

Linux Graphics Drivers: an Introduction

Version 3

Stéphane Marchesin
<stephane.marchesin@gmail.com>

March 15, 2012

Contents

1	Introduction	7
1.1	Book overview	7
1.2	What this book does not cover	8
2	A Look at the Hardware	9
2.1	Hardware Overview	9
2.2	Bus types	10
2.3	Virtual and Physical Memory	12
2.4	Anatomy of a Graphics Card	15
2.5	Programming the card	20
2.6	Graphics Hardware Examples	21
2.6.1	Forward renderers	21
2.6.2	Deferred Renderers	22
3	The Big Picture	25
3.1	The X11 infrastructure	25
3.2	The DRI/DRM infrastructure	26
4	Framebuffer Drivers	31
4.1	Creating a framebuffer driver	31
4.2	Framebuffer operations	32

CONTENTS

5	The Direct Rendering Manager	35
5.1	DRM batch buffer submission model	36
5.1.1	Hardware sharing	36
5.1.2	Memory management and security	37
5.2	Modesetting	37
5.3	libdrm	38
6	X.Org Drivers	39
6.1	Creating a basic driver	39
6.2	ShadowFB acceleration	40
6.3	XAA acceleration	41
6.4	EXA acceleration	41
7	Video Decoding	45
7.1	Video Standards	45
7.2	Video decoding pipeline	45
7.2.1	Entropy	45
7.2.2	Inverse DCT	46
7.2.3	Motion Compensation	46
7.2.4	Color Space Conversion	46
7.3	Video decoding APIs	49
8	OpenGL	51
8.1	The OpenGL Rendering Pipeline	51
8.1.1	Vertex processing	51
8.1.2	Geometry processing	51
8.1.3	Fragment processing	51
9	Mesa	53
9.1	Mesa	53
9.2	Mesa internals	53
9.2.1	Textures in mesa	53

10 Gallium 3D	55
10.1 Gallium3D: a plan for a new generation of hardware	55
10.2 State trackers	55
10.3 Pipe driver	56
10.4 Winsys	56
10.5 Vertex submission	56
10.6 Writing Gallium3D drivers	56
10.7 Shaders in Gallium	56
11 GPU Computing	59
12 Suspend and Resume	61
13 Technical Specifications	63
13.1 Obtaining official specifications	63
13.2 Reverse engineering	64
14 Beyond Development	67
14.1 Testing for conformance	67
14.2 Debugging	67
14.3 Upstreaming	68
15 Conclusions	69

CONTENTS

Introduction

Accelerating graphics is a complex art which suffers a mostly unjustified reputation of being voodoo magic. This book is intended as an introduction to the inner workings and development of graphics drivers under Linux. Throughout this whole book, knowledge of C programming is expected, along with some familiarity with graphics processors. Although its primary audience is the graphics driver developer, this book details the internals of the full Linux graphics stack and therefore can also be useful to application developers seeking to enhance their vision of the Linux graphics world: one can hope to improve the performance of one's applications through better understanding the Linux graphics stack. In this day and age of pervasive 3D graphics and GPU computing, a better comprehension of graphics is a must have!

1.1 Book overview

The book starts with an introduction of relevant hardware concepts (Chapter 2). Only concepts directly relevant to the graphics driver business are presented there. Then we paint a high-level view of the Linux graphics stack in Chapter 3 and its evolution over the years. Chapter 4 introduces framebuffer drivers, a basic form of graphics drivers under Linux that, although primitive, sees wide usage in the embedded space. Chapter 5 introduces the Direct Rendering Manager (or DRM), a kernel module which is in charge of arbitrating all graphics activity going on in a Linux system. The next chapter (Chapter 6) focuses on X.Org drivers and the existing acceleration APIs available to the developer. Video decoding sees its own dedicated part in Chapter 7. We then move on to 3D acceleration with Chapter 8 where we introduce the basic concepts of OpenGL. Chapter 9 and 10 are dedicated to Mesa and Gallium 3D, the two foundations of 3D graphics acceleration under Linux used as the framework for 3D drivers. Chapter 11 tackles an emerging field, GPU computing. Next, we discuss suspend and resume in Chapter 12. We then discuss two side issues with Linux graphics drivers: technical specifications in Chapter 13 and what you should do aside pure development in Chapter 14. Finally, we conclude in Chapter 15.

Each chapter finishes with the “takeaways”, a number of relevant points that we made during said chapter.

1.2 What this book does not cover

Computer graphics move at a fast pace, and this book is not about the past. Obsolete hardware (isa, vlb, ...), old standards (the vga standard and its dreadful int10, vesa), outdated techniques (user space modesetting) and old X11 servers (Xsun, XFree86, KDrive...) will not be detailed.

A Look at the Hardware

Before diving any further into the subject of graphics drivers, we need to understand the graphics hardware. This chapter is by no means intended to be a complete description of all the inner workings of your average computer and its graphics hardware, but only as an introduction thereof. The goal of this section is just to “cover the bases” on the knowledge we will require later on. Notably, most hardware concepts that will subsequently be needed are introduced here. Although we sometimes have to go through architecture-specific hoops, we try to stay as generic as possible and the concepts detailed thereafter should generalize well.

2.1 Hardware Overview

Today all computers are architected the same way: a central processor and a number of peripherals. In order to exchange data, these peripherals are interconnected by a bus over which all communications go. Figure 2.1 outlines the layout of peripherals in a standard computer.

The first user of the bus is the CPU. The CPU uses the bus to access system memory and other peripherals. However, the CPU is not the only one able to write and

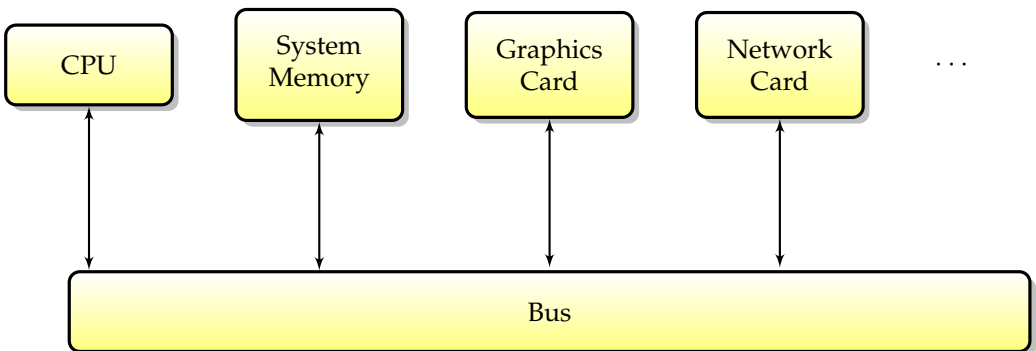


Figure 2.1: Peripheral interconnection in a typical computer.

2. A Look at the Hardware

read data to the peripherals, the peripherals themselves also have the capability to exchange information directly. In particular, a peripheral which has the ability to read and write to memory without the CPU intervention is said to be DMA (Direct Memory Access) capable, and the memory transaction is usually called a DMA. This type of transaction is interesting, because it allows the driver to use the GPU instead of the CPU to do memory transfers. Since the CPU doesn't need to actively work any more to achieve those transfers, and since it allows better asynchronicity between the CPU and the GPU, better performance can be attained. Common uses of DMA include improving the performance of texture uploads or streaming video. Today, all graphics processors feature this ability (named DMA bus mastering) which consists in the card requesting and subsequently taking control of the bus for a number of microseconds.

If a peripheral has the ability to achieve DMA to or from an un contiguous list of memory pages (which is very convenient when the data is not contiguous in memory), it is said to have DMA scatter-gather capability (as it can scatter data to different memory pages, or gather data from different pages).

Notice that the DMA capability can be a downside in some cases. For example on real time systems, this means the CPU is unable to access the bus while a DMA transaction is in progress, and since DMA transactions happen asynchronously this can lead to missing a real time scheduling deadline. Another example is small DMA memory transfers, where the CPU overhead of setting up the DMA is greater than the gain in asynchronicity and therefore transfers slow down. So while DMA has a lot of advantages from a performance viewpoint, there are situations where it should be avoided.

2.2 Bus types

Buses connect the machine peripherals together; each and every communication between different peripherals goes over (at least) one bus. In particular, a bus is the way most graphics card are connected to the rest of the computer (one notable exception being the case of some embedded systems, where the GPU is directly connected to the CPU). As shown in Table 2.2, there are many bus types suitable for graphics: PCI, AGP, PCI-X, PCI-express to name a (relevant) few. All the bus types we will detail are variants of the PCI bus type, however some of them feature singular improvements over the original PCI design.

PCI (Peripheral Component Interconnect)

PCI is the most basic bus allowing connecting graphics peripherals today. One of its key feature is called bus mastering. This feature allows a given peripheral to take hold of the bus for a given number of cycles and do a complete transaction (called a DMA, Direct Memory Access). The PCI bus is coherent, which means that no explicit flushes are required for the memory to be coherent across devices.

Bus type	Bus width	Frequency	Bandwidth	Capabilities
PCI	32 bits	33 Mhz	133 MB/s (33 Mhz)	-
AGP	32 bits	66 Mhz	2100 MB/s (8x)	SBA, FW, GART
PCI-X	64 bits	33, 66, 133 Mhz	533 MB/s (66 Mhz)	-
PCI-Express (1.0)	Serial	1.25 Ghz	4 GB/s (16 lanes)	-
PCI-Express (3.0)	Serial	4 Ghz	16 GB/s (16 lanes)	-

Figure 2.2: Common bus types.

AGP (Accelerated Graphics Port)

AGP is essentially a modified PCI bus with a number of extra features compared to its ancestor. Most importantly, it is faster thanks to a higher clock speed and the ability to send 2, 4 or 8 bits per lane on each clock tick (for AGP 2x, 4x and 8x respectively). AGP also three distinctive features:

- ▷ The first feature is AGP GART (Graphics Aperture Remapping Table), a simple form of IOMMU (as will be seen in section 2.3). It allows taking a (non contiguous) set of physical memory pages out of system memory and exposing it to the GPU for its use as a contiguous area. This increases the amount of memory usable by the GPU at little cost, and creates a convenient area for sharing data between the CPU and the GPU (AGP graphics cards can do fast DMA to/from this area, and since the GART area is a chunk of system RAM, CPU access is a lot faster than VRAM). One notable drawback is that the GART area is not coherent, and therefore writes to GART (be it from the GPU or CPU) need to be flushed before transactions from the other party can begin. Another drawback is that only a single GART area is handled by the hardware, and it has to be sub-allocated by the driver.
- ▷ The second feature is AGP side band addressing (SBA). Side band addressing consists in 8 extra bus bits used as an address bus. Instead of multiplexing the bus bandwidth between addresses and data, the nominal AGP bandwidth can be dedicated to data only. This feature is transparent to the driver developer.
- ▷ The third feature is AGP Fast Writes (FW). Fast writes allow sending data to the graphics card directly, without having the card initiate a DMA. This feature is also transparent for the driver developer.

Keep in mind that these last two features are known to be unstable on a wide range of hardware, and oftentimes require chipset-specific hacks to work properly. Therefore it is advisable not to enable them. In fact, they are an extremely frequent cause for strange hardware errors on AGP cards.

PCI-X

PCI-X was developed as a faster PCI for server boards, and very few graphics peripherals exist in this format (some Matrox G550 cards). It is not to be confused with PCI-Express, which sees real widespread usage.

PCI-Express (PCI-E)

PCI-Express is the new generation of PCI devices. It has more advantages than a simple improved PCI.

Finally, it is important to note that, depending on the architecture, the CPU-GPU communication does not always relies on a bus. This is especially common on embedded systems where the GPU and the CPU are on a single die. In that case the CPU can access the GPU registers directly.

2.3 Virtual and Physical Memory

The term “memory” has to two main different acceptions:

- ▷ Physical memory. Physical memory is real, hardware memory, as stored in the memory chips.
- ▷ Virtual memory. Virtual memory is a translation of physical memory addresses allowing user space applications to see their allocated chunks as if they were contiguous while they are fragmented and scattered on the chips.

In order to simplify programming, it is easier to handle contiguous memory areas. It is easy to allocate a small contiguous area, but allocating a bigger memory chunk would require as much contiguous physical memory which is difficult if not impossible to achieve shortly after bootup because of memory fragmentation. Therefore, a mechanism is required to keep the appearance of a contiguous piece of memory to the application while using scattered pieces.

To achieve this, memory is split into pages. For the scope of this book, it is sufficient to say that a memory page is a collection contiguous bytes in physical memory¹In order to make a scattered list of physical pages seem contiguous in virtual space, a piece of hardware called MMU (memory mapping unit) converts virtual addresses (used in applications) into physical addresses (used for actually accessing memory) using a page table as shown on Figure 2.3. In case a page does not exist in virtual space (and therefore not in the MMU table), the MMU is able to signal it, which provides the basic mechanism for reporting access to non-existent memory areas. This in turn is used by the system to implement advanced memory programming like swapping or on-the-fly page instantiations. As the MMU is only effective for CPU access to memory, virtual

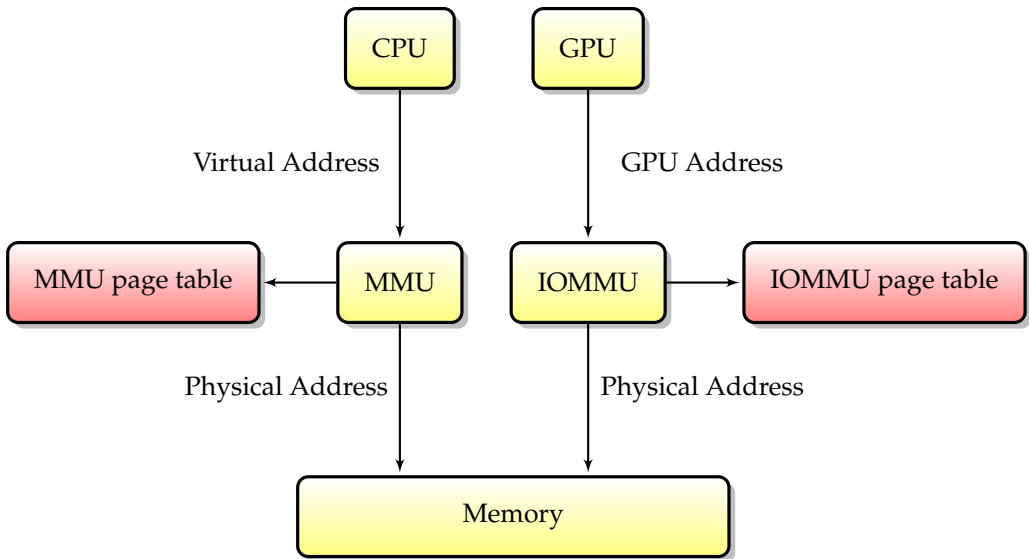


Figure 2.3: MMU and IOMMU.

addresses are not relevant to the hardware since it is not able to match them to physical addresses.

While the MMU only works for CPU accesses, it has an equivalent for peripherals: the IOMMU. As shown on figure 2.3, an IOMMU is the same as an MMU except that it virtualizes the address space of peripherals. The IOMMU can see various incarnations, either on the motherboard chipset (in which case it is shared between all peripherals) or on the graphics card itself (where it will be called AGP GART, PCI GART). The job of the IOMMU is to translate memory addresses from the peripherals into physical addresses. In particular, this allows “fooling” a device into restricting its DMAs to a given range of memory and it is required for better security and hardware virtualization.

A special case of IOMMU is the Linux swiotlb which allocates a contiguous piece of physical memory at boot (which makes it feasible to have a large contiguous physical allocation since there is no fragmentation yet) and uses it for DMA. As the memory is physically contiguous, no page translation is required and therefore a DMA can occur to and from this memory range. However, this means that this memory (64MB by default) is preallocated and will not be used for anything else.

AGP GART is another special case of IOMMU present with AGP graphics cards which exposes a single linear area to the card. In that case the IOMMU is embedded in the AGP chipset, on the motherboard. The AGP GART area is exposed as a linear area of virtual memory to the system.

Yet another special case of IOMMU is the PCI GART present on some GPUs, which allows exposing a chunk of system memory to the card. In that case the IOMMU table

¹On x86 and x86-64, a page is usually 4096 bytes long, although different sizes are possible on other architectures or with huge pages.

2. A Look at the Hardware

is embedded in the graphics card, and often the physical memory used does not need to be contiguous.

Obviously, with so many different memory types, performance is not homogeneous; not all combination of accesses are fast, depending on whether they involve the CPU, the GPU, or bus transfers. Another issue which arises is memory coherence: how can one ensure that memory is coherent accross devices, in particular that data written by the CPU is available to the GPU (or the opposite). These two issues are correlated, as higher performance usually means a lower level of memory coherence, and vice-versa.

As far as setting the memory caching parameters goes, there are two ways to set caching attributes on memory ranges:

- ▷ MTRRs. An MTRR (Memory Type Range Register) is a register describing attributes for a range of given physical memory. Each MTRR consists in a starting physical address, a size, and a caching type. The number of MTRR depends on the system, but is very limited. Although this applies to a physical memory range, the effect works on the corresponding virtual memory pages. This for example makes it possible to map pages with a specific caching type.
- ▷ PAT (Page Attribute Table) allows setting per-page memory attributes. Instead of relying on a limited number of memory ranges like with MTRRs, it is possible to specify caching attributes on a per-page basis. However it is an extension only available on recent x86 processors.

On top of these, one can use explicit caching instructions on some architectures, for example on x86 *movntq* is an uncached mov instruction and *clflush* can selectively flush cache lines.

There are three caching modes, usable both through MTRRs and PAT on system memory:

- ▷ UC (UnCached) memory is uncached. CPU read/writes to this area are uncached, and each memory write instruction triggers an actual immediate memory write. This is helpful to ensure that information has been actually written so as to avoid CPU/GPU race conditions.
- ▷ WC (Write Combine) memory is uncached, but CPU writes are combined together in order to improve the performance. This is useful to improve performance in situations where uncached memory is required, but where combining the writes together has no adverse effects.
- ▷ WB (Write Back) memory is cached. This is the default mode and leads to the best performance for CPU accesses. However this does not ensure that memory writes are propagated to central memory after a finite time.

Notice that these caching modes apply to the CPU only, the GPU accesses are not directly affected by the current caching mode. However, when the GPU has to access an area of memory which was previously filled by the CPU, uncached modes ensure that

the memory writes are actually done, and are not pending sitting in a CPU cache. Another way to achieve the same effect is the use of cache flushing instructions present on some x86 processors (like `cflush`). However this is less portable than using the caching modes. Yet another (portable) way is the use of memory barriers, which ensures that pending memory writes have been committed to main memory before moving on.

Obviously with so many different caching modes, not all accesses have the same performance:

- ▷ When it comes to CPU access to system memory, uncached mode provides the worst performance, write back provides the best performance, and write combine is in between.
- ▷ When the CPU accesses the video memory from a discrete card, all accesses are extremely slow, be they reads or writes, as each access needs a cycle on the bus. Therefore it is not recommended to access large areas of VRAM with the CPU. Furthermore on some GPUs synchronizing is required or this could cause a GPU hang.
- ▷ Obviously the GPU accessing VRAM is extremely fast.
- ▷ GPU access to system ram is unaffected by the caching mode, but still has to go over the bus. This is the case of DMA transactions. As those happen asynchronously, they can be considered “free” from the viewpoint of the CPU, however there is a non-negligible setup cost involved for each DMA transaction. This is why, when transferring small amounts of memory, a DMA transaction is not always better than a direct CPU access.

Finally, one last important point to make about memory is the notion of memory barriers and write posting. In the case of a cached (Write Combine or Write Back) memory area, a memory barrier ensures that pending writes have actually been committed to memory. This is used, for example, before asking the GPU to read a given memory area. For I/O areas, a similar technique called write posting exists: it consists in doing a dummy read inside the I/O area which will, as a side effect, wait until pending writes have taken effect before completing.

2.4 Anatomy of a Graphics Card

Today, a graphics card is basically a computer-in-the-computer. It is a complex beast with a dedicated processor on a separate card, and features its own computation units, its own bus, and its own memory. This section gives an overview of a graphics card, including those elements.

Graphics Memory

The GPU’s memory, which we will from now on refer to as video memory, can be either real, dedicated, on-card memory (in the case of a discrete card), or memory shared

2. A Look at the Hardware

with the CPU (also known as “stolen memory” or “carveout” in the case of an integrated card). Notice that the case of shared memory has interesting implications, as it means that system to video memory copies can be virtually free if implemented properly. In the case of dedicated memory it means that transfers back and forth will need to happen, and they will be limited by the speed of the bus.

It is not uncommon for modern GPUs to feature a form of virtual memory as well, allowing to map different resources (real video memory or system memory) into the GPU address space. This is very similar to the CPU’s virtual memory, but uses a completely separate hardware implementation. For example, older Radeon cards (actually since Rage 128) feature a number of surfaces which you can map into the GPU address space, each of which is a contiguous memory resource (video ram, AGP, PCI). Old Nvidia cards (everything up to NV40) have a similar concept based on objects which describe an area of memory which can then be bound to a given use. Recent cards (starting with NV50 and R800) let you build the address space page by page, with the ability of picking system and dedicated video memory pages at will. The similarity of these with a CPU virtual address space is very striking, in fact unmapped page access can be signaled to the system through an interrupt and acted upon in a video memory page fault handler. However, be careful playing with those as the implication here is that driver developers have to juggle with multiple address spaces from the CPU and GPU which are going to be fundamentally different.

Surfaces

Surfaces are the basic sources and targets for all rendering. Although they can be called differently (textures, render targets, buffers...) the basic idea is always the same. Figure 2.4 depicts the layout of a graphics surface. The surface width is rounded up to what we call the pitch because of hardware limitations (usually to the next multiple of some power of 2) and therefore there exists a dead zone of pixels which goes unused. The graphics surface has a number of characteristics:

- ▷ The pixel format of the surface. A pixel color is represented memory by its red, green and blue components, plus an alpha component used as the opacity for blending. The number of bits for a whole pixel usually matches hardware sizes (8,16 or 32 bits) but the repartition of the bits between the four components does not have to match those. The number of bits used for each pixels is referred to as bits per pixel, or *bpp*. Common pixel formats include 888 RGBX, 8888 RGBA, 565 RGB, 5551, RGBA, 4444 RGBA. Notice that most cards today work natively in ABGR 8888.
- ▷ Width and height are the most obvious characteristics, and are given in pixels.
- ▷ The pitch is the width in bytes (not in pixels!) of the surface, including the dead zone pixels. The pitch is convenient for computing memory usages, for example the size of the surface should be computed by $height \times pitch$ and not $height \times width \times bpp$ in order to include the dead zone.

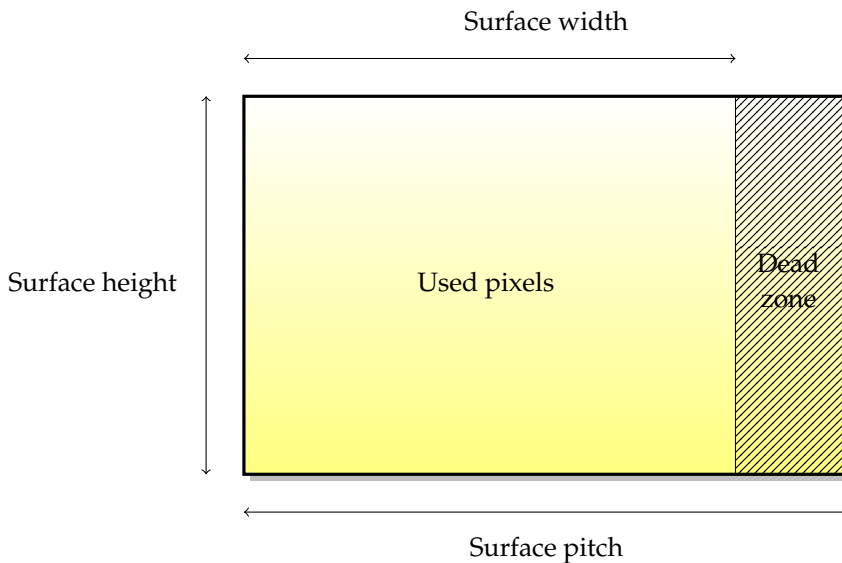


Figure 2.4: The layout of a surface.

Notice that surfaces are not always stored linearly in video memory, in fact for performance reasons it is extremely common that they are not, as this improves the locality of the memory accesses when rendering. Such surfaces are called *tiled*. The exact layout of a tiled surface is highly dependent on the hardware, but is usually a form of space-filling curve like the Z curve or Hilbert's curve.

2D Engine

The 2D engine, or blitter, is the hardware used for 2D acceleration. Blitters have been one of the earliest form of graphics acceleration and are still extremely widespread today. Generally, a 2D engine is capable of the following operations:

- ▷ Blits. Blits are a copy of a memory rectangle from one place to another by the GPU. The source and destination can be either video or system memory.
- ▷ Solid fills. Solid fills consist in filling a rectangle memory area with a color. Note that this can also include the alpha channel.
- ▷ Alpha blits. Alpha blits use the alpha component of pixels from of a surface to achieve transparency [porter & duff].
- ▷ Stretched blits.

Figure 2.5 shows an example of blitting a rectangle between two different surfaces. This operation is defined by the following parameters: the source and destination coordinates, the source and destination pitches, and the blit width and height. However,

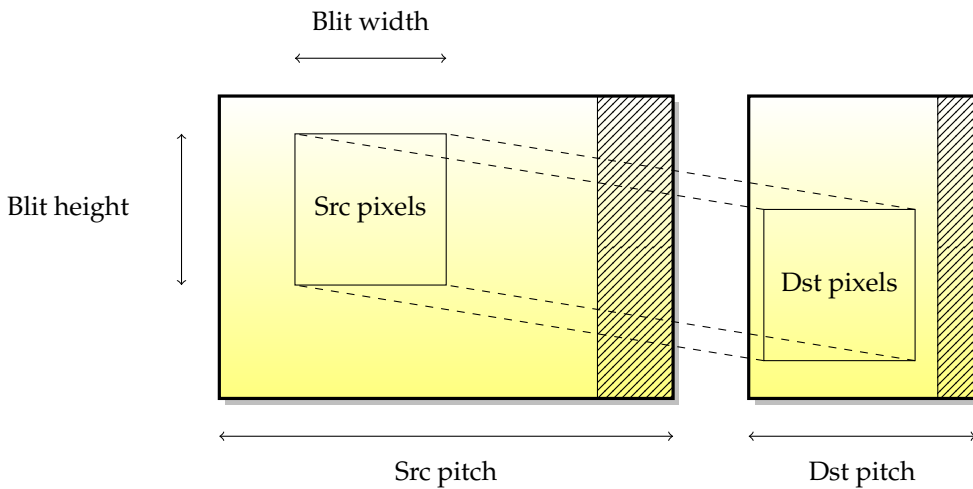


Figure 2.5: Blitting between two different surfaces.

this is limited to 2D coordinates, usually no perspective or transformation is possible with a blitting engine.

When a blit happens between two overlapping source and destination surfaces, the semantics of the copy is not trivially defined, especially if one considers that what happens for a blit is not a simple move of a rectangle, but is done pixel-by-pixel at the core. As seen on Figure 2.6, if one does a line-by-line copy top to bottom, some source pixels will be modified as a side effect. Therefore, the notion of blitting direction was introduced into the blitters. In this case, for a proper copy a bottom to top copy is required. Some cards will determine the blitting direction automatically according to surface overlap (for example nvidia GPUs), and others will not, in which case this has to be handled by the driver. This is why some GPUs actually support negative pitches in order to tell the 2D engine to go backwards.

Finally, keep in mind that not all current graphics accelerators feature a 2D engine. Since 3D acceleration is technically a super-set of 2D acceleration, it is possible to implement 2D acceleration using the 3D engine (and this idea is one of the core ideas behind the Gallium 3D design, which will be detailed in Chapter 10). And indeed some drivers use the 3D engine to implement 2D which allows GPU makers to completely part with the transistors otherwise dedicated to it. Yet some other cards do not dedicate the transistors, but microprogram 2D operations on top of 3D operations inside the GPU (this is the case for nVidia cards since nv10 and up to nv50, and for the Radeon R600 series which have an optional firmware that implements 2D on top of 3D). This sometimes has an impact on mixing 2D and 3D operations since those now share hardware units.

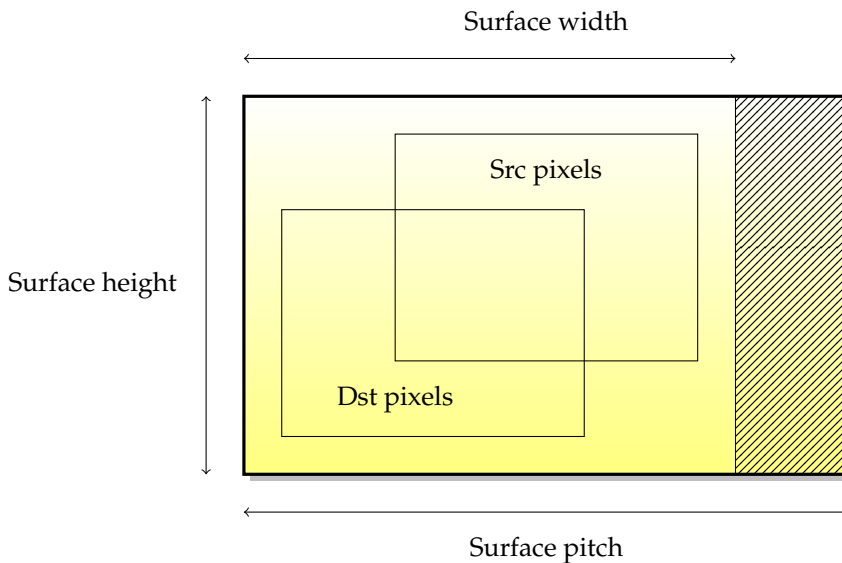


Figure 2.6: Overlapping blit inside a surface.

3D Engine

A 3D engine is also called rasterization engine. It contains a series of stages which exchange data in a pipeline (1-directional) fashion.

vertex -> geom -> fragment

graphics fifo

DMA

<http://www.x.org/wiki/Development/Documentation/HowVideoCardsWork>

To attain better cache locality, the textures and surface are often tiled. Tiling means that the texture isn't stored linearly in GPU memory, but instead is stored so as to make pixels which are close in texture space also close in memory space. Examples are the Z-order curve and the Hilbert curve.

Overlays and hardware sprites

Scanout

The last stage of a graphics display is presenting the information onto a display device, or screen.

Display devices are the last link of the graphics chain. They are charged with presenting the pictures to the user.

digital vs analog signal

hsync, vsync

sync on green

Connectors and encoders: CRT, TMDS, LVDS, DVI-I, DVI-A, DVI-D, VGA (D-SUB 15 is the proper name)

2.5 Programming the card

Each PCI card exposes a number of PCI resources; *lspci -v* lists these resources. These can be, but are not limited to, BIOSes, MMIO ranges, video memory (or only some part of it). As the total PCI resource size is limited, oftentimes a card will only expose part of its video memory as a resource, and the only way to access the remaining memory is through DMA from other, reachable areas (in a way similar to bounce pages). This is increasingly common as the video memory sizes keep growing while the PCI resource space stays limited.

MMIO

MMIO is the most direct access to the card. A range of addresses is exposed to the CPU, where each write goes directly to the GPU. This allows the simplest form of communication of commands from the CPU to the GPU. This type of programming is synchronous; writes are done by the CPU and executed on the GPU in a lockstep fashion. This leads to sub-par performance because each access turns into a packet on the bus and because the CPU has to wait for previous GPU commands to complete before submitting subsequent commands. For this reason MMIO is only used in the non-performance critical paths of today's drivers.

DMA

A direct memory access (DMA) is the use by a peripheral of the bus mastering feature of the bus. This allows one peripheral to talk directly to another, without intervention from the CPU. In the graphics card case, the two most common uses of DMAs are:

- ▷ Transfers by the GPU to and from system memory (for reading textures and writing buffers). This allows implementing things like texturing over AGP or PCI, and hardware-accelerated texture transfers.
- ▷ The implementation of command FIFO. As MMIO between the CPU and GPU is synchronous and graphics drivers inherently use a lot of I/O, a faster means of communicating with the card is required. The command FIFO is a piece of memory (either system memory or more rarely video memory) shared between the graphics card and the CPU, where the CPU places command for later execution by the GPU. Then the GPU reads the FIFO asynchronously using DMA and

executes the commands. This model allows asynchronous execution of the CPU and GPU command flows and thus leads to higher performance.

Interrupts

Interrupts are a way for hardware peripherals in general, and GPUs in particular, to signal events to the CPU. Usage examples for interrupts include signaling completion of a graphics command, signaling a vertical blanking event, reporting a GPU error, ... When an interrupt is raised by the peripheral, the CPU executes a small routine called an interrupt handler, which preempts other current executions. There is a maximum execution time for an interrupt handler, so the drivers have to keep it short (not more than a few microseconds). In order to execute more code, the common solution is to schedule a tasklet from the interrupt handler.

2.6 Graphics Hardware Examples

2.6.1 Forward renderers

Forward renderers (i.e. classical renderers) are GPU which render the primitives as they are submitted to the rendering API, and for each of those primitives it is drawn entirely before moving on to the next one. This is the most straightforward way of rendering 3D primitives. This is the approach used in most desktop-range GPUs.

ATI

Shader engine 4+1

Nvidia

NVidia hardware has multiple specificities compared to other architectures. The first one is the availability of multiple contexts, which is implemented using multiple command fifos (similar to what some high-end infiniband networking cards do) and a context switching mechanism to commute between those fifos. A small firmware is used for context switches between contexts, which is responsible for saving the graphics card state to a portion of memory and restoring another context. A scheduling system using the round robin algorithm handles the selection of the contexts, and the timeslice is programmable.

The second specificity is the notion of graphics objects. Nvidia hardware features two levels of GPU access: the first one is at the raw level and is used for context switches, an the second one is the graphics objects which microprogram the raw level to achieve high level functionality (for example 2D or 3D acceleration).

Shader engine nv40/nv50

<http://nouveau.freedesktop.org/wiki/HonzaHavlicek>

2.6.2 Deferred Renderers

Deferred renderers are a different design for GPUs. Instead of rendering each 3D primitive as it is submitted by the rendering API, the driver stores it in memory and when it notices the end of a frame, it issues a single hardware call to render the whole scene. This has a number of advantages over classic architectures:

- ▷ Much better rendering locality can be achieved by splitting the screen into tiles (usually in the 16×16 to 32×32 pixel range). The GPU can then iterate over these tiles, and for each of those can resolve per-pixel depth in an internal (mini) zbuffer. Once the whole tile is rendered it can be written back to video memory, saving precious bandwidth. Similarly, since visibility is determined before fetching texture data, only the useful texture data is read (again saving bandwidth) and the fragment shaders are only executed for visible fragments (which saves computation power).
- ▷ If the depth buffer values are not required, they don't need to be written to memory. The depth buffer resolution can happen per-tile inside the GPU and never be written back to video memory, therefore saving video memory bandwidth and space.

Of course tiled renders require the ability to store the whole scene in memory before starting, and will also add latency since you need to wait for an end of frame even before you start drawing. The latency problem can be partially hidden by drawing a given frame on the GPU while the driver already allows the application to submit data for the next frame. However in some situations (readbacks, cross-process synchronization) it's not always possible to avoid it.

All in all, the deferred renderers are particularly useful for embedded platforms where the bandwidth is generally very scarce and the applications are simple enough that the additional latency and the limitations of the approach don't matter.

SGX

The SGX is an example of a deferred rendering GPU. It uses a tiling architecture.

The SGX shaders combine blending and depth test

Another example of a deferred renderer is the Mali family of GPUs.

Takeaways:

- ▷ There are multiple memory domains in a computer, and they are not coherent.
- ▷ A GPU is a completely separate computer with its own bus, address space and computational units.
- ▷ Communication between the CPU and GPU is achieved over a bus, which has non-trivial performance implications.
- ▷ GPUs can be programmed using two modes: MMIO and command FIFOs.
- ▷ There is no standard output method for display devices.

The Big Picture

The Linux graphics stack has seen numerous evolutions over the years. The purpose of this section is to detail that history, as well as give the justification behind the changes which have been made over the years. Today, the design is still strongly rooted in that history, and this section will explain that history to better motivate the current design of the Linux graphics stack.

3.1 The X11 infrastructure

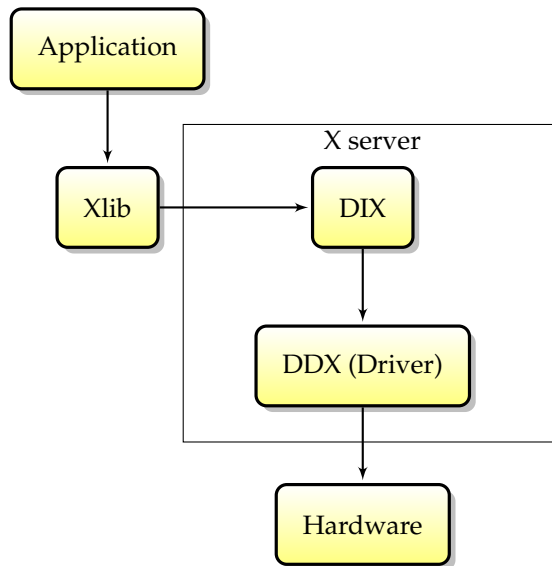


Figure 3.1: The X11 architecture.

DIX (Device-Independent X), DDX (Device-Dependent X),

modules

Xlib

3. The Big Picture

socket

X protocol

X extensions

shm -> shared memory for transport

XCB -> asynchronous

Another notable X extension is Xv, which will be discussed in further detail in the video decoding chapter.

3.2 The DRI/DRM infrastructure

Initially (when Linux first supported graphics hardware acceleration), only a single piece of code would access the graphics card directly: the XFree86 server. The design was as follows: by running with super-user privileges, the XFree86 server could access the card from user space and did not require kernel support to implement 2D acceleration. The advantage of such a design was its simplicity, and the fact that the XFree86 server could be easily ported from one operating system to another since it required no kernel component. For years this was the most widespread X server design (although there were notable exceptions, like XSun which implemented modesetting in the kernel for some drivers).

Later on, Utah-GLX, the first hardware-independent 3D accelerated design, came to Linux. Utah-GLX basically consists in an additional user space 3D driver implementing GLX, and directly accesses the graphics hardware from user space, in a way similar to the 2D driver. In a time where the 3D hardware was clearly separated from 2D (because the functionality used for 2D and 3D was completely different, or because the 3D card was a completely separate card, à la 3Dfx), it made sense to have a completely separate driver. Furthermore, direct access to the hardware from user space was the simplest approach and the shortest road to getting 3D acceleration going under Linux.

At the same time, framebuffer drivers (which will be detailed in Chapter 4) were getting increasingly widespread, and represented another component that could simultaneously access the graphics hardware directly. To avoid potential conflicts between the framebuffer and XFree86 drivers, it was decided that on VT switches the kernel would emit a signal to the X server telling it to save the graphics hardware state. Asking each driver to save its complete GPU state on VT switches made the drivers more fragile, and life became more difficult for developers who suddenly faced bug-prone interaction between different drivers. Keep in mind that there were at least two possibilities for XFree86 drivers (xf86-video-vesa and a native XFree86 driver) and two possibilities for kernel framebuffer drivers (vesafb and a native framebuffer driver), so each GPU had at least four combinations of cohabitating drivers.

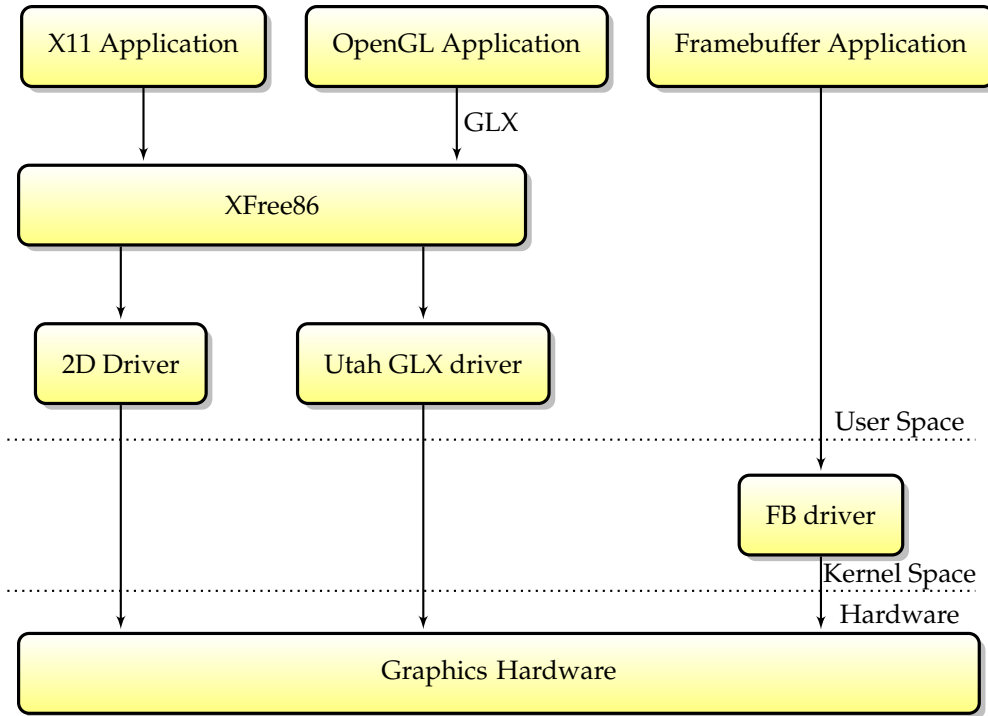


Figure 3.2: Early implementation of the Linux graphics stack using Utah-GLX.

Obviously, this model had drawbacks. First, it required that unprivileged user space applications be allowed to access the graphics hardware for 3D. Second, as can be seen on Figure 3.2 all GL acceleration had to be indirect through the X protocol, which would slow it down significantly, especially for data-intensive functions like texture uploads. Because of growing concerns about the security in Linux and performance shortcomings, another model was required.

To address the reliability and security concerns with the Utah-GLX model, the DRI model was put together; it was used in both XFree86 and its successor, X.Org. This model relies on an additional kernel component whose duty is to check the correctness of the 3D command stream, security-wise. The main change is now that instead of accessing the card directly, the unprivileged OpenGL application would submit command buffers to the kernel, which would check them for security and then pass them to the hardware for execution. The advantage of this model is that trusting user space is no longer required. Notice that although this would have been possible, the 2D command stream from XFree86 still did not go through the DRM, and therefore the X server still required super-user privileges to be able to map the GPU registers directly.

3. The Big Picture

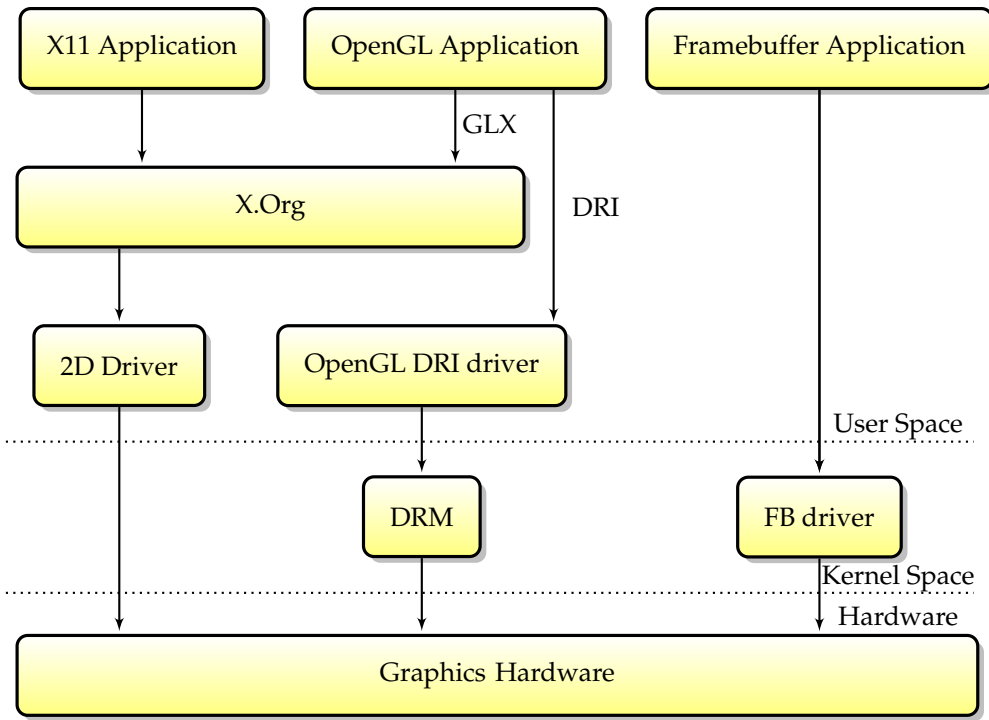


Figure 3.3: The old picture of the Linux graphics stack.

The current stack evolved from a new set of needs. First, requiring the X server to have super-user privileges always had serious security implications. Second, with the previous design different drivers were touching a single piece of hardware, which would often cause issues. In order to resolve this the key is two-fold: first, merge the kernel framebuffer functionality into the DRM module and second, have X.Org access the graphics card through the DRM module and run unprivileged. This is called Kernel Modesetting (KMS); in this model the DRM module is now responsible for providing modesetting services both as a framebuffer driver and to X.Org.

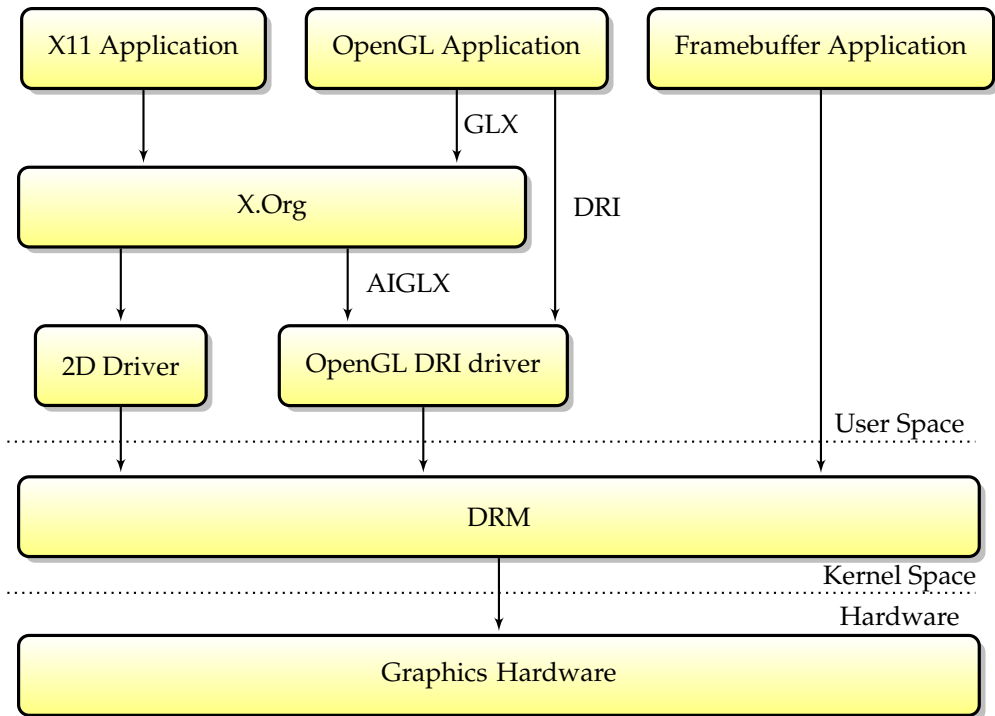


Figure 3.4: The new picture of the Linux graphics stack.

VT switches

http://dri.sourceforge.net/doc/dri_data_flow.html

http://dri.sourceforge.net/doc/dri_control_flow.html

<http://nouveau.freedesktop.org/wiki/GraphicStackOverview>

<http://people.freedesktop.org/~ajax/dri-explanation.txt>

<http://dri.sourceforge.net/doc/DRIntro.html>

<http://jonsmirl.googlepages.com/graphics.html>

<http://wiki.x.org/wiki/Development/Documentation/Glossary>

<http://mjules.littleboboy.net/carnet/index.php?post/2006/11/15/89-comment-marche-x11-xorg-et-toute-la-clique-5-partie>

3. The Big Picture

Takeaways:

- ▷ Applications communicate with X.Org through a specific library which encapsulates drawing calls.
- ▷ The current DRI design has evolved over time in a number of significant steps.
- ▷ In a modern stack, all graphics hardware activity is moderated by a kernel module, the DRM.

Framebuffer Drivers

Framebuffer drivers are the simplest form of graphics drivers under Linux. A framebuffer driver is a kernel graphics driver exposing its interface through `/dev/fb*`. This interface implements limited functionality (basically it allows setting a video mode and drawing to a linear framebuffer), and the framebuffer drivers are therefore extremely easy to create. Despite their simplicity, framebuffer drivers are still a relevant option if the only thing you are after is a basic two-dimensional display. It is also useful to know how framebuffer drivers work when implementing framebuffer acceleration for a kernel modesetting DRM driver, as the acceleration callbacks are the same. In short, framebuffer drivers are especially interesting for embedded systems, where memory footprint is essential, or when the intended applications do not require advanced graphics acceleration.

At the core, a framebuffer driver implements the following functionality:

- ▷ Modesetting. Modesetting consists in configuring video mode to get a picture on the screen. This includes choosing the video resolution and refresh rates.
- ▷ Optional 2d acceleration. Framebuffer drivers can provide basic 2D acceleration used to accelerate the linux console. These operations include copies in video memory and solid fills. Acceleration is sometimes made available to user space through a hook (the user space must then program card specific MMIO registers, and this requires root privileges).

By implementing only these two pieces, framebuffer drivers remain the simplest and most amenable form of linux graphics drivers. Framebuffer drivers do not always rely on a specific card model (like nvidia or ATI). Drivers on top of vesa, EFI or Open-firmware exist; instead of accessing the graphics hardware directly, these drivers call firmware functions to achieve modesetting and 2D acceleration.

<http://www.linux-fbdev.org/HOWTO/index.html>

4.1 Creating a framebuffer driver

struct platform_driver with a probe function

probe function in charge of creating the fb_info struct and register_framebuffer() on it.

4.2 Framebuffer operations

The framebuffer operations structure is how non-modesetting framebuffer callbacks are set. Different callbacks can be set depending on what functionality you wish to implement, like fills, copies, or cursor handling. By filling struct fb_ops callbacks, one can implement the following functions:

Set color register

```
int fb_setcolreg(unsigned regno, unsigned red, unsigned green, unsigned blue, unsigned transp, struct fb_info *info);
```

Set color registers in batch

```
int fb_setcmap(struct fb_cmap *cmap, struct fb_info *info);
```

Blank display

```
int fb_blank(int blank, struct fb_info *info);
```

Pan display

```
int fb_pan_display(struct fb_var_screeninfo *var, struct fb_info *info);
```

Draws a solid rectangle

```
void fb_fillrect (struct fb_info *info, const struct fb_fillrect *rect);
```

Copy data from area to another

```
void fb_copyarea(struct fb_info *info, const struct fb_copyarea *region);
```

Draws an image to the display

```
void fb_imageblit(struct fb_info *info, const struct fb_image *image);
```


Draws cursor

```
int fb_cursor(struct fb_info *info, struct fb_cursor *cursor);
```

Rotates the display

```
void fb_rotate(struct fb_info *info, int angle);
```

Wait for blit idle, optional

```
int fb_sync(struct fb_info *info);
```

Note that common framebuffer functions (cfb) are available if you do not want to implement everything for your device specifically. These functions are `cfb_fillrect`, `cfb_copyarea` and `cfb_imageblit` and will perform the corresponding function in a generic, unoptimized fashion using the CPU.

Takeaways:

- ▷ Framebuffer drivers are the simplest form of linux graphics driver, requiring little implementation work.
- ▷ Framebuffer drivers deliver a low memory footprint and thus are useful for embedded devices.
- ▷ Implementing acceleration is optional as software fallback functions exist.

The Direct Rendering Manager

The use of a kernel module is a requirement in a complex world. This kernel module is called the Direct Rendering Manager (DRM, not to be confused with Digital Rights Management) and serves multiple purposes:

- ▷ Put critical initialization of the card in the kernel, for example uploading firmwares or setting up DMA areas.
- ▷ Share the rendering hardware between multiple user space components, and arbitrate access.
- ▷ Enforce security by preventing applications from performing DMA to arbitrary memory regions, and more generally from programming the card in any way that could result in a security hole.
- ▷ Manage the memory of the card, by providing video memory allocation functionality to user space.
- ▷ More recently, the DRM was improved to achieve modesetting. This simplifies the previous situation where both the DRM and the framebuffer driver were fighting to access the same GPU. Instead, the framebuffer driver is removed and instead framebuffer support is implemented in the DRM.

Kernel module (DRM)

Global DRI/DRM user space/kernel scheme (figure with libdrm - drm - entry points - multiple user space apps)

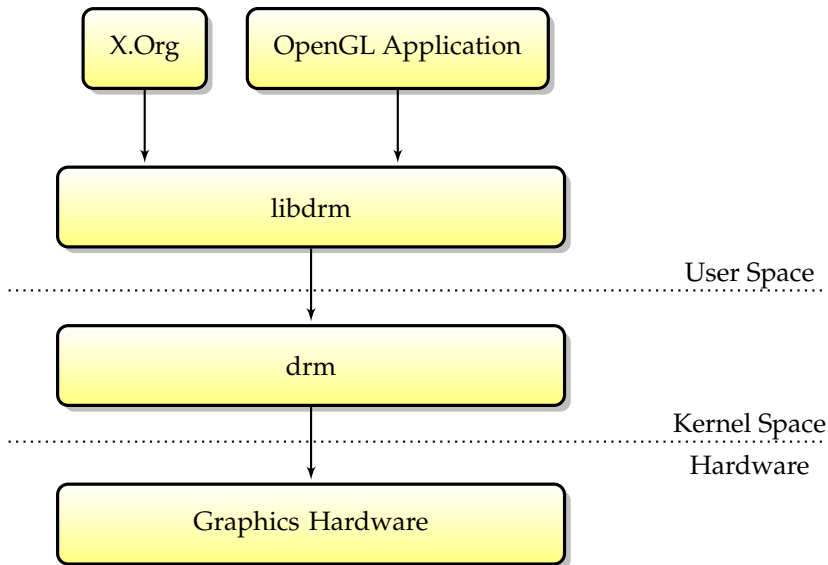


Figure 5.1: Accessing the DRM through libdrm.

When designing a Linux graphics driver aiming for more than simple framebuffer support, a DRM component is the first thing to do. One should derive a design that is both efficient and enforces security. The DRI/DRM scheme can be implemented in different ways and the interface is indeed entirely card-specific. Do not always follow the existing models that other drivers use, innovate!

5.1 DRM batch buffer submission model

At the core of the DRM design is the `DRM_GEM_EXECBUFFER` ioctl; which lets a user space application submit a batch buffer to the kernel, which in turns puts it on the GPU. This ioctl allows many things like sharing the hardware, managing memory and enforcing memory protection.

5.1.1 Hardware sharing

One of the duties of the DRM is to multiplex the GPU itself between multiple user space processes. Given that the GPU holds graphics state, a problem arises when multiple applications use the same GPU: if nothing is done, the applications can stomp over each other's state. Depending on the hardware at hand, there are two major cases:

- ▷ When the GPU features hardware state tracking, the hardware sharing is simpler since each application can send to a separate context, and the GPU tracks each application's state itself. This is the way the nouveau driver works.

- ▷ When the GPU doesn't have multiple hardware contexts, the common way of multiplexing the hardware is to reemit the state at the beginning of each batch buffer; it's the way the intel and radeon drivers multiplex the GPU. Note that this duty of reemitting the state relies on user space entirely. If the user space doesn't reemit the state at the beginning of each batch buffer, the state from other DRM processes will leak onto it.

The DRM also prevent simultaneous access to the same hardware.

5.1.2 Memory management and security

The kernel has the ability to move memory areas around to handle high memory pressure situations. Depending on the hardware, there are two ways to achieve it:

- ▷ If the hardware has complete memory protection and virtualization, then it is possible to page in memory resources into the GPU as they get allocated and isolate the per-process. Therefore not much is required to support memory protection of GPU memory.
- ▷ When the hardware doesn't have memory protection, this can still be achieved entirely in the kernel, in a way where the user space is completely oblivious to it. In order to allow relocations to work for a user space process which is otherwise unaware of them, the command submission ioctl will rewrite the command buffers in the kernel by replacing all the hardware offsets to their current locations. This is possible since the kernel knows about the current position of all memory buffers.

To prevent access to arbitrary GPU memory, the command submission ioctl can also check that each of these offsets is owned by the calling process, and reject the batch buffer if it isn't. That way it is possible to implement memory protection when the hardware doesn't provide that functionality.

GEM, TTM

Share video memory

5.2 Modesetting

Modesetting is the act of setting a mode on the card to display. This can range from extremely simple procedures (calling a VGA interrupt or VESA call is a basic form of modesetting) to directly programming the card registers (which brings along the advantage of not needing to rely on a VGA or VESA layer). Historically, this was achieved in user space by the DDX.

However, these days it makes more sense to put it in the kernel once and for all, and share it between different GPU users (framebuffer drivers, DDXes, EGL stacks...). This extension to modesetting is called kernel modesetting (also known as KMS). A number of concepts are used by the modesetting interface (those concepts are mainly inherited from the Randr 1.2 specification).

5. The Direct Rendering Manager

Crtc

Crtc is in charge of reading the framebuffer memory and routes the data to an encoder

Encoder

Encoder encodes the pixel data for a connector

Connector

The connector is the name physical output on the card (DVI, Dsub, Svideo...). Notice that connectors can get their data from multiple encoders (for example DVI-I which can feed both analog and digital signals)

Also, on embedded or old hardware, it is common to have encoders and connectors merged for simplicity/power efficiency reasons.

+++ Ajouter ici un schema crtc-encoder-connector

5.3 libdrm

libdrm is a small (but growing) component that interfaces between user space and the DRM module, and allows calling into the entry points.

Obviously security should not rely on components from libdrm because it is an unprivileged user space component

Takeaways:

- ▷ The DRM manages all graphics activity in a modern linux graphics stack.
- ▷ It is the only trusted piece of the stack and is responsible for security. Therefore it shall not trust the other components.
- ▷ It provides basic graphics functionality: modesetting, framebuffer driver, memory management.

X.Org Drivers

This chapter covers the implementation of a 2D driver inside X.Org. There are multiple ways to implement a 2D X.Org driver: ShadowFB, XAA, EXA. Another simple way of implementing X.Org support is through the `xf86-video-fbdev` module. This module implements X.Org on top of an existing, in-kernel framebuffer driver. It can be a “good enough” option if all you need is basic X compatibility.

<http://www.x.org/wiki/DriverDevelopment>

6.1 Creating a basic driver

Mandatory entry points

PreInit

This function is in charge of the initialization.

```
Bool PreInit (ScrnInfoPtr pScreen, int flags);
```

ScreenInit

This function gets called on startup. It is responsible for setting up all per-screen state.

```
Bool ScreenInit(int scrnIndex, ScreenPtr screen, int argc, char **argv);
```

EnterVT

This is called when VT switching back to the X server. In a KMS-enabled X driver, this will only need to acquire the DRM master bit and set the video mode.

```
Bool EnterVT(int scrnIndex, int flags);
```

6. X.Org Drivers

LeaveVT

This is called when switching away from the X server to another VT. In a KMS-enabled X driver, this only needs to drop the DRM master bit.

```
/Bool LeaveVT(int scrnIndex, int flags);
```

Optional functions (but very useful)

SwitchMode

Sets a video mode.

```
Bool SwitchMode(int scrnIndex, DisplayModePtr mode, int flags);
```

AdjustFrame

This function is used to initialize the Start Address - the first displayed location in the video memory (randr 1.3 panning)

```
void AdjustFrame(int scrnIndex, int x, int y, int flags);
```

FreeScreen

Cleanup the ScreenInit

```
void FreeScreen(int scrnIndex, int flags);
```

6.2 ShadowFB acceleration

ShadowFB provides no acceleration proper, a copy of the framebuffer is kept in system memory. The driver implements a single hook that copies graphics from system to video memory. This can be implemented using either a DMA copy, or a CPU copy (depending on the hardware and copy size, either can be better).

Despite the name, shadowFB is not to be confused with the kernel framebuffer drivers.

Although ShadowFB is a very basic design, it can result in a more efficient and responsive desktop than an incomplete implementation of EXA.

To implement shadowFB acceleration, a driver simply calls `Bool ShadowFBInit(ScreenPtr pScreen, RefreshAreaFuncPtr refreshArea)`. `refreshArea` is a function pointer with the following profile:

```
void RefreshAreaFuncPtr(ScrnInfoPtr pScreen, int numBoxes, BoxPtr pBox);
```

The callback should refresh `numBoxes` contained in the `pBox[]` array. It can be achieved either with a CPU copy to video memory or with a DMA on the GPU.

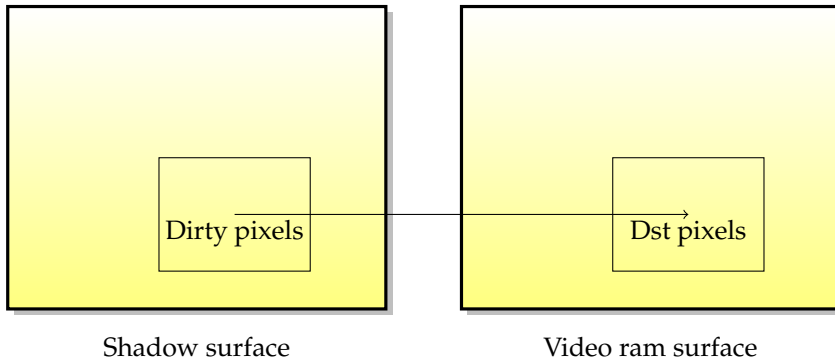


Figure 6.1: Shadowfb acceleration.

6.3 XAA acceleration

Scanline based acceleration

Offscreen area, same pitch as the screen

6.4 EXA acceleration

EXA is an interface inside X.Org implemented by drivers for 2D acceleration. It was originally designed as KAA in the Kdriver X server, and then was adapted into X.Org. The interface used is pretty simple; for each acceleration function three hooks are available: `PrepareAction`, `Action` and `FinishAction`. `PrepareAction` is called once before the operation is used. `Action` can be called many times in a row after a single `PrepareAction` call for different surfaces. `FinishAction` is called after all the `Action` calls have been made. The number of `Action` calls can be just one or many, but the `PrepareAction` and `FinishAction` function will always be called once, first and last. The `PrepareAction` functions return a boolean, and can return false if they fail at accelerating the specific type of operation, in which case a software fallback is used instead. Otherwise the function returns true and subsequent `Action` calls are expected to succeed.

EXA is implemented in the driver as a series of callbacks; the following gives a detail of the EXA acceleration functions that a driver should implement; some of them like `Composite()` are optional.

Solid

Solid just fills an area with a solid color (RGBA). Just like with all EXA callbacks, there are three main functions: the first one is the `Prepare()` function which sets the graphics

6. X.Org Drivers

state required to do solid fills. The second one is the `Solid()` function which actually does the solid fill. The last one is `Done()` which signals to the driver that the current series of `Solid()` calls is over, so that it can restore graphics state and/or flush required GPU states.

Note that `Solid()` can be called many times in a row between a `Prepare()` and `Done()` (the same applies to other EXA callbacks).

```
Bool PrepareSolid(PixmapPtr pPixmap, int alu, Pixel planemask, Pixel fg);
```

```
void Solid(PixmapPtr pPixmap, int x1, int y1, int x2, int y2);
```

```
void DoneSolid(PixmapPtr pPixmap);
```

Copy

`Copy` is a simple blit function; it copies a rectangle area in video memory from one pixmap to another pixmap (possibly the same one). Just like with `Solid()` it has a `Prepare/Copy/Done` triplet of callbacks, and `Copy()` can be called many times in a row.

```
Bool PrepareCopy(PixmapPtr pSrcPixmap, PixmapPtr pDstPixmap, int dx, int dy, int alu, Pixel planemask);
```

```
void Copy(PixmapPtr pDstPixmap, int srcX, int srcY, int dstX, int dstY, int width, int height);
```

```
void DoneCopy(PixmapPtr pDstPixmap);
```

Composite

`Composite` is an optional interface used to achieve composite operations like blending. This allows accelerating 2D desktop effects like blending, scaling, operations with masks... All in all, the `composite()` hook is sufficient to implement a basic 2D composite window manager (KDE and XFCE are examples of window compositors indirectly using EXA through the Render API to implement compositing).

If the driver doesn't support the required operation, it is free to return `false`, in which case the EXA layer will call into the pixman library as a software fallback. Of course this will be done on the CPU as a fallback.

UploadToScreen

`UploadToScreen` copies an area from system memory to video memory

```
Bool UploadToScreen(PixmapPtr pDst, int x, int y, int w, int h, char *src, int src_pitch);
```

DownloadFromScreen

DownloadFromScreen copies an area from video memory to system memory

```
Bool DownloadFromScreen(PixmapPtr pSrc, int x, int y, int w, int h, char *dst, int dst_pitch);
```

PrepareAccess

PrepareAccess makes the pixmap accessible from the CPU. This includes mapping it into memory, copying it from unmappable video memory, untiling the pixmap... What this exactly does is very dependent from the GPU, but the core of the matter is that you must provide a piece of CPU-accessible memory which is stored in a linear form. This can be achieved by either mapping GPU memory into the CPU domain with a linear view, or by doing a copy from GPU to CPU memory.

FinishAccess

FinishAccess is called once the pixmap is done being accessed, and must undo what PrepareAccess did to make the pixmap usable by the GPU again.

A note about EXA performance

EXA Pixmap migration. EXA tries to be smart about pixmap migration, and will only migrate the parts of a pixmap that are required for an operation. Migration heuristics Greedy/Mixed/Driver. Since fallbacks might require pixmap migration, it is not always better to implement some of the composite interface. For example if the usage pattern often calls operations A->B->C and only B is GPU accelerated, A will be done on the CPU, B will trigger a migration to GPU memory and will do the operation on the GPU, and C will trigger a migration back to system memory and do the third operation on the CPU. Since the overhead and cost of pixmap migration is so high, the end result is probably slower than doing all three operations on the CPU, and therefore the newly added composite() functionality actually results in a slowdown!

As a side effect, it is often better to profile before implementing specific EXA composite() functions, and look at the common calling patterns; a very common example is antialiased fonts (which will also show different calling patterns if subpixel rendering is enabled or not).

Takeaways:

- ▷ Multiple choices exist for accelerating 2D in X.Org.
- ▷ The most efficient one is EXA, which puts all the smart optimizations in a common piece of code, and leaves the driver implementation very simple.
- ▷ Today, most 2D acceleration is implemented using the 3D engine of the graphics card.
- ▷ If your card cannot accelerate 2D operations, shadowfb is the path to take.

Video Decoding

7.1 Video Standards

H262 (mpeg 2, DVD)

H263 (divx/mpeg4)

H264 (used on blu-ray)

7.2 Video decoding pipeline

Two typical video pipelines : mpeg2 and h264

The H262 decoding pipeline

iDCT -> MC -> CSC -> Final display

The H.264 decoding pipeline

entropy decoding -> iDCT -> MC -> CSC -> Final display

7.2.1 Entropy

Entropy encoding is a lossless compression phase. It is the last stage of encoding and therefore also the first stage of decoding.

CABAC/CAVLC

7. Video Decoding

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.13983 \\ 1 & -0.39465 & -0.58060 \\ 1 & 2.03211 & 0 \end{bmatrix} \begin{bmatrix} Y \\ U \\ V \end{bmatrix}$$

Figure 7.1: YUV to RGB Conversion formula as per ITU-R RB recommendation 601.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.13983 \\ 1 & -0.39465 & -0.58060 \\ 1 & 2.03211 & 0 \end{bmatrix} \begin{bmatrix} Y \\ U \\ V \end{bmatrix}$$

Figure 7.2: YUV to RGB Conversion formula as per ITU-R RB recommendation 709.

7.2.2 Inverse DCT

7.2.3 Motion Compensation

7.2.4 Color Space Conversion

A color space is the way color is represented.

In order to save space, perception principles are applied to movie encoding. The color space conversion stage exploits the fact that the human eye is more perceptive of the luminance than the chrominance, and separates the color data into those two components. Intuitively, the luminance is the amount of light in the color, and the chrominance is the name of the color (red, green, yellow...) and its amount of saturation.

The quality of the chrominance data can be lowered to conserve space, for example by downsampling it, and the missing chrominance data is interpolated at playing time while minimally impacting the perceived video quality.

This is how the YUV color space works: it separates the color information into one luminance component (Y) and two chrominance components (U and V). The chrominance information is less relevant to the eye than the luminance, so usually chrominance is subsampled and luminance is at the original resolution. Therefore, the Y plane usually has a higher resolution than the U and V planes.

Bandwidth gain example (RGBA32 vs YV12)

The color conversion from the YUV to the RGB color space is a linear relation, and can therefore be represented by a matrix/vector multiplication, the matrix being the conversion formula.

Figure 7.1 shows the conversion matrices from ITU-R BT Recommendation 601 (standard content) and recommendation 709 (intended for HD content). Notice that although these matrices are very similar, there are numerical differences which will result in slight off-colored rendering if one is used in place of the other. This is indeed often the case that video decoders with YUV to RGB hardware are used to playback high definition content but no attention is made to the proper conversion matrix that should be used. Since the colors are only slightly wrong, this problem is commonly

Format name	Y:U:V bits per pixel	Layout	Comments
YV12	8:2:2	1 Y plane, 1 V 2*2 sub-sampled plane, 1 U 2*2 sampled plane	Same as I420 except U and V are reversed.
I420	8:2:2	1 Y plane, 1 U 2*2 sub-sampled plane, 1 V 2*2 sub-sampled plane	Same as YV12 except U and V are reversed.
NV12	8:2:2	1 Y plane, 1 packed U+V 2*2 sub-sampled plane	Convenient for hardware implementation on 3D-capable GPUs
YUY2 (YUYV)	8:4:4	1 Packed YUV plane	Packed as Y0U0Y1V0

Table 7.1: Common YUV color space formats

overlooked, whereas most hardware features at least a BT601/BT709 switch, or a fully programmable conversion matrix.

YUV Planar and packed (interlaced) formats on Figure 7.3.

Plane order (YV12 vs NV12)

Order of the planes (YV12, I420)

<http://en.wikipedia.org/wiki/YUV>

<http://www.fourcc.org/yuv.php>

<http://www.glennchan.info/articles/articles.html>

http://www.poynton.com/papers/SMPTE_98_YYZ_Luma/index.html

The final stage of video decoding is video upscaling. Video upscaling consists in upscaling the movie resolution to the screen resolution. It can be done by specialized hardware, the 3D engine, or the 2D blitter if it has scaling capabilities. Since the conversion from YUV space to RGB space is linear, filtered upscaling can be done either in the YUV or RGB space, which conveniently allows using bilinear texture filtering which is available on 3D hardware to sample the YUV data. This allows a single pass color space conversion and scaling. For example, bi-linear filtering will work just fine with three textures for the three Y, U and V planes. Notice that higher quality can be obtained at the expense of performance by using better filtering modes, such as bi-cubic [citer papier hadwiger], even though this can prove to be costly. A trade-off can be achieved by implementing bi-cubic filtering for the (most eye-visible) Y plane, and keeping bi-linear filtering for U and V planes.

7. Video Decoding

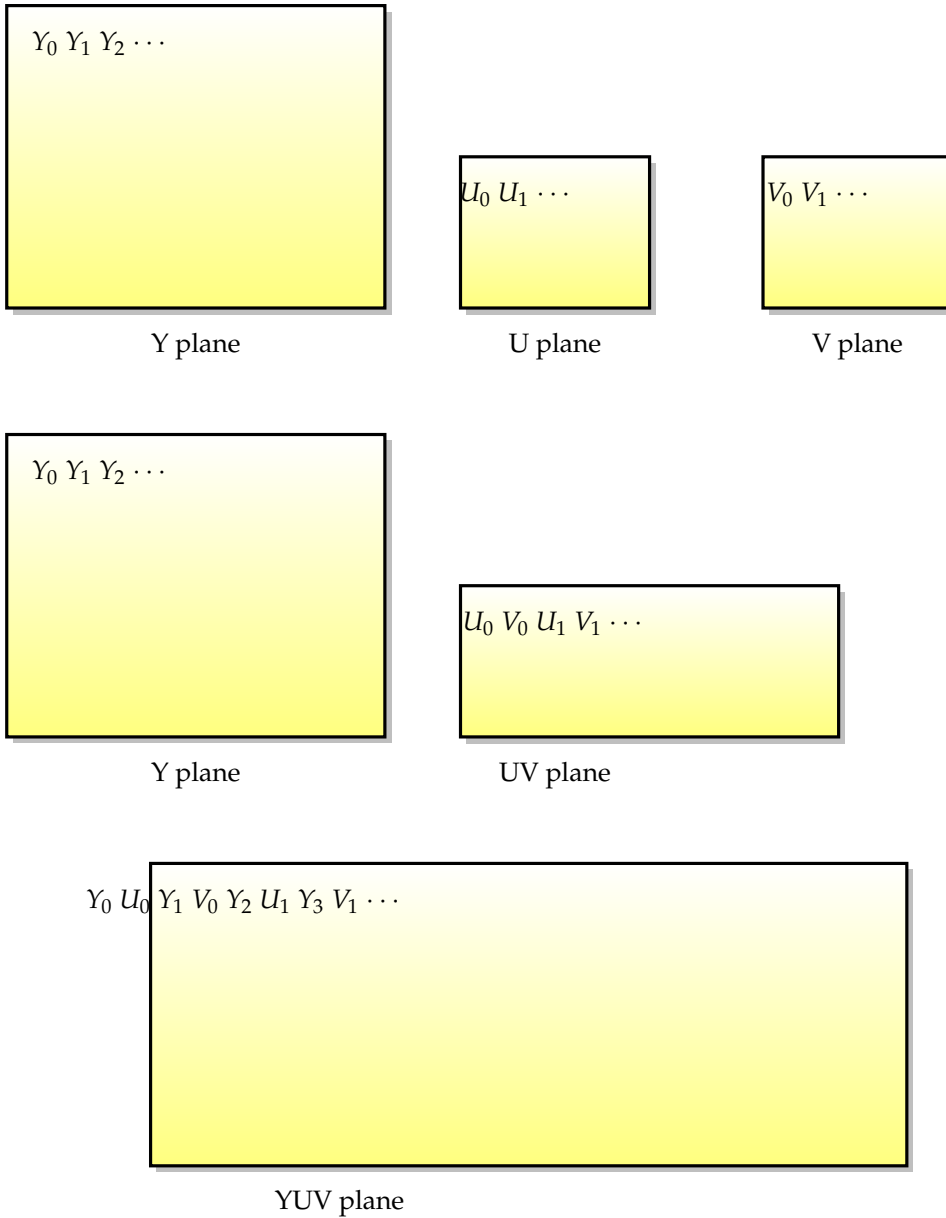


Figure 7.3: YUV layouts in memory: planar format example (YV12, top), partially interleaved format example (NV12, middle), fully interleaved format example (YUY2, bottom).

If the hardware cannot achieve color space conversion and scaling at the same time (for example if you have a YUV->RGB blitter and a shader less 3D engine), it is equivalent to first do color space conversion from YUV to RGB and then scaling in RGB space. Again this is only possible when the color conversion operation is a linear operation, in particular that means gamma correction has to be ignored.

gamma conversion

using a conversion shader or a conversion texture lookup

7.3 Video decoding APIs

Xv

Xv is simply about CSC and scaling. In order to implement Xv, a typical X.Org driver will have to implement this space conversion. Although the Xv API is a little complex for what it implements, the guts of it consists in the PutImage function, which puts an YUV image on screen. Multiple YUV formats can be handled, planar or interlaced mainly. Note that Xv has RGB support as well. Thanks to the bandwidth gains and DMA transfers, even an Xv implementation already provides a relevant level of video decoding acceleration, and can prove sufficient depending on the target hardware (for example, it can prove to be fine when coupled with a powerful CPU to decode H264 content).

XvMC

idct + mc +csc

VAAPI

VAAPI was initially created for intel's poulsbo video decoding. The API is very tailored to embedded platforms and has many entry points, at different pipeline stages, which makes it more complex to implement.

VDPAU

The VDPAU was initiated by nvidia for H264 & VC1 decoding support

XvBA

All 3 APIs are intended for full

7. Video Decoding

OpenMax

<http://x264dev.multimedia.cx>

Takeaways:

- ▷ A video decoding pipeline consists in multiple stages chained together.
- ▷ Color space conversion and scaling is the most important stage, and if your driver implements only one operation for simplicity, this is it.
- ▷ Implementing a full pipeline can provide a high performance boost, and save battery life on mobile systems.

OpenGL

OpenGL is a specification. There are many OpenGL implementations, both hardware accelerated and in software. As a driver author, our job is sometimes to provide a hardware-accelerated OpenGL implementation. In this section we describe the OpenGL pipeline from the point of view of the driver.

OpenGL ARB, khronos, bla bla...

8.1 The OpenGL Rendering Pipeline

8.1.1 Vertex processing

vertex stage
vertex buffers

8.1.2 Geometry processing

8.1.3 Fragment processing

Rasterization
Render buffers
Textures

Takeaways:

- ▷ OpenGL is a suite of stages arranged in a pipeline.

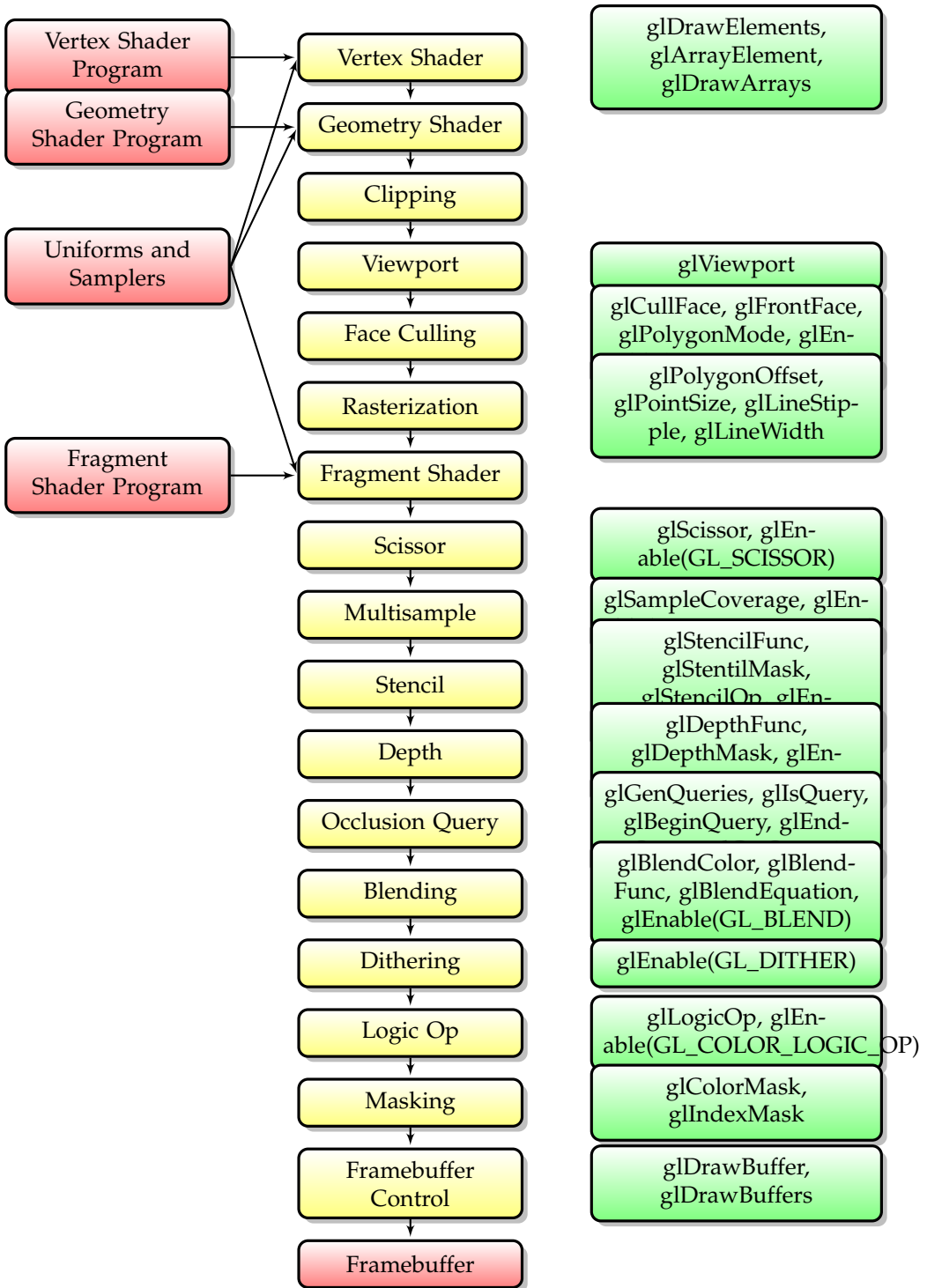


Figure 8.1: The OpenGL 3.2 pipeline.

Mesa

Mesa is both a software OpenGL implementation, and the common rendering architecture for all open source hardware accelerated graphics drivers. We now describe the internals of Mesa and the available interfaces and infrastructure required for graphics drivers.

9.1 Mesa

Mesa serves two major purposes:

- ▷ Mesa is a software implementation of OpenGL. It is considered to be the reference implementation and is useful in checking conformance, seeing that the official OpenGL conformance tests are not publicly available.
- ▷ Mesa provides the OpenGL entry points for Open Source graphics drivers under linux.

In this section, we will focus on the second point.

9.2 Mesa internals

9.2.1 Textures in mesa

Takeaways:

- ▷ Mesa is the reference OpenGL implementation under Linux.
- ▷ All Open Source graphics drivers use Mesa for 3D

Chapter 10

Gallium 3D

Gallium 3D is the Future of 3D Acceleration.

<http://jrfonseca.blogspot.com/2008/04/gallium3d-introduction.html>

<http://people.freedesktop.org/~csimpson/gallium-docs/>

10.1 Gallium3D: a plan for a new generation of hardware

Ten years ago, GPUs were a direct match with all the OpenGL or Direct3D functionality; back then the GPUs had specific transistors dedicated to each piece of functionality. With the explosion in the amount of 3D functionality, this quickly made it impractical both for application developers (who saw the 3D APIs growing huge) and hardware designers (who faced an explosion of the number of specific functionality a GPU needed), and shaders were created. Instead of providing specific functionality, the 3D APIs would now let the programmers create these little programs and run them on the GPU. As the hardware was now programmable in a way which was a superset of fixed functionality, the fixed function pipelines were not required any more and were removed from the cards. Gallium 3D is modeled around the simple observation that today's GPUs do not have fixed pipe any more and only feature shaders, but drivers still have to "emulate" fixed function on top of the shaders to provide API compatibility. Doing so in every driver would require a lot of code duplication, and the Gallium model is to put this code in a common place. Therefore gallium drivers become smaller and easier to write and to maintain.

everything is a shader, including inside the driver

thin layer for fixed pipe -> programmable functionality translation

global diagram

10.2 State trackers

A state tracker implements an API (for example OpenGL, OpenVG, Direct3D...) by turning it into API-agnostic and hardware-agnostic TGSI calls.

10.3 Pipe driver

A pipe driver is the main part of a hardware-specific driver.

10.4 Winsys

The winsys is in charge of talking to the OS/Platform of choice. The pipe driver relies on the Winsys to talk to the hardware. For example, this allows having a single pipe driver with multiple winsyses targetting different Operating systems.

10.5 Vertex submission

The VBO interface

10.6 Writing Gallium3D drivers

screen

context

pipe_transfer

10.7 Shaders in Gallium

In order to operate shaders, Gallium features an internal shader description language which uses 4-component vectors. We will later refer to the 4 components of a vector as x,y,z,w . In particular, $v.x$ is the first component of vector v , $v.xyzw$ are all 4 components of v in that order, and swizzling is allowed, for example $v.wzyx$ reverses the component order. It is also legal to replicate a component, for example $v.xxxx$ means four times the x component of v and $v.yyzz$ means two times y and two times z .

These components usually carry no semantics, and despite their name they can very well carry a color or an opacity value indifferently.

TGSI instruction set

`mesa/src/gallium/auxiliary/tgsi/tgsi-instruction-set.txt`

Takeaways:

- ▷ Gallium 3D is the new graphics API.
- ▷ Everything is converted to a shader internally, fixed functionality is gone.
- ▷ Drivers are simpler than classic Mesa drivers, as one only has to implement shaders to get all fixed functionality to work.

Chapter **11**

GPU Computing

Chapter 12

Suspend and Resume

VT switches

Card state

Suspend/resume hooks in the DRM

Takeaways:

- ▷ Suspend and resume has long been very clumsy, but this is solved now thanks to the DRM implementing more functionality.

Chapter 13

Technical Specifications

Technical specifications are the nuts and bolts of graphics driver work. Without hardware specifications, no work can be started. However, manufacturing companies are usually wary of sharing said specifications, as they think this will hinder their business. While this claim is false (because you can't copy a GPU from just its specifications), it is still very widespread and prevents a lot of hardware from being properly documented. Therefore, getting hold of hardware specifications will be the first major step in any graphics driver development project.

13.1 Obtaining official specifications

Public specifications Some vendors distribute the technical documentation for their hardware publicly without restrictions.

Sometimes, things can be as simple as asking the vendor, who might share the documentation (possibly under NDA, see below).

NDA (Non-Disclosure Agreement) Put simply, an NDA is a contract signed between the developer and the hardware company, by which the developer agrees not to spread the docs he received. However, there can be more restrictions in an NDA.

Terms of the NDA

Before signing an NDA, think. Whatever lawyers say, there is no such thing as a "standard" NDA, you can always negotiate.

Can Open Source drivers be written from that documentation under that NDA?

What happens when the NDA expires? Can code still be free, are you bound by any clause?

What about yourself? Are you prevented from doing further work on this hardware?

13.2 Reverse engineering

When specifications are not easily available or just incomplete, an alternate route is reverse engineering. Reverse engineering consists in figuring out the specifications for a given piece of hardware by yourself, for example by looking at what a black-box binary driver does to the hardware under certain circumstances.

Reverse engineering is not just a tool to obtain missing hardware specifications, it is also a strong means of Open Source advocacy. Once a reverse engineered driver exists and ships in linux distributions, pressure shifts on the hardware vendor for support. This, in turn, can force the vendor to support Open Source drivers.

not as difficult as it seems, requires organization, being rigorous. Write down all bits of information (even incomplete bits), share it among developers, try to work out bits one by one. Do not hesitate writing ad-hoc tools, as they will save precious time down the road (if you hesitate, you have crossed the line already!).

Mmiotrace

The basic idea behind mmio-trace is simple: it first hooks the ioremap call, and therefore prevents mapping of a designated I/O area. Subsequently, accesses to this area will generate page faults, which are caught by the kernel. For each page fault, the faulting instruction is decoded to figure out the write or read address, along with the value written/read. The page is put back, the faulting instruction is then single-stepped, and the page is then removed again. Execution then continues as usual.

mmio trace is now part of the official Linux kernels. Therefore, any pre-existing driver can be traced.

Libsefault

libsefault is similar to mmio-trace in the way it works: after removing some pages which one want to track accesses to, it will generate a segmentation fault on each access and therefore be able to report each access. The difference is that libsefault is a user space tool while mmio-trace is a kernel tool.

Valgrind-mmt

Valgrind is a dynamic recompiling and instrumentation framework. Valgrind-mmt is a plugin for valgrind which implements tracing of read and writes to a certain range of memory addresses, usually an mmio range accessed from user space. Memory accesses are dynamically instrumented thanks to valgrind and each access to the zones we want to see traced is logged.

vbetool/vbtracetool

Virtualization

Finally, one last pre-existing tool to help reverse engineering is virtualization. By running a proprietary driver in a controlled environment, one can figure out the inner workings of a GPU. The plan is then to write an emulated GPU while doing the reverse engineering (which imposes the use of an open source virtualization solution like Qemu).

Ad-hoc tools

In addition to these generic tools, you will often find it useful to implement your own additional tools, tailored for specific needs. Renouveau is an example of such a tool that integrates the reverse engineering mechanisms, the command decoding and printing. In order to achieve decoding of the commands, it carries a database of the graphics commands of nvidia GPUs. This allows quick testing of new database entries. Headers generated from this database are later used in the driver development process.

Takeaways:

- ▷ Technical specifications of course very important for authoring graphics drivers.
- ▷ NDAs can have unforeseen implications on yourself and your work.
- ▷ When they are unavailable, incomplete or just plain wrong, reverse engineering can help you figure out how the hardware actually works.

Chapter 14

Beyond Development

14.1 Testing for conformance

Rendercheck

OpenGL conformance test suite

The official OpenGL testing suite is not publicly available, and (paying) Khronos Membership is required. Instead, most developers use alternate sources for test programs.

Piglit

glean

glean.sourceforge.net

Mesa demos

[mesa/progs/*](#)

14.2 Debugging

gdb and X.Org

gdb needs to run on a terminal emulator while the application debug might be with a lock held. That might result in a deadlock between the application stuck with a lock and gdb waiting to be able to output text.

printk debug

crash

(surcouche gdb pour analyser les vmcore)

kgdb

serial console

diskdump

linux-uml

systemtap

14.3 Upstreaming

Submitting your code for inclusion in the official trees is an important part of the graphics driver development process under linux. There are multiple motivations for doing this.

First, this allows end users to get hold of your driver more easily.

Second, this makes it easier for your driver maintenance in the future: in the event of interface changes,

Why upstream?

How?

When?

Takeaways:

- ▷ Thoroughly testing all your changes can save you the cost of bisection later on.
- ▷ Debugging is not easy for graphics drivers.
- ▷ By upstreaming your code in official repositories, you save yourself the burden of adapting it to ever-moving programming interfaces in X.Org, Mesa and the kernel.

Chapter 15

Conclusions