorg*.

The xfree86 backend has a “plugin” architecture; it loads (at runtime) a suitable “userspace driver” for the graphics card it is using. There is of course therefore an *API* that such drivers must implement; actually this api comes in two parts: a “base” api and an “acceleration” api. The card-specific xfree86 drivers are usually named `xf86-video-*` or `xserver-xorg-video-*`. Each driver declares a structure of type `XF86ModuleData` which contains a pointer to a “setup function”; X invokes that setup function and the function then calls `xf86AddDriver`, passing a structure of type `DriverRec` which has pointers to several callback functions including a “probe” function. That eventually gets called, and returns a `ScrnInfoPtr` struct which (yet again) holds pointers to various functions, including a `ScreenInit` function. When that gets called, it invokes `xf86LoadSubModule` to *load the relevant generic acceleration module* (eg loads module “`exa`”), then registers a bunch of callback methods with that module.

There have been several “acceleration APIs” over the years: XAA (obsolete), [EXA](#), UXA - and Intel are currently working on a new one named SNA.

Xorg maintains a collection of xfree86-compatible drivers for various graphics cards. Traditional drivers are 100% *userspace code* which communicate with graphics cards directly, without using any kernel functionality other than `mmap` calls on `/dev/mem` to map arbitrary PCI address ranges (corresponding to the graphics card) into the userspace X server’s address-space [1]. These drivers are responsible for generating the appropriate commands to the graphics card to do any 2D acceleration, ie how fast X 2D graphics goes depends very much on how clever the xfree86 driver is in generating commands for the GPU. These userspace drivers are quite portable across different unix-like operating systems; once the graphics card control registers have been mapped into the X server’s memory space then all further operations are card-specific but not operating-system-specific.

[1] Question: I thought that the PCI bus itself merely determines what BARs exist, and leaves it to a matching driver to select which BARs to actually map. But if there is no matching kernel driver, then what maps the BARs so that they appear in `/dev/mem`?

The disadvantage of having purely user-space drivers (ie drivers where 100% of the functionality is in userspace) is that other processes, and even the kernel itself, no longer has any knowledge of the current state of the graphics card. Switching control of the card between processes is therefore very tricky to implement; examples of such handovers include:

- On login, shutting down the X server instance used by the “greeter” and starting a user-specific instance
- Switching user session without logging the original user off (“switch user”)
- Switching from an X session to a virtual console and back

The result of the above operations is therefore often “blinks” as the card is reset to a clean state, or sometimes graphics corruption.

Suspending/resuming a laptop is also tricky as the kernel cannot restore the appropriate graphics mode itself but must instead rely on a user-space app to do that.

Most of the xfree86 drivers have therefore been updated to support “mixed” operation, where at least the modesetting is delegated to a card-specific linux kernel driver; this is referred to as KMS (kernel modesetting).

There is a fairly new xfree86 driver named `xf86-video-modesetting` which works with any card for which there is an in-kernel card-specific DRM driver that supports KMS. However this driver does not support any kind of 2D acceleration

(it cannot as the driver is not card-specific so cannot perform any card-specific optimisations).

There is a fairly new xorg project named [GLAMOR](#) which implements 2D graphics operations by mapping them to OpenGL drawing operations. The initial implementation of GLAMOR was as an “acceleration API” that xfree86 drivers could optionally use (though the Intel driver is the only current user of it). It is expected that in the near future a complete DDX driver will be created that internally combines the xfree86-video-modesetting driver with GLAMOR, resulting in a driver that can configure *and* accelerate *all types of cards* as long as they have a suitable DRM driver and OpenGL implementation.

A GLAMOR-based driver will be particularly useful for the Wayland project, making it possible for them to provide XWayland (a “proxy” X server) without major effort. Existing xfree drivers write to the underlying card directly, bypassing Wayland’s compositing process - ie no existing xfree86 driver could be used unmodified with Wayland, and modifying *every* existing driver is an unreasonable amount of effort. However using the GLAMOR approach, it is possible to develop a single GLAMOR-based driver which is compatible with Wayland and handles *all* cards. This is what the “xwayland” DDX driver does: X client apps think they are talking to a normal X server; Xwayland uses OpenGL/DRI to draw window contents to a buffer then uses the Wayland protocol to hand the buffer over to a compositor for actually writing to the scanout buffer.

Kernel Based Graphics Drivers

Over the years, three different types of kernel driver “helpers” for graphics have evolved: framebuffer, DRMv1 and DRMv2.

First were the “framebuffer” kernel drivers. These provide a fairly simple API that is capable of setting the graphics mode on the “main” output for a graphics chip, allowing applications to explicitly set individual pixels in the output framebuffer, and do a few simple accelerated operations (bitblits and rectangular fills). Framebuffer drivers expose a file `/dev/fb{n}` which userspace performs reads/writes and ioctls on; all card-specific drivers provide exactly the same API so userspace apps can use any framebuffer driver. X has a simple userspace DDX driver which can handle all cards with framebuffer drivers, but with very little in the way of acceleration - the 2D drawing API (ie computing individual pixels) is implemented in software. See [this article](#) for more details.

Next were the DRM (Direct Rendering Manager) drivers; these were created to *support* the “DRI” (Direct Rendering Interface) extension for X, ie in order to perform “direct” 3D rendering. The kernel provides a “drm core” kernel module that contains common code, and card-specific drivers register themselves with the drm core to add card-specific logic (see the `drm_*_init` functions). The drivers pass a set of flags to indicate which capabilities they support (eg `DRIVER_MODESET` indicates kernel-modesetting aka KMS support).

The first DRM drivers provided just basic “DMA-style” buffer-management. Later the DRM core was enhanced to support many more (optional) capabilities, and the card-specific drivers were updated to implement the optional functionality, effectively producing a “DRM version 2” API. These new drivers support:

- the complete functionality of the old framebuffer drivers (ie the “old” modesetting API, pixel-level read/write, bitblits and rectangular fills)
- the complete functionality of the old DRM drivers
- a new graphics modesetting API (also known as Kernel Mode Setting aka KMS) which is an improvement over the framebuffer-style modesetting
- a new graphics buffer management API (GEM or TTM)

The DRM drivers primarily exist to allow userspace code (whether X client or X server) to allocate memory buffers, fill them with data or GPU instructions, and pass them to the graphics card. The “DRM v2” drivers (note: my invented name, not an official term) also support mode-setting (aka KMS). When an “old DRMv1 style” driver is registered with the DRM core, the core creates a device node named `/dev/drm*`; when the flags passed to the core indicate more modern capabilities then the DRM core create nodes named `/dev/dri/card*`, `/dev/dri/control*`, `/dev/dri/render*` and `/dev/fb*`. The DRM core handles `ioctl` syscalls on these device-nodes, returning errors for operations that the underlying driver does not support.

Because newer DRM drivers support both framebuffer modesetting and KMS modesetting APIs, graphics card state can be consistently tracked regardless of which API userspace applications use. TODO: what if old-style userspace DDX drivers is also concurrently in use? This allows smooth switching between framebuffer applications (eg ‘fbcon’ virtual consoles) and X applications; all information related to the state of a card is known to the corresponding driver rather than being embedded in the userspace DDX driver. The existence of DRMv2 drivers which support modesetting also opens up the possibility of performing graphics *without* using X at all, ie much of the tricky logic related to mode-configuration which was formerly embedded in DDX drivers has now been migrated to the DRM drivers. This allows things like Weston to be implemented without massive duplication of code from xorg. It also makes it possible to reliably switch control of graphics between various applications, eg between different X-servers (user-switch), or server and virtual console.

While DRMv1 and DRMv2 both provide a kind of “buffer management” API, the newer (GEM/TTM) APIs are far superior to the older “DMA management” style api provided by DRMv1 drivers - although AFAIK both are supported.

For each graphics card, there is one `/dev/dri/card*` node, one `/dev/dri/render*` node, and one or more `/dev/dri/control` nodes:

- The ‘card’ node supports ioctls for the full set of driver functionality.
- The ‘render’ node provides just the buffer-management functions. Separating this out makes it possible to grant applications access to a GPU without allowing them to change output configuration. This can be useful for

applications that just wish to perform off-screen rendering, or which want to use the GPU for number-crunching (GPGPU) rather than graphics. In particular, such an application can open the render node directly rather than via a DRI call to the display server.

- There is one ‘control’ node for each independent output (“head”) that a graphics card supports - ie the max number of concurrent scan-out buffers. Simple cards have only one set of display-output circuits and so can only generate one output stream at a time; “multihead” cards can generate multiple independent outputs from multiple scan-out buffers. Having separate control nodes makes it possible to grant an application modesetting rights on just *one* of the outputs (“heads”). This is particularly useful in a “multi-seat” environment where one server supports multiple concurrent users each with their own display; multiple X instances can each “own” a different control-file and perform modesetting independently.

The exact ioctl operations supported by the DRM core is not considered a stable API; drm functionality is accessed instead through userspace library `libdrm` which does have a stable API.

For 2D operations, X has a separate userspace `xfree86` driver for each DRMv2 driver, but they are somewhat simpler than the “pure userspace” `xfree86` drivers as they no longer need code to poke card-specific registers to set up graphics modes - the logic for these is now in the kernel driver. In all cases except the “framebuffer”, userspace drivers are responsible for generating the right instructions to a GPU for accelerating 2D graphics operations (ie OpenGL or similar is not used - though see information on GLAMOR later). In older cards, this was a reasonably simple job - the cards provided operations that mapped fairly directly to 2D operations. Modern cards, however, often require drivers to generate GPU “programs” to do even 2D graphics - a far from trivial process.

Although a DRMv2 driver implements the framebuffer API, there may also be a separate (simpler) framebuffer driver available for the same card. This is true for some intel and nvidia cards for example, while AMD have just a single combined driver. Duplicated code is bad, but a simple framebuffer driver is good - the tradeoff is difficult to judge.

With the invention of KMS, it would have been possible to rewrite DDX drivers to use the DRM apis to pass commands to the graphics hardware rather than map registers into memory directly; this would then have made it possible to run X as a non-root process. However this is a big job, and there were also some other corners of X that needed tidying up to allow non-root use. This was eventually achieved for at least some graphics drivers, but has never made it into a production linux distribution AFAIK - and may well be moot with Wayland looking to replace X as the standard display server. Converting xorg DDX drivers to use DRM is also controversial as the xorg server runs on non-linux systems too, and many do not have DRM/KMS.

Atomic Modesetting

At the current time (mid 2015), work is underway on something called “atomic modesetting”. This is yet another rework of the APIs used to tell graphics cards to change resolution, output device, and other such settings (KMS). Often it is required to change several display-related settings at once - but the original framebuffer API and even the improved KMS-style API only allow settings to be changed one-at-a-time, ie one change per system call. This can lead to undesirable flickering and other visible artifacts while the changes are in progress. Worse, there are some changes that the card can refuse - which then means the caller needs to “undo” the previous steps in order to restore the display to its original state. Failing to “undo” such steps can leave graphics output in an unintended “intermediate” mode.

The solution is to provide an API that is effectively a *list* of state-changes to apply, and then let the card-specific driver apply them “atomically”, ie all-or-none, and in the most appropriate order for that particular card. Of course this requires changes to the drm-core, `libdrm`, and the card-specific drivers so is taking some time to complete.

See [this LWN article](#) for related information.

Update: see also [this presentation](#).

More on DRI and DRM

The [DRI](#) functionality has gone through several iterations. However the basic concept has remained the same: the client fills memory buffers with graphics data such as textures, and other buffers with card-specific GPU instructions, and submits the buffers for execution. A DRM driver validates the buffers and forwards them to the graphics hardware. The results are rendered into another buffer which is then submitted to the X server for composition.

Traditionally, memory for graphics cards and memory for the main system CPUs have been separate, ie graphics cards have had separate “on board” memory. This memory was quite often *faster* than normal memory, with wider buses from it to the GPU. Data was transferred between CPU memory and GPU memory with a kind of DMA system - a significant performance hit. Graphics rendering also often requires lots of memory, ie only data *currently needed* by the GPU should be kept in GPU memory. The result is that allocating of “buffers” to share data between CPU and GPU needs to be handled very carefully to get good performance. The GEM/TTM kernel modules exist to perform exactly this task: efficient allocation of buffers. While they are kernel modules, they are not intended for direct access from userspace; they are called from the DRM drivers instead. Some GPUs share their memory and memory-bus with the CPU, in which case memory-buffer-management is somewhat easier.

The original DRMv1 drivers were just responsible for buffer management. New DRMv2 drivers also support KMS, ie the same device node can be used for mode-setting as well.

DRI1 is now completely obsolete.

In [DRI2](#) buffer allocation is mostly done by the X server, ie there is a function in the X DRI extension which allocates a suitable buffer and returns a *global integer id* for that buffer. The X client could then use this ID as a source or

destination in various DRI/libdrm functions, or could map the buffer into its local memory-space via the ID. Because these IDs are *global*, other applications on the same machine could potentially interfere with them. The DRI2 “create context” method returns the filename of the `/dev` node which the X client must use with libdrm; when this file is opened there is a tricky “authentication” dance to allow the driver to figure out what rights this X client should have.

In DRI3 (since late 2013) buffers are allocated by the client via libdrm, which returns a *file descriptor* rather than an integer id - much more secure. DRI3 functions take file-descriptors as parameters to identify buffers (passing file-descriptors over a local socket between processes is permitted). The DRI3 “create context” method opens the `/dev` node which the client must use with libdrm, and returns the *file descriptor*; this is much better than the DRI2 “filename” approach as it allows the X server to directly configure the driver with any security-restrictions etc. before the client gets access to the file-descriptor.

Presumably, closing the `/dev/dri/*` node will release all associated graphics resources - ie no need for a “close” function in DRI. And nicely, this means everything gets cleaned up on client crash too.

libdrm provides about 60 functions for X client use, including:

- `drmOpen`
- `drmCommandWrite`
- `drmMap` - map graphics card memory directly into caller’s memory space (subject to limits set by display-server via `drmAddMap`)
- `drmDMA` - reserve DMA buffers from the pool allocated by the display server (see `drmAddBufs`), or request transfer of the contents of a reserved buffer between client memory and videocard-memory (subject to checks applied by DRM kernel module)

libdrm provides about 60 functions specifically for display server use, including:

- `drmSetMaster/drmDropMaster` : gain exclusive access over a DRM node (done by the display server process)
- `drmCreateContext` - configure a “DMA queue” and return a handle for use by clients in the `drmDMA` operation
- `drmCreateDrawable`
- `drmAddBufs` - Allocate blocks of system memory in an address-range suitable for performing DMA to/from the GPU.
- `drmAddMap` - defines which graphics-card address-ranges X clients are permitted to map directly into their memory-space. In effect, this allows non-root applications to mmap parts of `/dev/mem` which the X server has determined are appropriate for the client to map.

See [this documentation on drmv1](#) for some documentation on the (original) DRM functions.

There appears to be a different libdrm library implementation for each specific graphics card - ie the userspace/kernel implementation differs depending upon which card-specific DRM kernel module is loaded. TODO: how does the X client know which libdrm to load?

See definitions in files `xf86drm.h`, `xf86drmMode.h` and `include/drm/libdrm.h` from repo `cgит.freedesktop.org/mesa` for the full API. Note that “xf86” refers to the old name of the xorg server (‘xfree86’) rather than any chipset.

As noted, some buffers will contain sequences of instructions for a GPU to execute. While theoretically various mechanisms could be used to generate such instructions, the X client typically dynamically loads an appropriate card-specific OpenGL library and passes the buffers to the OpenGL functions as the “target”. That is, DRI can theoretically be used to accelerate all sorts of graphics but in practice it is used for OpenGL.

Some [DRI2 functions](#) (from about 10 total):

- `DRI2Connect`
- `DRI2Authenticate` : links the buffers associated with the calling X client to a token, so the client can then pass that token to libdrm and get access to the same buffers
- `DRI2CopyRegion` : schedule a copy from one buffer to another
- `DRI2SwapBuffers` : replace the “active” buffer (ie displayed window) with the specified “offscreen” one

The *full* set of DRI3 functions:

- `Open`
- `PixmapFromBuffer`
- `BufferFromPixmap`
- `FenceFromFD` (a kind of synchronization primitive)

DRI3 also relies on the “PresentPixmap” function of the new Present extension, which tells the X server to “make this buffer the contents of this window”, ie effectively to composite the contents of the buffer into the scan-out buffer at that window’s current location.

There is a [DRM Developers Guide](#) which is an excellent source of more details on the DRM module, GEM and TTM.

GEM and TTM

One of the big bottlenecks in 3D is passing data between userspace applications and the graphics card.

Discrete graphics cards have onboard memory, and physical-address ranges on the system-bus are mapped to address-ranges on the graphics cards by writing to the PCI BAR registers. When the CPU writes to a physical address in this range, the PCI controller detects the write and forwards the data on to the graphics card. Similarly, when the CPU reads a physical address within this range, the PCI controller passes the read-command on to the graphics card which returns data that the PCI controller then passes on to the system bus, ie back to the CPU. Data can therefore be transferred either via direct reads/writes from the CPU (as just described), or by setting up DMA transfers between “normal” system memory and graphics-card memory address ranges. Because graphics cards can have large amounts of memory, it is sometimes necessary to map only *parts* of the graphics-card memory to physical system bus addresses; deciding which addresses to map is important for high performance, and non-trivial. Mapping graphics-ram to system-physical-addresses is particularly difficult on 32-bit systems; a 32-bit physical address can only address 4GB of data at a time.

Integrated GPUs use memory that sits on the standard system bus, ie memory that can also be addressed by the CPU; this is called Unified Memory Access (UMA). Of course code running on a CPU should be very careful accessing physical memory that is also being used by the graphics card - all sorts of race conditions could occur. *AFAIK*, there are two possible solutions:

- allow the GPU to perform DMA copies between the pages “owned” by the GPU and the pages “owned” by the CPU;
- *map* pages in and out of the GPU/CPU MMUs, ie “transfer ownership” of pages.

Question: do integrated GPUs always access memory via an MMU? How can the operating system ensure that code running on the GPU does not access other physical memory, ie allow applications with access to the GPU to read/write all physical memory?

Question: can discrete cards also reference system memory directly? A PCI device can become “bus master” and read/write system memory directly..

Question: can graphics chips on the motherboard have dedicated memory?

The userspace DDX drivers previously managed this CPU/GPU data-transfer (buffer management) themselves, but presumably not very effectively as they don’t have complete control over the CPU’s MMU settings. They do have complete control over the graphics card’s DMA functionality so DMA-based data transfer works fine (ie where all memory is on the card) but UMA-style systems (where the GPU can page in system memory) was presumably rather ugly.

The TTM kernel module was eventually created to provide a common kernel-private API for managing the transfer of data between system and GPU. Because TTM covered both the mem-on-card and UMA usecases, the API was quite complicated. The GEM kernel module was invented shortly after, with a much simpler API and a much simpler implementation - but it only supported UMA (it was developed by Intel who only produce integrated graphics). The TTM code was later reworked to keep the same functionality but provide the same simple API as GEM (“gemified ttm”). Both TTM and GEM are now in the kernel, and which is used depends on the underlying card: GEM works faster for UMA-only cards, but TTM is needed for NUMA cards.

The interface to GEM/TTM is hidden behind the /dev/dri/card0 ioctls which are hidden by libdrm - ie an X client manipulates buffers by invoking libdrm functions which perform ioctls which are handled by the DRM driver which forwards to GEM or TTM.

TODO: Strangely, it appears that libdrm provides very card-specific APIs, eg function `drm_intel_bo_gem_create_from_name(...)`. So the user of libdrm must have card-specific code in it too - well, at least card-family-specific. The DRM driver defines different IOCTL commands for GEM and TTM operations; a DRM driver which uses GEM will return an error if TTM ioctls are invoked, and vice-versa. Somewhat ugly, saved only by the fact that there are very few users of libdrm which perform buffer management - in fact, really only OpenGL implementations: mesa, intel-opengl, and nvidia-opengl.

Question: do userspace DDX drivers interact with GEM/TTM at all? If not, how do they perform buffer management? Presumably they (a) do what they used to do, and (b) don’t actually need to do much complex buffer management because they only support 2D graphics which has far lower data volumes than 3D.

- [Keith Packard's announcement of GEM with lots of API info](#)
- [kernel.org info on GEM and TTM](#)
- [DRM Developers Guide](#)

3D Drivers (OpenGL etc)

3D libraries (eg implementations of OpenGL) running in userspace access the graphics card by asking X or libdrm to set up an appropriate output buffer, then filling other buffers with vertex data, textures, shader programs, etc., and asking (via libdrm) the appropriate DRM kernel driver to pass those data buffers on to the GPU for rendering into the output buffer.

Note that there is very little kernel-specific code in an OpenGL library; such libraries are mostly portable across posix-compatible systems. The OpenGL implementation for a card is therefore more a *library* than a *driver*.

The OpenGL implementation for many graphics cards is part of the [Mesa](#) project, and often share significant portions of code. The Gallium framework is also developed as part of the Mesa project, and the OpenGL support for many cards (both within and outside the mesa project) is based on Gallium which thus also allows code to be reused. One important exception to this are the OpenGL drivers created by Intel for their own GPUs; this code is open-source but is currently

not hosted on the Mesa site nor uses Gallium. However there is a driver for Intel GPUs developed by an independent team which is in Mesa and uses Gallium (though features and performance lag the intel-developed version). There is a driver for nvidia cards in Mesa, but this lags well behind the proprietary drivers from Nvidia themselves. Until 2015, AMD had separate open-source (mesa/gallium) drivers and closed-source “catalyst” drivers; for their latest GPUs they now have a single mesa/gallium open-source driver and a proprietary user-space “helper” which accelerates performance for some uses.

The first accelerated 3D rendering library was Utah-GLX, and is now referred to as “GLX indirect rendering”. OpenGL API calls on the client are simply sent to the X server (via the GLX extension), which passes them to a suitable GLX-enabled driver. A GLX driver is userspace code that uses a card-specific OpenGL library to generate GPU instructions and card-specific code to pass these instructions directly to the graphics card. This suffers from the drawback that there is no “shortcut” for local x clients to access the 3D hardware; all operations must be encapsulated as X calls to the local X server which is simply a *huge* impact for something as data-intensive as 3D graphics. It would be preferable for clients to directly access the graphics card themselves, but this simply cannot be done from userspace; X can do it as (a) it is a “trusted app”, (b) it can run as “root”, and (c) only one instance runs at a time.

The DRI approach effectively pushes parts of the graphics rendering down into a DRM kernel module so that (a) the module can perform security checks on the data passed by the client, and (b) the kernel module can arbitrate between multiple processes. The rest of the functionality for performing 3D graphics can remain in userspace (OpenGL, libdrm, and DRI). This approach lets client applications safely generate graphics directly, solving the performance problems of “GLX indirect rendering”.

For remote clients, this “shortcut” is of course not possible, and they must use the GLX indirect rendering approach as originally designed.

Wayland

Recently, Wayland has been developed as an alternative to X. In some ways, Wayland can be thought of as a factoring-out of the lowest levels of X, dealing just with mapping of multiple windows into a common framebuffer and little else. Alternatively, Wayland can be considered an X server that exposes DRI to client apps. Wayland does not provide a 2D drawing protocol like X; instead it just provides a simple API for clients to pass full windows (buffers of pixels). Wayland still needs to set the graphics mode (but that is easy with KMS support), and to do lots of bit-blitting.

With Wayland, therefore, client applications use DRMv2 directly to allocate output buffers (rather than asking X), and use DRMv2 to pass data (vertexes, textures, shader programs, etc) to the GPU. The GPU then renders directly into the client app’s buffers and the client eventually “hands over” the completed buffer to the Wayland server for *compositing* (merging) into the scan-out buffer.

Cairo

X defines a very primitive 2D drawing API. The XRender extension defines an X network protocol extension for transferring far more sophisticated drawing primitives between an X client and an X server - but doesn’t provide a *client API* for generating such data. [Cairo](#) provides a modern API for 2D drawing which maps well to the XRender extension - ie when an application uses the Cairo API to draw, the results can be efficiently represented as an XRender package for sending to the X server. The [Cairo](#) API also (not coincidentally) maps well to the MS-Windows drawing API, and to other windowing systems too.

Potentially, the DRI system could also be used to render Cairo graphics - ie for a Cairo “back-end” to use libdrm to allocate buffers, write GPU-specific commands into them, and then use the X DRI commands to “hand over” the results to the X server for rendering. This would avoid significant amounts of client-to-server network traffic, but would require the client and server to be on the same host. The “opengl backend” for Cairo is currently marked *experimental*.

OpenGL, Mesa, Gallium

The above discussions omit the hardest part in all this - generating the vertexes and textures that describe a 3D scene, and generating the GPU-specific commands to implement vertex and pixel shading. The OpenGL specification defines a suitable API. Some graphics vendors provide their own opengl implementation that knows how to map the calls into tables and instruction-streams that the card understands. Others are supported by the open-source Mesa project which has common code for handling the vertices stuff, and card-specific back-end drivers for generating the card-specific instructions.

There are also 2D drawing APIs (eg Cairo) which want to take advantage of graphics acceleration. Cairo has one back-end that generates OpenGL calls; there are also experimental card-specific back-ends that generate GPU-specific commands directly.

The [Mesa](#) project is a collection of graphics-related libraries. It started as a *software rendered* implementation of OpenGL, but now also hosts various card-specific opengl implementations, and various wrappers.

Q: according to one source, when using Mesa, OpenGL features that are not yet implemented in a card-specific driver are instead *rendered in software*. Is this true?

Q: The [mesamatrix](#) site shows the current state of Mesa’s opengl drivers. AFAIK, the first “mesa” column indicates the state of the “opengl Gallium state tracker”, ie the “gallium front-end” that provides an OpenGL API to applications. The other columns then indicate the state of the card-specific “gallium back-ends” - except for the i965 column, as the i965 driver is not gallium-based. Hmm - maybe not; some features (for example,

OpenGL4.5:GL_ARB_direct_state_access) were implemented *simultaneously* for all hardware types. This suggests that MESA actually is a “front end” that the various card-specific impls plug in to, ie that there is an internal Mesa API that all the “back ends” implement - whether gallium-based or not.

As [described by Iago Toral](#), “Mesa is best seen as a framework for OpenGL implementators that provides abstractions and code that can be shared by multiple drivers.”

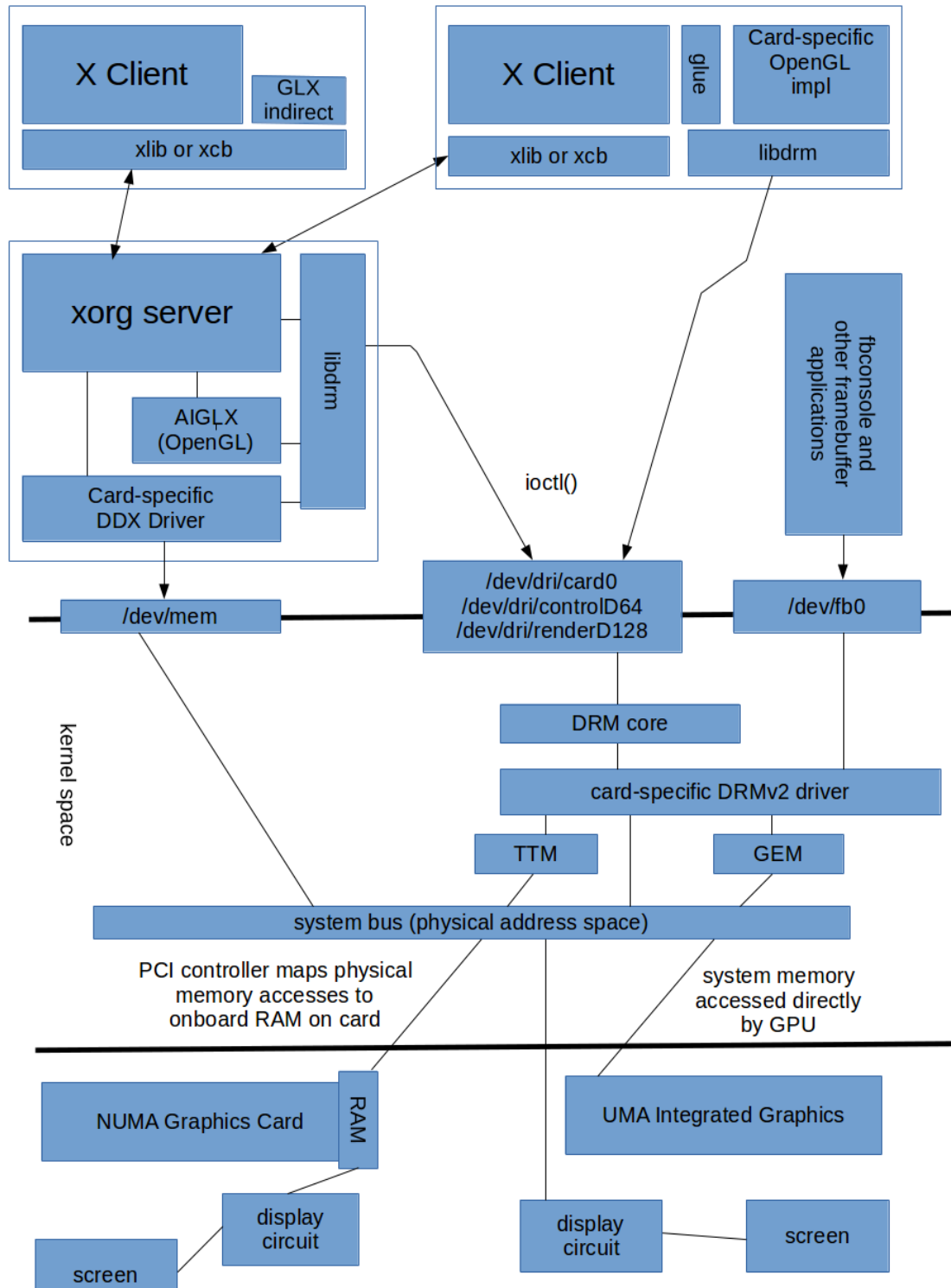
Gallium is an open-source framework that aims to split the task into two parts: a front-end “state tracker” provides a user-level api and generates “intermediate code”, and card-specific back-ends then generate the card-specific output. There are “state trackers” for OpenGL, Cairo, DirectX11(!) and potentially other APIs which can all reuse the same card-specific back-end code. Of course a “state tracker” for a complex API typically reuses significant amounts of code from the existing project (eg mesa, cairo). The above article by Iago Toral also has a useful overview of Gallium.

EGL is an API for managing windows and buffers. An OpenGL implementation is primarily responsible for generating appropriate blocks of data for a card to process, ie textures, tables of vertices, and streams of instructions - and all of this is *card-specific* but *not operating system specific*. However the data needs to be *put* into some buffer; OpenGL just requires the necessary buffers to be *passed in* as parameters, in order to retain its OS-independence. These buffers also need to be *pushed* to the card at some time. The responsibility of managing “windows” and “buffers” and “submitting” data has always been done external to OpenGL; the GLX API traditionally performed that glue for X on posix systems. However having a graphical application directly invoke a windowing-and-os-specific library is undesirable; EGL was therefore invented as a standard API which can have a different implementation for each OS/windowing system. An app coded against the EGL and OpenGL interfaces therefore just needs to be linked to the right library at compile or runtime. Note that EGL does *not* provide any “drawing” operations like OpenGL (or OpenVG) does; it simply handles the buffer management. There is an EGL implementation that uses the MS-Windows GDI interface to output the generated graphics, and one that supports DRI on linux (ie uses DRI X extension functions together with libdrm to talk to the DRM kernel module). Because EGL is a “helper for OpenGL”, and GLX (on the client side) is a *proxy* for OpenGL (ie doesn’t actually generate any GPU instructions), EGL is not a wrapper for GLX - so an app that wants to be able to render 3D on a remote graphics card must support GLX explicitly.

The standard implementation of the GLX API is [Mesa libGL](#).

The Big Picture

Now that the general topics have been presented, here is a diagram that shows the dependencies in graphical form..



Below the main points from this diagram are discussed; this will often repeat the information from the first part of this article, but in context.

X Client

An X client opens a socket to whatever location is specified in environment variable `$DISPLAY`. The X client app then uses either the xlib API or the xcb API to communicate with the X server; window-management and drawing commands are sent to the remote server, and events (keyboard, mouse, etc) are sent by the server to the client. The xlib API is the original api, and has many functions that “block” the calling thread; xcb is a newer API which is more consistent and based around event-queues rather than blocking. However both libraries generate identical output to the X server.

The X11 network protocol is very generic; basically a kind of “RPC” mechanism. Each packet specifies a “function” and parameters; the client can query the server to see which functions it supports.

2D drawing may be done via the original X functions (draw-line, fill-rectangle) or via the more modern functions belonging to the “XRender extension”. In theory an X server might not support the XRender extension - but in practice

most clients simply assume it is available on the server.

In practice, most applications do not use either `xlib` or `xcb` directly, but instead use a higher-level API such as Qt or GTK+.

Performing 3D graphics is more complex : the “direct” and “indirect” approaches are significantly different.

Direct OpenGL with X

In DRI (“direct 3D”), the client application uses some “glue api” to load a suitable OpenGL library implementation and initialise it. OpenGL defines an API for drawing to “surfaces”, but deliberately does not concern itself with concepts such as “windows”; it is therefore necessary to somehow integrate OpenGL with whatever windowing system is being used on this host (whether X, news, ms-windows GDI, Mac, etc). In particular, an *opengl context* structure needs to be created for use with the OpenGL APIs.

The traditional API used on linux systems to bind OpenGL to X is [GLX](#). The WGL library does something similar for MS-Windows, and CGL for Mac. GLX/WGL/CGL all have different APIs, ie an application written for one window system is not useable on another.

The Khronos group which specifies the OpenGL API has created its own API named EGL which can be used to bind OpenGL to various windowing systems; initially the EGL API could only be used to create contexts that are compatible with the “OpenGL ES” subset of OpenGL, but this restriction has recently been lifted - a flag can now be passed to EGL to ask it to return a “full opengl-compatible” context. There are dynamic libraries that implement the EGL API for various windowing systems including X11 and Windows, ie an X client app just needs to load the appropriate one at runtime.

The [GLUT](#) API is older than EGL, and solves the same problem: a portable API for binding OpenGL to various windowing systems.

The Mesa project provides a single library `libGL` which provides the EGL and GLX apis. This library has support for using DRI and `libdrm` for direct rendering; the GLX implementation also supports “indirect” mode (see later). Mesa also provides open-source card-specific OpenGL implementations which `libGL` can load and integrate with.

Some vendors choose to provide their own OpenGL libraries rather than work with the Mesa project (eg the closed-source NVidia driver, the closed-source AMD Catalyst driver, and the open-source Intel driver). TODO: do these suppliers then need to implement EGL and GLX corresponding to their custom OpenGL implementations, or can `libGL` load these drivers anyway?

The Mesa libraries can also be [used on BSD](#) operating systems. The DRM driver code is linux-specific but generally released under a liberal license to permit porting to non-linux systems, so that the BSDs can also provide DRI-based accelerated graphics.

To display graphics in a “window”, the client first uses standard X calls to create a window, then uses either (a) DRI2 to get back the filename of a `/dev/dri/card*` node to open or (b) DRI3 to get back an already-open filehandle for that node. It then uses `libdrm` calls to manage buffers, and OpenGL calls (together with the correct card-specific driver) to generate data to put into those buffers. Eventually the buffer gets handled back to the X server for output. The `libdrm` primarily uses `ioctl()` calls to communicate with the DRM driver. `libdrm` provides a rather ugly API, with many functions relevant only to specific families of graphics cards - but is guaranteed to be backwards-compatible.

Indirect OpenGL with X

In “indirect 3D”, the client application uses the GLX (OpenGL-for-X) APIs to create a “GL context”, and then uses normal OpenGL calls on that context. These calls are simply passed over the network to the X server. On the server, these calls are then processed via OpenGL and `libdrm` in a similar manner to how X clients perform “direct 3D” rendering; this part of the xorg server is called [AIGLX](#).

Mesa’s `libGL` library supports indirect GLX; as GPU instructions are generated only on the server, there is no card-specific code needed on the client side.

3D graphics is very data-intensive, in particular when making heavy use of *textures*. The performance of “indirect OpenGL” is therefore not always acceptable - though it depends on the application and on the network bandwidth.

Wayland Clients

The diagram does not show any client application integrating with Wayland rather than an X server; that really needs another separate diagram which would make this article way too long.

However in general, the same principles apply. A client app opens a socket to a wayland-compatible server, and sends commands over the socket to manage windows. The wayland-compatible server sends back events such as keystrokes and mouse movements. However unlike X, no “drawing” commands are sent over the socket; instead the client uses `libdrm` to allocate buffers, uses card-specific libraries to generate GPU commands and other data into those buffers, and requests the graphics card to process these commands. When a buffer full of graphics is ready for display, the client then “passes control” of the buffer to the wayland-compatible server which then “composes” the buffer contents into the scan-out buffer.

Buffer handles can be passed across the socket between wayland client and server, but AFAIK the DRM module also provides ways for buffers to be “handed over” to the wayland server as soon as processing is complete.

There is technically no such thing as a “wayland server”; Wayland is a *protocol* rather than an actual implementation. Because the protocol is reasonably simple, implementation of the API is not a huge job. Typically a *window manager* implements the wayland protocol, ie when a client communicates with a “wayland-compatible server” it is actually talking directly to the desktop’s window-manager. This is somewhat different from X, where the X-server and window-manager are separate applications with a complex protocol between them.

Because Wayland clients are expected to use libdrm, they must be on the same host as the wayland server. Performing “remote graphics” is supported by having a *proxy* application which the Wayland server sees as a normal wayland client (ie something that passes it buffers full of graphics). The proxy can then implement any network protocol it desires to communicate with the real client app - ie network protocols have been factored out of the core display server. XWayland is a modified X server which talks normal X network protocol to clients, and acts like a normal Wayland client - allowing most X applications to run unmodified. Proxies for other network protocols may be added in the future, eg one based on rendering on the client and transferring compressed images to the server (a kind of remote-desktop-protocol).

Note that XWayland does not support 100% of the X API, in particular things related to status-bars, messing with keyboard events, etc. Such applications do need to be *adapted* to work properly with a wayland display server. Examples of such programs are screen-savers and login managers (“greeters”).

GPU Computation

Some math-intensive applications can be accelerated by using a GPU for things other than generating graphics for immediate output. Such applications can simply open a `/dev/dri/render*` node and use libdrm directly without ever connecting to an X (or Wayland) server.

X Server

DDX Drivers

The xorg server is split internally into a “front end” (DIX) and “back end” (DDX); on Linux xorg usually uses the “xfree86” DDX back end, which in turn loads a suitable card-specific userspace driver.

A traditional 100% xfree86 userspace driver relies on the fact that the Linux kernel will have already scanned the PCI bus and created datastructures to represent all the devices present on the bus(es), including the device ids and the requested memory ranges. This information can be seen under `/sys` (and the `lspci` command prints it out nicely). The driver can therefore use `/sys/` to determine which physical addresses correspond to the graphics card, open `/dev/mem` (which represents all *physical memory* on the system), and use `mmap` calls to map those physical pages into its (ie the X server’s) virtual memory. The driver then has a direct method of communication to the graphics card. Note that opening `/dev/mem` is restricted to *root* for obvious reasons; this means that with this approach the X server also needs to run as *root* - even though an X server instance typically “serves” just one user.

Originally, these xfree86 drivers would use this “direct method of communication” to configure the “display modes” of the card. All modern xfree86 drivers have now been updated to use libdrm to communicate (via `/dev/dri/card0`) with the in-kernel DRM driver instead, so that display mode information is centralized. I am not currently sure whether 2D drawing commands performed by DDX drivers also goes through libdrm or whether the DDX driver still communicates directly with the card.

If xfree86 drivers were to avoid using `/dev/mem` and instead only use libdrm to talk to the card, then running X as a “normal user” would also be possible (with [some complications](#)).

When an xfree86 driver opens `/dev/dri/card0` it immediately makes an `ioctl` systemcall to make itself the “master” of that node. A card node can be opened by multiple processes concurrently, but some functionality is only available to the process marked as the “master”.

Question: Somehow, the graphics card BARs are set up to map appropriate physical memory ranges to the card (the kernel doesn’t do this automatically AFAIK). I would guess that when a DRM driver for that card is available then that driver will configure the mappings. But how does this get done when using old-fashioned DDX drivers?

AIGLX

When a client app uses GLX in “indirect mode”, then OpenGL commands are passed from the client to the server. The xorg server then invokes the AIGLX component which uses libdrm and a card-specific OpenGL implementation to basically do what a client in “direct mode” would have done.

Other Topics

? maybe more info on GLAMOR ?

? displaying video: VDPAU, VA-API, Xvideo/Xv ?

? brief intro to X “window managers” ?

DDX drivers originally wrote *directly* into the frame-buffer. A driver was expected to *clip and translate* its output to match the visible part of the relevant “window”. No “compositor” was possible; instead when a window moved then an

“expose event” was sent to the owner of the “underlying window” to draw the necessary section. Hacks were later added to redirect output into per-window buffers, and a compositor would then *merge* the buffers (max once per frame). Each window actually has two buffers: the *front buffer* which is the one that gets copied to the framebuffer, and the *back buffer* which is not visible until the app owning the window performs a *buffer switch*. Compositing is a more *secure* manner of graphics than allowing applications to write direct to the framebuffer; it ensures that an app cannot overwrite output from other apps (eg hide warnings), cannot pretend to be other apps (eg draw a popup “enter network password” window), and cannot *read* graphics generated by other apps (eg password input fields). It is because the “target buffer” was originally a part of the framebuffer that DRI1/DRI2 originally had the X server allocate buffers, and the client then simply “gained access” to them. DRI3 instead has the client using the DRM APIs to allocate buffers, and then “transfer” them to the display server; this *requires* the display server to perform composition (ie cannot use the “fast path” of having a client write directly into the framebuffer) - but the advantages of composition are so high that all modern systems (even embedded ones) do that anyway.

Some graphics hardware supports “composing” multiple framebuffers in hardware at scanout time - ie there can be multiple framebuffers. This can reduce the amount of copying a compositor needs to do; as long as there are not *too many* windows active, then the hardware can composite directly from the per-window buffers. Or at least, desktop-background, cursor and all-other-windows can be in separate buffers. Such hardware is common in embedded systems (where power is important and IO bandwidth is limited); the different buffers accessed directly by the hardware are sometimes called “sprites”.

AFAIK, a Gallium3D “State Tracker” is the front-end part of a gallium3d-based driver. The state-tracker provides the API to the client application (eg provides the OpenGL API, or OpenVG, or Direct3D, or other). It then generates appropriate calls to the Gallium3D back-end which generates a card-independent stream of data which is then passed to a card-specific library to generate card-specific instruction and data streams. TODO: add more info on Gallium3D.

A 3D “texture” can be used to perform the equivalent of a 2D “bitblit”: executing a 3D operation to draw a flat polygon with a specified texture-image places the specified texture unaltered into the output buffer at the polygon’s coordinates. Rotating/skewing the specified polygon will transform the specified texture. This is used for example to display individual frames of video.

The following projects/acronyms are completely obsolete and can be ignored if you see them referenced elsewhere:

- xgl - replaced by AIGLX
- glitz - replaced by GLAMOR?
- compiz

[Freedesktop.org](http://freedesktop.org) is a “hub” for development of open-source graphics-related software on unix-like systems. [Many projects](#) are hosted there, including DRI, Glamor, Cairo, libinput - though many of these projects have their primary home-page on a dedicated domain.

References

Update: The official kernel docs for graphics are now *much* better than they were when this article was originally written:

- [Web version of the official Linux kernel graphics documentation](#)

There are other articles on this site that are related to graphics:

- [Detailed Info on Modern Graphics Cards](#)
- [Detailed Info on Linux Framebuffer Drivers](#)
- [Detailed Info on DRM/KMS Implementation](#)
- [The PCI Bus](#)


Information in this article has been pulled from *many* sources. Here are a few of them:

- [Mecheye: The Linux Graphics Stack](#) - a well-written article that covers many of the same topics as this article, but in less detail
- [Stephane Marchesin: Linux Graphics Drivers: an Introduction](#) - general info about the linux graphics stack. Sadly very incomplete.
- [Intel: Intel Linux Graphics Stack](#) - general info about the linux graphics stack
- [Iago Toral: Brief Introduction to the Linux Graphics Stack](#). There are many articles on this blog which provide *excellent* introductions to 3D rendering and modern graphics chips.
- [Iago Toral: Diving into Mesa](#)
- [Wikipedia on DRI](#)
- [DRM Developers Guide](#) - from 2012. Very detailed discussion of internal DRM details.
- [DRM Memory Management](#) - very detailed info on GEM/TTM
- [GL Dispatch in Mesa](#) - some information about Mesa’s implementation, and about the OpenGL APIs in general.

- [Xorg dri/drm page](#) - appears quite out-of-date
- [Xorg drm redesign](#) - proposal for what became the current DRM architecture
- [LWN: GEM v. TTM](#) - somewhat obsolete (written while GEM/TTM were under development) but has some interesting background info
- [LWN: Memory management for graphics processors](#) - Nov 2007
- [DRM article from freedesktop.org wiki](#) - currently describes DRMv1 only
- [EGL specification](#) - the official specification of the Khronos EGL API (see p42 in particular)
- [Jon Smirl: The State of Linux Graphics](#) - Jon Smirl, 2005
- [Nouveau Introductory Course](#)
- [Keith Packard: Sharpening the Intel Driver Focus](#) - from 2009
- [xorg glossary](#)
- [xfree86 DDX Design](#) - 2010. Excellent low-level information on the internals of the xorg DDX layer, including how userspace drivers interact with graphics cards.
- [Wikipedia on graphics hardware](#)
- [Wikipedia on the Xvideo extension](#)
- [Wikipedia on Open Source Graphics Drivers](#)
- [Mecheye: XWayland](#). Not just information on Xwayland, but also a nice overview of the whole X environment
- [Gallium3D Architecture](#)
- [Martin Fiedler: Linux Graphics Demystified](#) - from 2014.
- [Techspot: history of the GPU](#)
- [Fabien Giesen: A trip through the graphics pipeline](#)
- [Nvidia: Life of a triangle](#)
- [Phoronix: DRM Janitors](#)

2 Comments

Mine of Information

 Login ▾ Recommend 1 Tweet Share

Sort by Oldest ▾



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS **Animanteum** • 3 years ago

First of all, Thank you for this very useful explanation of the Linux graphics stack. This document focuses primarily on open source components, but you also touched a bit on NVIDIA's proprietary drivers. Do you know of a good resource that goes into detail about how they work?

  • Reply • Share >**Simon Kitching** Mod  Animanteum • 3 years ago

Sorry, I don't know of any NVIDIA-specific reference info.

  • Reply • Share >

ALSO ON MINE OF INFORMATION

A JUnit Rule for Elasticsearch Integration Testing

1 comment • 2 years ago

srikanth krishnamurthy — Hey, thank you for sharing this. can you please let me know what does the ESIndicesLoader refer to? Looks like it is missing in the

Mine of Information - Kafka Manager

1 comment • 2 years ago

Vasu Jinagam — Good one :)

Beginner's Guide to Installing from Source

14 comments • 3 years ago

Ardhia Mangku Ikhsan — linuxfromscratch brought me here

Mine of Information - Simple TOTP on Linux

1 comment • 6 months ago

TadFisher — `pass-otp` author here. It is packaged in Debian and it does not contain native code whatsoever, as it's a pure Bash script. It does depends on oath-toolkit