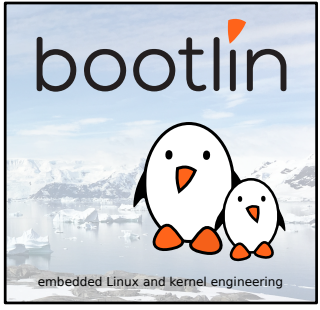
Understanding the Linux Graphics Stack training



Latest update: October 26, 2024.

Document updates and training details:
https://bootlin.com/training/graphics

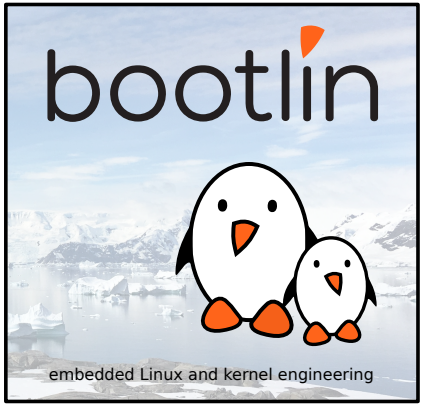
Corrections, suggestions, contributions and translations are welcome!
Send them to feedback@bootlin.com

Understanding the Linux Graphics Stack training

- These slides are the training materials for Bootlin's *Understanding the Linux Graphics Stack* training course.
- If you are interested in following this course with an experienced Bootlin trainer, we offer:
  - **Public online sessions**, opened to individual registration. Dates announced on our site, registration directly online.
  - **Dedicated online sessions**, organized for a team of engineers from the same company at a date/time chosen by our customer.
  - **Dedicated on-site sessions**, organized for a team of engineers from the same company, we send a Bootlin trainer on-site to deliver the training.
- Details and registrations: https://bootlin.com/training/graphics
- Contact: training@bootlin.com

Icon by Eucalyp, Flaticon

About Bootlin



Bootlin introduction

- Engineering company
  - In business since 2004
  - Before 2018: *Free Electrons*
- Team based in France and Italy
- Serving **customers worldwide**
- **Highly focused and recognized expertise**
  - Embedded Linux
  - Linux kernel
  - Embedded Linux build systems
- **Strong open-source** contributor
- Activities
  - **Engineering** services
  - **Training** courses
- https://bootlin.com

bootlin

Bootlin engineering services

Bootloader / firmware development

U-Boot, Barebox, OP-TEE, TF-A, .../

Linux kernel porting and driver development

Linux BSP development, maintenance and upgrade

Embedded Linux build systems

Yocto, OpenEmbedded, Buildroot, ...

Embedded Linux integration

Boot time, real-time, security, multimedia, networking

Open-source upstreaming

Get code integrated in upstream Linux, U-Boot, Yocto, Buildroot, ...

Bootlin training courses

Embedded Linux system development On-site: 4 or 5 days Online: 7 * 4 hours	Linux kernel driver development On-site: 5 days Online: 7 * 4 hours	Yocto Project system development On-site: 3 days Online: 4 * 4 hours	Buildroot system development On-site: 3 days Online: 5 * 4 hours
Understanding the Linux graphics stack On-site: 2 days Online: 4 * 4 hours	Embedded Linux audio On-site: 2 days Online: 4 * 4 hours	Real-Time Linux with PREEMPT_RT On-site: 2 days Online: 3 * 4 hours	Linux debugging, tracing, profiling and performance analysis On-site: 3 days Online: 4 * 4 hours

Bootlin, an open-source contributor

- Strong contributor to the **Linux** kernel
  - In the top 30 of companies contributing to Linux worldwide
  - Contributions in most areas related to hardware support
  - Several engineers maintainers of subsystems/platforms
  - 9000 patches contributed
  - https://bootlin.com/community/contributions/kernel-contributions/
- Contributor to **Yocto Project**
  - Maintainer of the official documentation
  - Core participant to the QA effort
- Contributor to **Buildroot**
  - Co-maintainer
  - 6000 patches contributed
- Significant contributions to U-Boot, OP-TEE, Barebox, etc.
- Fully **open-source training materials**

Bootlin on-line resources

- Website with a technical blog:
  https://bootlin.com
- Engineering services:
  https://bootlin.com/engineering
- Training services:
  https://bootlin.com/training
- Twitter:
  https://twitter.com/bootlincom
- LinkedIn:
  https://www.linkedin.com/company/bootlin
- Elixir - browse Linux kernel sources on-line:
  https://elixir.bootlin.com

Icon by Freepik, Flaticon

Training quiz and certificate

- You have been given a quiz to test your knowledge on the topics covered by the course. That's not too late to take it if you haven't done it yet!
- At the end of the course, we will submit this quiz to you again. That time, you will see the correct answers.
- It allows Bootlin to assess your progress thanks to the course. That's also a kind of challenge, to look for clues throughout the lectures and labs / demos, as all the answers are in the course!
- Another reason is that we only give training certificates to people who achieve at least a 50% score in the final quiz **and** who attended all the sessions.

Participate!

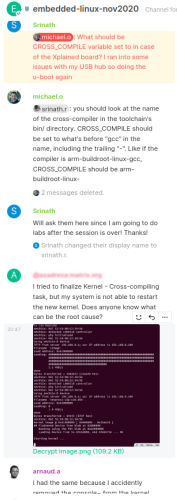
During the lectures...

- Don't hesitate to ask questions. Other people in the audience may have similar questions too.
- Don't hesitate to share your experience too, for example to compare Linux with other operating systems you know.
- Your point of view is most valuable, because it can be similar to your colleagues' and different from the trainer's.
- In on-line sessions
  - Please always keep your camera on!
  - Also make sure your name is properly filled.
  - You can also use the "Raise your hand" button when you wish to ask a question but don't want to interrupt.
- All this helps the trainer to engage with participants, see when something needs clarifying and make the session more interactive, enjoyable and useful for everyone.

Collaborate!

As in the Free Software and Open Source community, collaboration between participants is valuable in this training session:

- Use the dedicated Matrix channel for this session to add questions.
- If your session offers practical labs, you can also report issues, share screenshots and command output there.
- Don't hesitate to share your own answers and to help others especially when the trainer is unavailable.
- The Matrix channel is also a good place to ask questions outside of training hours, and after the course is over.



Base Theory and Concepts About Graphics

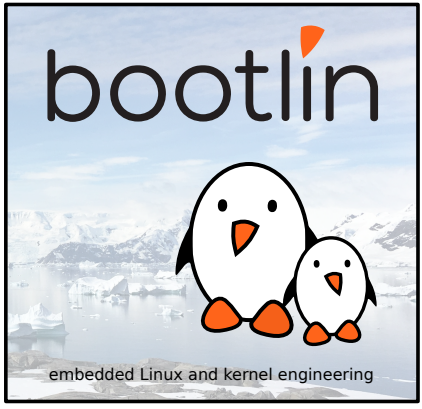
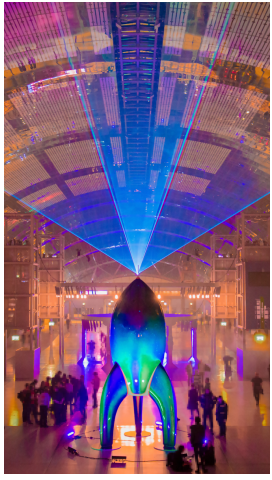


Image and Color Representation

Light, pixels and pictures

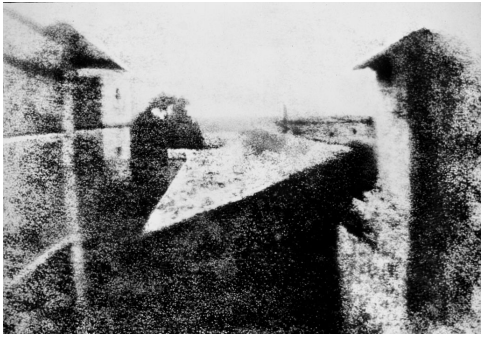
- Pictures are **representations of light emissions**
- Analog representations are spatially **continuous**:
  - With an **infinite** number of elements
  - Example: photosensitive paper
- Digital representations are spatially **quantified**:
  - With a **finite** number of elements
  - Example: discrete LED-based display
- Producing a digital representation is called **quantization**
  - Reduction of information from the continuous world
  - Quantization requires a **base element unit** or quantum
  - This quantum is called picture element or **pixel**
  - Quantization is also called **sampling** in this context



Light, pixels and pictures

- **Pictures** are bi-dimensional ordered ensembles of pixels (also called **frames**):
  - Frames have **dimensions**: **width** (horizontal) and **height** (vertical)
  - The aspect ratio is the **width:height** ratio (e.g. 16:9, 4:3)
  - Pixels are located with a **position**: **(x,y)**
  - The dimension and position unit is the number of pixels
- Quantified pixels have a **spatial density** or spatial resolution:
  - *How many pixels are found in n inches?*
  - The usual pixel resolution unit is the **dot per inch** (DPI)
  - Vertical and horizontal spatial densities are usually not distinguished
    pixels are assumed to have a square shape most of the time

Light, pixels and pictures (illustrated)



View from the Window at Le Gras picture

Analog representation,
on a metal plate

A monochrome dot-matrix display

Digital representation,
on a LED display

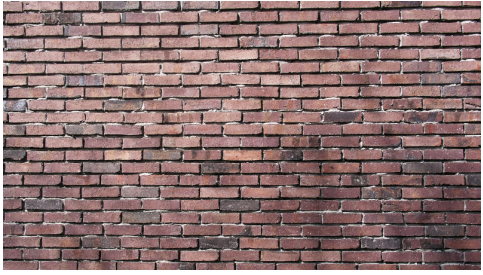
Sampling and frequency domain

- Pixels are quantized/sampled representations of a **spatial domain**
- The initial (continuous) domain has a corresponding **frequency spectrum**
  high frequencies provide details in pictures
- A 2D **Fourier transform** translates from spatial (x, y) to frequency (u, v) domain

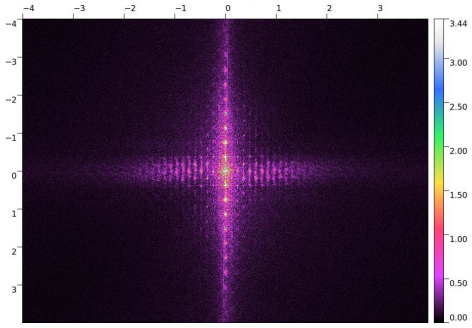
$$F(u,v) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} f(x,y) e^{-j2\pi(ux+vy)} dxdy$$

- The transform decomposes the domain in **periodic patterns**
- Adapted for discrete signals as **Discrete Fourier Transform**
- Implemented with optimized algorithms as **Fast Fourier Transform** (FFT)
- **Frequency domain analysis** is very useful for signal processing
  used at the roots of image compression

Sampling and frequency domain (illustrated)



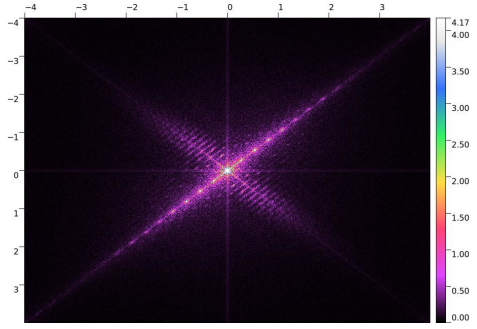
A wall of bricks represented in the spatial domain



The wall of bricks represented in the frequency domain

Sampling and frequency domain (illustrated)

A wall of bricks rotated $45°$ represented in the spatial domain



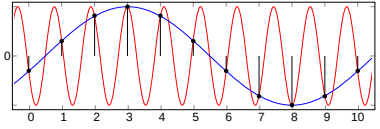
The wall of bricks rotated $45°$ represented in the frequency domain

Sampling and aliasing

- The spatial domain is quantized with a bi-dimensional **sampling resolution**
- Matching **sampling frequencies** exist, for each axis: (u_s, v_s)
- They **limit the frequencies** that can be sampled from the initial domain
- The **Shannon-Nyquist theorem** provides a sufficient condition for (u_s, v_s):

$$u_s > 2 \times u_{max}, \quad v_s > 2 \times v_{max}$$

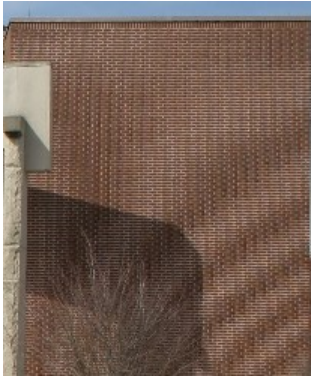
- Frequencies such that $u \geq \frac{u_s}{2}$ and $v \geq \frac{v_s}{2}$ are **not correctly sampled**
- Can result in **incorrect frequencies** being introduced: **Moiré pattern** in 2D



Aliasing example in a uni-dimensional domain

Sampling and aliasing (illustrated)

Another wall of bricks

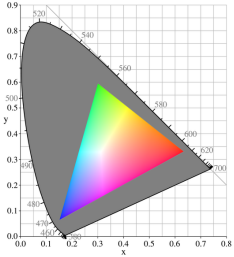


Moiré on the bricks

Moiré on the garage door

Light and color representation

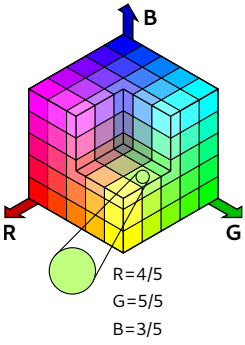
- **Light** itself must be quantized in digital representations
  distinct from and unrelated to spatial quantization
- Perceived as **colors** based on the Human visual system
  - Perception based on **trichromacy** (red, green, blue)
  - Not necessarily a unique frequency of the spectrum
    e.g. pink is not a color of the visible spectrum
- **Translating** light information (colors) to numbers:
  - A **color model** defines a base of color components
    typically 3 components (e.g. red, green, blue)
  - A **colorspace** is a precise translation referential
    unique association of a color and coordinates in the base
  - The **color gamut** is the range of colors in the colorspace
    not every color can be represented in every colorspace



Color gamut of a given colorspace

Color quantization approaches

- Different approaches exist for **color quantization**:
  - **Uniform** quantization in the color range (most common)
    values are attributed to colors with a regular step (resolution)
  - **Irregular** quantization with indexed colors (palettes)
    values are attributed to colors as needed
- Uniform color coordinates are quantized with:
  - A given **resolution**: *the smallest possible color difference*
  - A given **range**: *the span of representable colors*
- A given number of bits are used for quantization: **bit depth**
- A **trade-off** between range and resolution must be defined
  - Increasing the resolution reduces the range
  - Increasing the range reduces the resolution



Light representation, color quantization (illustrated)

A pair of Merops feeding

16 million colors (24 bits per pixel)

- high color resolution
- high color range

16 colors (4 bits per pixel)

- medium color resolution
- low color range

Light representation, color quantization (illustrated)

A pair of Merops feeding

16 million colors (24 bits per pixel)

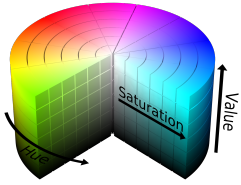
- low color resolution
- high color range

16 colors (4 bits per pixel)

- low color resolution
- high color range

Colorspaces and channels

- Each component of a color model is called a **channel**
- Examples for usual types of color models:
  - RGB, with 3 channels:
    R (red) / **G** (green) / **B** (blue)
  - HSV, with 3 channels:
    H (hue) / **S** (saturation) / **V** (value)
  - YUV or Y/Cb/Cr, with 3 channels:
    Y (luminance) / **U or Cb** / **V or Cr** (chrominance)



HSV diagram

- An additional channel can exist for transparency: the **alpha channel**
  mostly relevant for composition, not for final display
- Color coordinates can be **converted** between colorspaces and color models
  using translation formulas and associated constants

Colorspaces and channels (illustrated with YUV)

Original picture

Decomposition in Y, U and V channels

$$\begin{cases} R = Y + 1.140 \times V \\ G = Y - 0.395 \times U - 0.581 \times V \\ B = Y + 2.032 \times U \end{cases}$$

$$\begin{cases} Y = +0.299 \times R + 0.587 \times G + 0.114 \times B \\ U = -0.147 \times R - 0.289 \times G + 0.436 \times B \\ V = +0.615 \times R - 0.515 \times G - 0.100 \times B \end{cases}$$

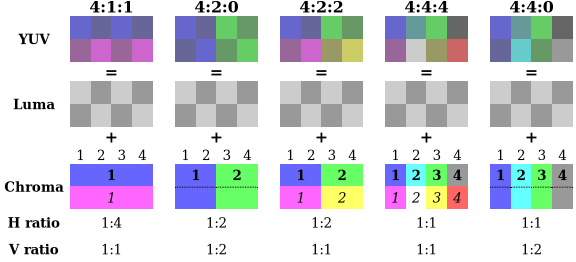
Translation between BT.601 YUV and sRGB colorspaces

Frame size and chroma sub-sampling

- Digital pictures easily take up a lot of space (more so for videos)
- The minimal size for a picture depends on:
  - Dimensions (*width* and *height*)
  - Number of bits per pixel (*bpp*): color (and alpha) depth and dead bits
  - Roughly: $width \times height \times bpp \div 8$ *bytes*
  - For 12 Mpixels with 16 Mcolors and alpha: $4000 \times 3000 \times 32 \div 8 = 45.8 MiB$
- The human visual system has specificities:
  - High sensitivity to **luminosity** (luminance)
  - Low sensitivity to **colors** (chrominance)
- The YUV color model offers the relevant channel separation
- Sub-sampling can be applied to the chrominance channel
  less data (and precision) on colors to reduce size

Frame size and chroma sub-sampling

- Chrominance samples are used for multiple luminance samples
- With specific vertical and horizontal ratios (usually integer)
- Usually summarized using a three-part ratio: $J : a : b$



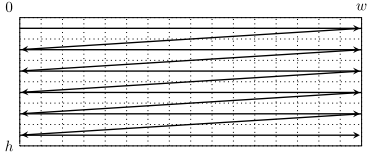
YUV 4:2:0 usual example:

$$bpp_Y = 8,\ bpp_U = bpp_V = 8 \div 2 \div 2 = 2$$

$$\Rightarrow\ bpp = bpp_Y + bpp_U + bpp_V = 12\ bits/pixel$$

Pixel data distribution in memory

- Pixel data can be **distributed** in different ways in memory
- Different ways to aggregate color components in **data planes** (memory chunks):
  - **Packed**: Components are stored in the same data plane in memory
  - **Semi-planar** (YUV): Luma and chroma are stored in distinct data planes
  - **Planar**: Each component has its own data plane in memory
- When multiple color components are grouped, **bit order** must be specified:
  - Which component comes first in memory?
  - Affected by endianness when read by hardware!
- **Scan order** must also be specified:
  - How to calculate the address for position (x, y) and back?
  - Raster order (most common) specifies: row-major, left-to-right, top-to-bottom



Raster order

Pixel formats, FourCC codes

- Many meta-data elements are needed to fully describe how a picture is coded
  - Some describe **picture-level attributes** (e.g. dimensions)
  - Some describe **pixel-level attributes** (e.g. colorspace, bpp)
- Pixel-level attributes are grouped as a **pixel format** that defines:
  - Color model in use
  - Number of bits per channel and per pixel (bpp)
  - Bit attribution and byte order
  - Per-channel sub-sampling ratios
  - Pixel data planes distribution in memory
- Often represented as a 4-character code called **FourCC**
- Not really standardized and implementation-specific:
  DRM in Linux uses `XR24` *for* `DRM_FORMAT_XRGB8888`.
  Not really standardized but widely used in various forms
- Scan order is specified separately with a **modifier**
  Assumed to be raster order if unspecified

Level of detail of quantized pictures

Depends on a number of factors, including:

- Spatial density (pixel resolution)
- Quantized dimensions (picture width and height)
- Colorspace limits (chromaticity diagram)
- Color depth (number of bits per pixel)
- Color resolution and range trade-off

Generally speaking:

- Many factors are involved
- The major bottleneck is not always obvious
- Implementation choices do matter

Pixel Drawing

Accessing pixel data

- Information required to access pixel data in memory:
  - Pixel format (also modifier if not linear/raster order)
  - Dimensions (and total size)
  - Pointer to the base buffer address
- The size of each line is called **stride** or **pitch**
  - Usually equals: $stride = width \times bpp \div 8$
  - Can contain an extra dead zone at the end
  - Also needs to be specified explicitly
- CPU access is either byte or word-aligned
  - Good fit for formats with $bpp = 32$ (very common)
  - Good fit for formats with $bpp = 8 \times n$
  - Not always easy to manage otherwise

Iterating over pixel data

- Selected format (slides and demos): **XRGB8888**
  - $bpp = 32 = 8 \times 4$, one byte per channel, one memory plane
- Pixel data can be access by iterating nested variables:

```
for (y = 0; y < height; y++)
  for (x = 0; x < width; x++)
    data = base + y * stride + x * 4;
```

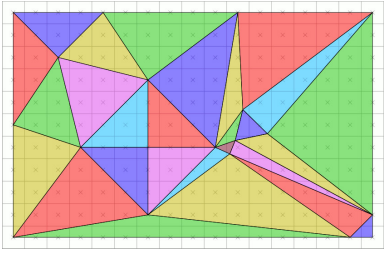
- Iterating over all pixels takes numerous CPU-cycles, tips:
  - Incrementing the address instead of re-calculating it:

```
data = base;
for (y = 0; y < height; y++)
  for (x = 0; x < width; x++)
    data += 4;
  data += stride - width * 4;
```

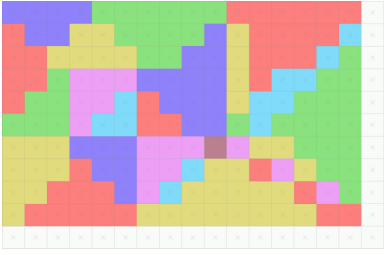
  - Iterating in y-major is also better for cache hits
  - Beware: C pointer arithmetic uses type size as unit

Concepts about rasterization

- Rasterization is the process of drawing vector shapes as discrete pixels
  - Vector shapes are defined with mathematical equations
  - Converted from a continuous space domain: $\mathbb{R}^2$ to a discrete domain
  - Results in a discretization/quantization error due to integer rounding
  - Also subject to aliasing-related trouble



Continuous source representation



Rasterized destination representation

Rectangle drawing

- A rectangle is defined with two boundaries per axis:

$$x_{min} \leq x \leq x_{max},\ y_{min} \leq y \leq y_{max}$$

- Another expression involves a (top-left) start point and size:

$$x_{start} \leq x \leq x_{start} + x_{size},\ y_{start} \leq y \leq y_{start} + y_{size}$$

- Allows iterating in the rectangle area only

Linear gradient drawing

- Same base as drawing a rectangle
- A linear gradient involves interpolation between two colors
- Following one of the two axes as major
- Involves weighting the two colors depending on the advancement
- Equations in x-axis major:

$$r = r_{start} + (r_{stop} - r_{start}) \frac{x - x_{start}}{x_{size}}$$

$$g = g_{start} + (g_{stop} - g_{start}) \frac{x - x_{start}}{x_{size}}$$

$$b = b_{start} + (b_{stop} - b_{start}) \frac{x - x_{start}}{x_{size}}$$

Disk drawing

- A disk is delimited with a radius test ($(0,0)$-centered):

$$\sqrt{x^2 + y^2} \leq \mathit{radius}$$

- Given a center point (x_c, y_c):

$$\sqrt{(x - x_c)^2 + (y - y_c)^2} \leq \mathit{radius}$$

- Requires iterating in:

$$x_c - \mathit{radius} \leq x \leq x_c + \mathit{radius},\ \ y_c - \mathit{radius} \leq y \leq y_c + \mathit{radius}$$

Circular gradient drawing

- Same base as drawing a disk
- Interpolation between two colors using the radius as major:

$$d = \sqrt{(x - x_c)^2 + (y - y_c)^2}$$

$$r = r_{start} + (r_{stop} - r_{start}) \frac{d}{radius}$$

$$g = g_{start} + (g_{stop} - g_{start}) \frac{d}{radius}$$

$$b = b_{start} + (b_{stop} - b_{start}) \frac{d}{radius}$$

Line drawing

- A line is defined as an affine function:

$$y(x) = a \times x + b$$

- Given start and end points, iterating in x-major:

$$y(x) = y_{start} + (x - x_{start}) \times \frac{y_{end} - y_{start}}{x_{end} - x_{start}}$$

$$x_{start} \leq x \leq x_{stop}$$

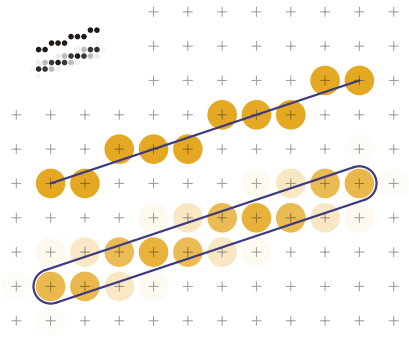
- Axis major depends on the largest per-axis span ($axis_{stop} - axis_{start}$)
  - Iterating with smaller-span axis-major results in visual holes
  - Iterating on both axes provides coherent results
- Algorithms producing better-looking results:
  - **Bresenham's** line algorithm, optimized for implementation
  - **Xiaolin Wu's** line algorithm, with sub-pixel rendering

Line and shape aliasing, sub-pixel drawing

- Lines are often subject to aliasing:
  - Sampled from the continuous domain with pixel sampling resolution
  - Selecting the best axis gives a better resolution
  - Limited display resolutions still make them look pixelated
- Any geometric shape is affected, especially fonts
- Sub-pixel rendering is used to provide anti-aliased results:
  - Surrounding pixels are given an intermediate value
  - Specific algorithms perform sub-pixel drawing
  - Also obtained with high-resolution rendering and anti-aliased downscaling

Shapes rendered without and with sub-pixel anti-aliasing

Line and shape aliasing, sub-pixel drawing (illustrated)



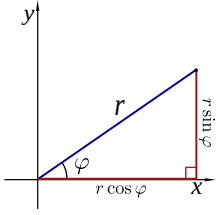
Pixel drawing versus sub-pixel drawing (x-axis major)

- Circles centered on (x_c, y_c) are defined (Pythagoras theorem) as:

$$(x - x_c)^2 + (y - y_c)^2 = radius^2$$

- Which is always verified with the expression:

$$x = x_c + radius \times cos(\phi)$$
$$y = y_c + radius \times sin(\phi)$$



- Corresponds to a translation in polar coordinates
  - From a (x, y) base to (r, ϕ)
- Iteration on ϕ with a specific range: $\phi \in [0; 2\pi]$

Parametric curves

- Parametric curves generalize the idea of using independent parameters
- Each curve has defining equations and ranges for parameters
  - Equations allow calculating (x, y) (or (r, ϕ))
- Drawing is achieved by iterating over parameter values
  - Sampling is done on the range to get a finite number of points
  - X/Y coordinates are calculated for each point
  - Line interpolation is used between consecutive points
- Ellipse: $\phi \in [0; 2\pi]$

$$x = x_c + a \times cos(\phi)$$
$$y = y_c + b \times sin(\phi)$$

- Many more parametric curves exist:
  - Cycloid, Epicycloid, Epitrochoid, Hypocycloid, Hypotrochoid (spirograph)
  - Lissajous Curve, Rose curve, Butterfly curve

Pixel Operations

Region copy

- Requirements for all pixel operations:
  - The source and destination must use the same pixel format
  - Must be converted before any operation otherwise
- Most basic operation on pixels: copying a region
  - Also known as bit blit or BITBLT (in reference to the hardware opcode)
- Implemented as a line-per-line copy (maximum memory-contiguous block)
- Overwriting destination memory with source memory
  - Copies within the same image are not always safe!
  - Destination must not overlap source

Alpha blending

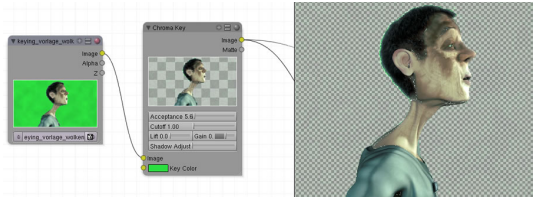
- Compositing multiple alpha-enabled pixel sources into a single result
  - Simplest case: aggregating sources with z-ordered stacking
  - Equation for A over B (with α the alpha and C the color component value):

$$C_o = \frac{C_a \alpha_a + C_b \alpha_b (1 - \alpha_a)}{\alpha_a + \alpha_b (1 - \alpha_a)}$$

- With alpha available, many more operations become possible
  - Shapes can be used as masks, with logic operators
  - Formalized by Porter and Duff in 1984

Color-keying

- Color-keying (or chroma-keying): replacing given colors with alpha
- Specified with color ranges (3 RGB ranges)
- Pixels either within or outside of the range are made transparent
- Used in conjunction with alpha blending
- The famous video green-screen method uses color-keying



Color-keying implemented in Blender

Scaling and interpolation

- Scaling is a resizing operation on a pixel picture
  - Involves a scaling factor (integer or real)
  - Values are resampled with a new resolution
  - Requires reconstructing the original signal
- Implemented with some form of interpolation:
  - nearest-neighbor: uses the nearest pixel value from the source

$$x_{source} = x_{destination} \div scale$$

  - bilinear interpolation: sub-pixel linear weighting of neighbor colors
  - bicubic interpolation: smooth spline sub-pixel fitting with neighbor colors
- Sub-pixel methods provide better visual results
- Down-sampling:
  - Reduces the maximum image frequency
  - Can cause aliasing: high frequencies need to be removed

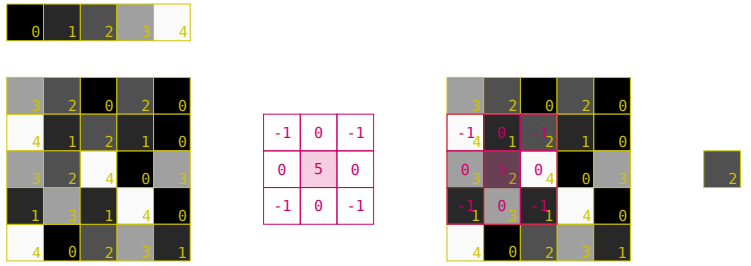
Linear filtering and convolution

- Filtering is a transformation of each pixel based on its neighbors
- The pixel output value is a linear sum of weighted neighboring input values

$$o_{x,y} = \alpha_0 i_{x,y} + \alpha_1 i_{x-1,y} + \alpha_2 i_{x+1,y} + ...$$

- Weighting coefficients are represented in a 2D matrix: the **filter kernel**
  - Comes with $2n + 1$ columns and $2m + 1$ rows
  - The coefficients are applied to each input pixel and its neighbors
  - The element at the kernel center weights the current input pixel
- Corresponds to a **convolution** operation between pixels and the filter kernel
- High computational cost (optimizations are implemented)
- Allows many applications for 2D signal processing

Linear filtering and convolution (illustrated)



Linear filtering in application

$$g(x,y) = \omega * f(x,y) = \sum_{s=-n}^{n} \sum_{t=-m}^{m} \omega(s,t) f(x-s, y-t)$$

Bi-dimensional convolution operation on f with the ω kernel

Blur filters

- Blurring is a common example of linear filtering
- Corresponds to a low-pass filter
  - Removes high frequencies from the picture (details)
  - Good fit for pre-scaling anti-aliasing
- Implemented with different algorithms:
  - Box blur: rough but easy to optimize
  - Gaussian blur: reference smooth blur

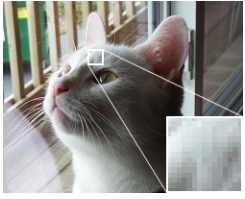
$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}$$

$$\frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix}$$

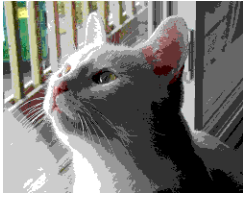
- A repeated box blur converges towards a Gaussian one (central-limit theorem)

Dithering

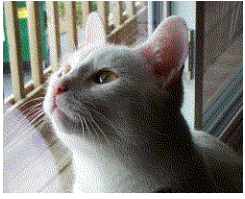
- Reducing the color depth can lead to visually-unpleasant results
  - Corresponds to color-space down-sampling
  - Increases color quantization error
- Floyd–Steinberg dithering is a method for improving quality with low depth
- Quantization error is evaluated and distributed to neighboring pixels
- Used in hardware display engines and the GIF file format



Cat at initial depth



Cat at reduced depth without dithering



Cat at reduced depth with dithering

Graphics theory online references

- Wikipedia (https://en.wikipedia.org/):
  - Color model
  - Color depth
  - YCbCr
  - Chroma subsampling
  - Nyquist–Shannon sampling theorem
  - Spatial anti-aliasing
  - Aliasing
  - Line drawing algorithm
  - Parametric equation
  - Alpha compositing
  - Image scaling
  - Kernel (image processing)
- http://ssp.impulsetrain.com/porterduff.html
- https://magcius.github.io/xplain/article/regions.html
- https://magcius.github.io/xplain/article/rast1.html

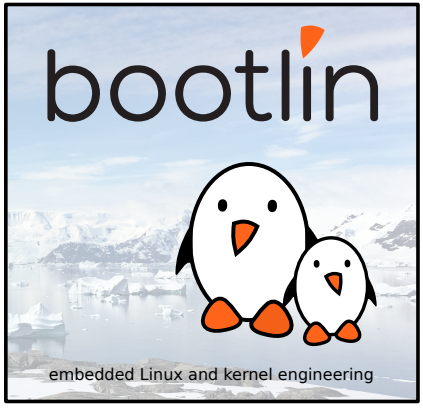
Graphics theory illustrations attributions

- 34C3 Fairy Dust: Freddy2001, CC BY-SA 3.0
- Point de vue du Gras: Joseph Nicéphore Niépce, public domain
- Pinball Dot Matrix Display: ElHeineken, CC BY 3.0
- Soderledskyrkan brick wall: Xauxa, CC BY-SA 3.0
- Aliasing Sines: Moxfyre, CC BY-SA 3.0
- Moiré pattern of bricks: Colin M.L. Burnett, CC BY-SA 3.0
- Moiré Pattern at Gardham Gap: Roger Gilbertson, CC BY-SA 2.0
- RGB cube: Datumizer, CC BY-SA 4.0
- Pair of Merops apiaster feeding: Pierre Dalous, CC BY-SA 3.0

Graphics theory illustrations attributions

- Hsl-hsv models: Datumizer, CC BY-SA 3.0
- Barns grand tetons: Jon Sullivan, public domain
- Top-left triangle rasterization rule: Drummyfish, CC0 1.0
- Line scan-conversion: Phrood, CC BY-SA 3.0
- Alpha compositing: Prometeusm, Wereon, public domain
- Blender3D com key chroma: Toni Grappa, Blender Foundation, CC BY-2.5
- Dithering example: Jamelan, CC BY-SA 3.0

Hardware Aspects



Pipeline Components Overview and Generalities

Technological types of graphics hardware implementations

- Dedicated graphics hardware is often used along a general-purpose CPU
- Two commonly-used technologies, with typical pros/cons:

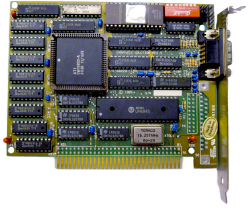
	Fixed-function	**Programmable**
Technology	Circuit	Software
Source form	HDL	Source code
Product form	Silicon, bitstream	Firmware binaries
Implementation	FPGA, ASIC, SoC block	DSP, custom ISA
Arithmetic	Fixed-point	Fixed-point, floating point
Clock rate / power	Low	High
Pixel data access	Queue (FIFO)	Memory
CPU control	Direct registers	Mailbox
Die surface	High	Low
Reusability	Low	High
Example	Allwinner SoCs Display Engine	TI TMS340 DSP

Graphics memory and buffers

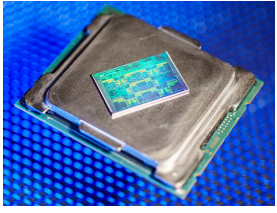
- Pixel data is stored in memory buffers, called **framebuffers**
- Framebuffers live either in:
  - **System memory**: shared with the rest of the system (e.g. SDRAM or SRAM)
  - **Dedicated memory**: only for graphics (e.g. SGRAM)
- Framebuffers that can be displayed are called **scanout framebuffers**
  hardware constraints don't always allow any framebuffer to be scanned out
- CPU access to pixel data in dedicated memory is neither always granted nor easy!
- Graphics hardware **needs configuration** to interpret framebuffer pixel data
  pixel meta-data is rarely to never stored aside of the pixel data

Display hardware overview

- **Stream pixel data** to a display device, via a display interface
- Internal pipeline with **multiple components**
- Generally **fixed-function** hardware, pipeline sink only
- Either **discrete** (video card) or **integrated**
- Connected to the CPU (and RAM) via a **high-speed bus**:
  e.g. AXI with ARM, ISA, PCI, AGP, PCI-e with x86

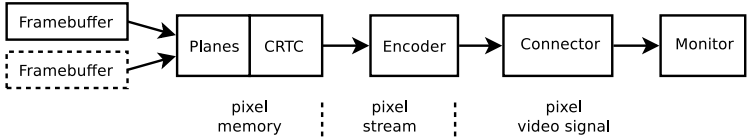


A 1986 Hercules discrete video card



An Intel processor with integrated graphics

Common components of a display pipeline overview



1. **Framebuffers** for storing the pixel data
   streamed using a DMA engine
2. **Planes** for associating a framebuffer with its dimensions and position
   composited into a single result on-the-fly
3. **CRTC** for streaming resulting pixels with specific timings
   terminology comes from the legacy Cathode-Ray Tube Controller
4. **Encoder** for meta-data addition and physical signal conversion
5. **Connector** for video signal, display data channel (DDC), hotplug detection
6. **Display** for decoding and displaying pixels (panel or monitor)

Render hardware overview

Rendering hardware includes a wide range of aspects (usual cases below):

- **Basic** pixel processing:
  - Common operations: pixel format conversion, dithering, scaling, blitting and blending
  - Fixed-function hardware, pipeline sink and source
- **Complex** pixel processing:
  - Defined by the application: any computable operation
  - Programmable hardware (DSP), pipeline sink and source
- **2D vector** drawing:
  - Rasterization from equations, parameters and data (e.g. points)
  - Either fixed-function or programmable hardware (custom), pipeline source
- **3D scene** rendering:
  - Rasterization from programs (shaders) and data (e.g. vertices, lines, triangles textures)
  - Programmable hardware (GPU), pipeline source
- Rendering can **always fallback** to general-purpose CPU operations

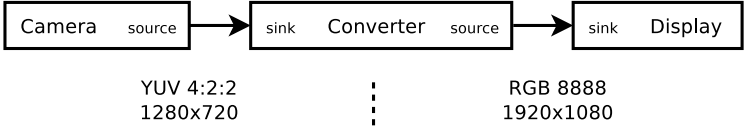
Video hardware overview

Video-oriented hardware comes in different forms (usual cases below):

- **Hardware video decoder** (VPU/video codec decoder)
  - Decodes a video from compressed data (bitstream) to pixel frames
  - Fixed-function hardware, pipeline source
- **Hardware video encoder** (VPU/video codec encoder)
  - Encodes a video from pixel frames to compressed data (bitstream)
  - Fixed-function hardware, pipeline sink
- **Camera sensors, video input, video broadcasting** (DVB)
  - Receives/sends data in a given configuration from/to *the outside*
  - Can be compressed data (bitstream) or raw pixel data
  - Fixed-function hardware, pipeline source

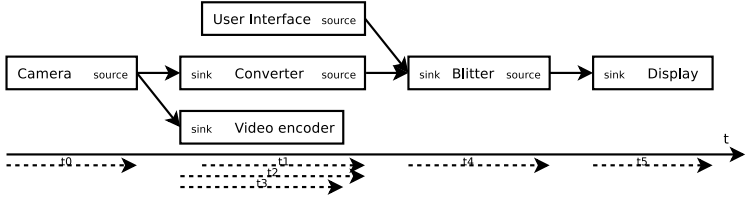
I/O with graphics hardware, pipelines

- Graphics hardware is **I/O-based** and interacts with pixel data
- Pipeline elements have input-output abilities:
  - **Source** components: **feed pixel data**: *e.g. camera*
  - **Sink** components: **grab pixel data**: *e.g. display*
- Some components are **both a source and a sink**: *e.g. converters*
- Graphics components can be chained in **pipelines**
  Usually from a source-only element to a sink-only element



Building complex pipelines

- Display, rendering and video elements are chained from source(s) to sink(s)
- On source-sink boundaries:
  - Mutually-supported pixel format (or conversion)
  - Mutually-accessible memory (or copy) or internal FIFO
- Target frame rate (*fps*) gives a time budget for pipeline traversal:
  $t_0 + t_2 + t_4 + t_5 < fps^{-1},\ t_0 + t_3 < fps^{-1},\ t_1 + t_4 + t_5 < fps^{-1}$



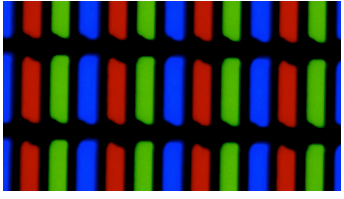
Debugging pipelines

- Debugging a complex pipeline can become **hard**:
  - Many elements in the chain means many possibilities for trouble
  - A single point of failure is enough
  - Can be difficult to identify
- Debugging **tips**:
  - Validating each element separately
  - Dumping (intermediary) framebuffers to memory/storage for validation
  - Using hardware pattern generators
- Common **pitfalls**:
  - Mismatch between advertized format and reality
  - Lack of output-input sync between elements
  - Input data updated too fast

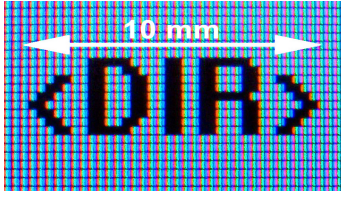
Display Hardware Specifics

Visual display technologies generalities

- Pixel data is pushed from the display interface to a visible surface *using a dedicated controller on the display device*
- Pixels are split into 3 color cells (R-G-B)
  - The human eye naturally merges light from the 3 cells
- Pixel frames are displayed as (physical) arrays of color cells
- Smooth sequences require at least 24 fps, more is usually best



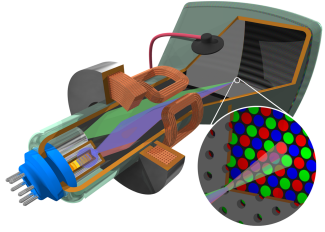
Pixel color cells on a LCD TN panel



A pixel array displaying text

CRT display technology

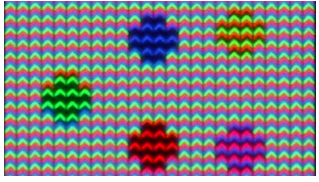
- Color **cathode-ray tubes** (CRTs), since the 1950s:
  - Using electron beams to excite a phosphorescent screen
  - Beams are guided by magnetic deflection
  - One beam for each color with increased intensity for increased luminosity
  - High energy consumption
  - High contrast, low response time ($1 - 10\ \mu s$)
  - Other issues: monitor size, burn-in (screensavers), remnant magnetic field (degaussing), high voltages and magnetic fields



Plasma display panels technology

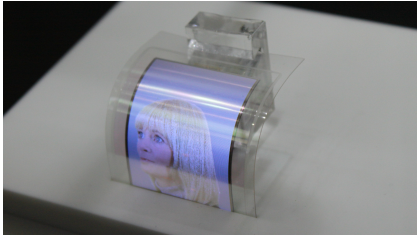
- **Plasma display panels** (PDPs), since the 1990s-2000s:
  - Using gas cells brought to plasma state to strike light-emitting phosphor
  - Flat array of cells, scales to large surfaces
  - Medium energy consumption (depends on luminance)
  - High contrast, low response time ($\leq 1\ \mu s$)
  - Other issues: burn-in
  - Gradually being **deprecated** in favor of other flat-panel technologies

- **Liquid crystal displays** (LCDs) using **Thin-film-transistors** (TFT):
  - Using the electrically-controlled alignment of crystal structures to block light
  - Does not emit light: needs an always-on backlight source (usually LEDs)
  - Low energy consumption (depends on backlight)
  - Medium to low contrast, medium response time ($1 - 10$ *ms*)
  - **Twisted nematic** (TN): limited color quality and viewing angles, since the 1980s
  - **In-plane switching** (IPS): improved color and viewing angles, since the 2000s



Chevron shapes that improve the viewing angle on IPS LCDs

- **Organic light-emitting diodes** (OLEDs), since 2010:
  - Using organic compounds (carbon-based) to emit light as R-G-B LEDs
  - Allows flat and flexible surfaces, with a large viewing angle
  - Low energy consumption
  - Very high contrast, low response time ($1 - 10\ \mu s$)
  - Issues: burn-in, independent cells aging, affected by UV light
  - Rapidly becoming **very popular** and used

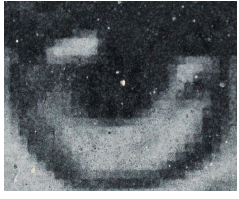


A flexible OLED display panel

EPD display technology

- **Electrophoretic displays** (EPDs), since the 2000s:
  - Using black and white electrically-charged ink-like particles in oil
    e.g. positive charge for black and negative for white
  - Electric fields attract one or the other color with current flow
    the particles stay in place after they were moved
  - Using incident light, does not emit light itself
  - Very low consumption (only for changes)
  - Very high response time (100 *ms*) and ghosting

An e-reader with an EPD display

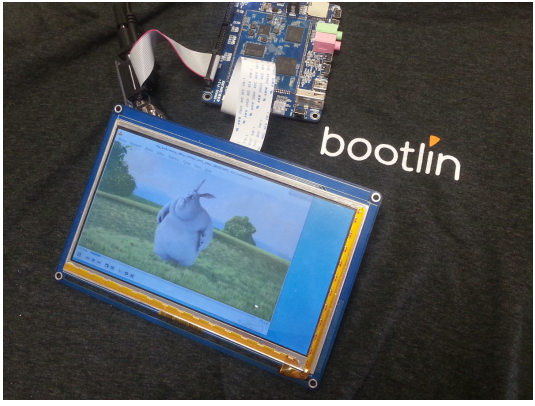


Detail of an EPD display

Display panels integration and monitors

- Panels come with a dedicated **controller** to:
  - Decode pixels from the display interface
  - Electrically control the color cells of the panel
- Panels can be used **standalone**, usually with:
  - A single simple display interface
  - No standardized connector, weak and short cables
  - Only native dimensions supported and little to no configuration
- Or they can be **integrated in monitors**, usually with:
  - More complex and multiple display interfaces
  - Standardized connectors, external cables
  - Configuration (buttons and UI overlay), multiple dimensions

Display panels integration and monitors (illustrated)



A display panel used standalone with an embedded board

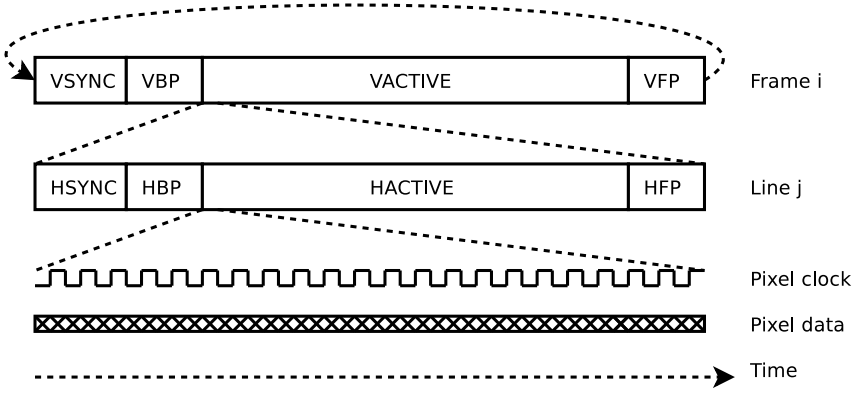
Display panels refreshing

- Most display panel technologies need frequent pixel updates: **refreshing**
  - Colors would fade out without refresh
  - Panels usually lack internal memory to self-refresh
- Requires a **fixed refresh rate**:
  - Capped by the display technology or display interface
  - Directly impacts smoothness: minimum is usually 30 *Hz*
  - The whole frame must be sent over each time
- Requires **synchronization points**:
  - Vertical: beginning of a new frame
  - Horizontal: beginning of a new line
- Requires **blank times**:
  - Account for line/frame preparation time
  - Initially for CRT electron gun repositioning
  - More or less useful depending on the technology

Display timings and modes

- Timings coordinate how a frame is **transmitted to a display over time**
- Required signals are synced to a **pixel clock** (time base)
- Display timings are split in a few stages (horizontal and vertical):
  1. Sync pulse (vsync/hsync)
  2. Back porch (vbp/hbp): blanking
  3. Active region (vactive/hactive)
  4. Front porch (vfp/hfp): blanking
- Pixels are transmitted during the **horizontal active region** (one line)
- A **display mode** groups all timing and signal characterization information
  - Signals are **generated by the CRTC** according to the display mode
- Monitors usually support **multiple modes** (and dimensions):
  - Standard modes are defined by VESA
  - Extra specific modes for the monitor can be supported

Display timings and modes (illustrated)



- The unit for horizontal stages is **one pixel clock period**
- The unit for vertical stages is **one line's duration**

Display timings and modes (panel example)

Item	Symbol	Values			Unit	Remark
		Min.	Typ.	Max.		
Horizontal Display Area	thd	-	800	-	DCLK	
DCLK Frequency	fclk	26.4	33.3	46.8	MHz	
One Horizontal Line	th	862	1056	1200	DCLK	
HS pulse width	thpw	1	-	40	DCLK	
HS Blanking	thb	46	46	46	DCLK	
HS Front Porch	thfp	16	210	354	DCLK	

Item	Symbol	Values			Unit	Remark
		Min.	Typ.	Max.		
Vertical Display Area	tvd	-	480	-	TH	
VS period time	tv	510	525	650	TH	
VS pulse width	tvpw	1	-	20	TH	
VS Blanking	tvb	23	23	23	TH	
VS Front Porch	tvfp	7	22	147	TH	

AT070TN94 panel datasheet

- $hsync = thpw = 20 \in [\![1; 40]\!]$
  $hbp = thb - thpw = 46 - 20 = 26$ (from diagram)
  $hactive = thd = 800$
  $hfp = thfp = 210 \in [\![16; 354]\!]$
  $htotal = hsync + hbp + hactive + hfp = 1056$
- $vsync = tvpw = 10 \in [\![1; 20]\!]$
  $vbp = tvb - tvpw = 23 - 10 = 13$ (from diagram)
  $vactive = tvd = 480$
  $vfp = tvfp = 22 \in [\![7; 147]\!]$
  $vtotal = vsync + vbp + vactive + vfp = 525$
- 1 frame takes: $vtotal \times htotal = 554400\ t_{clk}$
  60 frames take: $vtotal \times htotal \times 60 = 33264000\ t_{clk}$
  60 fps requires: $f_{clk} \geq 33.264\ MHz$

- Panels usually support a **range of timings**
- The pixel clock rate is often **rounded** (refresh rate not always strictly respected)

Side-channel and identification

- Monitor display connectors often come with a **Display Data Channel** (DDC)
  - Side-bus to allow communication between host and display
  - Usually based on I2C, quite slow (≈ 100 *kHz*)
- DDC provides access to the **Extended Display Identification Data** (EDID)
  - Contains the list of supported modes in a standard format
  - Usually stored in an EEPROM at I2C address $0x50$
- Another common monitor signal is **Hotplug Detect** (HPD)
  - Connected to a pin of the connector, asserted with a cable plugged
  - Can be wired to an interrupt pin to detect connection changes
- Direct panel interfaces (not monitors) usually lack DDC, EDID and HPD
  - Panel is always considered connected
  - Modes need to be known in advance

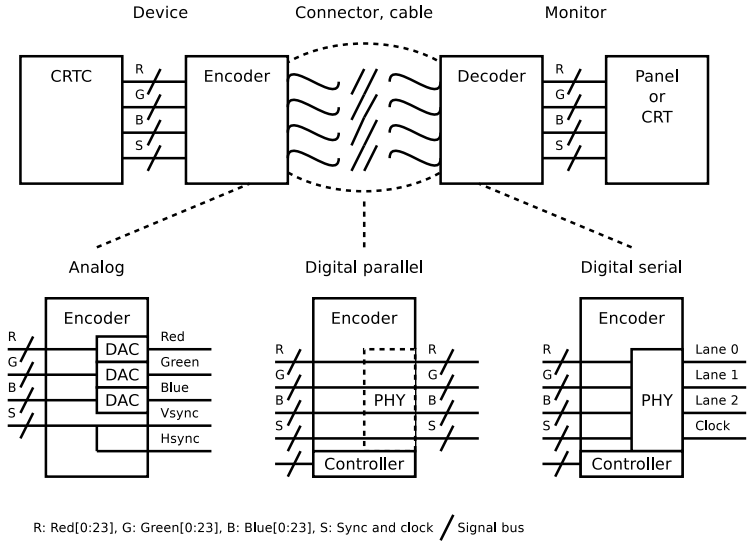
Extra display interface features and EDID extensions

- The EDID standard keeps evolving and exposes new features through extensions
- Configuration data for each feature is embedded in the EDID
- More or fewer features are supported depending on the display interface
- Common extra display interface features:
  - **Interlaced**: Every other pixel line is sent at a time, alternating between top-fields and bottom-fields; Allows faster refreshing for CRTs, needs deinterlacing for progressive panels;
  - **Audio**: Send audio in addition to pixels, during blanking periods;
  - **Stereoscopy**: Pixel data is split between two screens that show a different geometrical perspective, providing 3D perception;
  - **Variable Refresh Rate** (VRR): Pixel data can be sent at any point and does not need to conform to a given refresh rate;
  - **Consumer Electronic Control** (CEC): Remote control features on a dedicated bus;
  - **High-Bandwidth Digital Content Protection** (HDCP): Anti-copy protection

Types of display interfaces

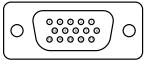
- Legacy display interfaces are usually **analog**:
  - Transmission through a DAC-based chain
  - Lack of precision, noise and chain error: not pixel-perfect, capped
  - Requires few signal pins (1 per color channel and sync or less)
- Recent interfaces are usually **digital**:
  - Encoded binary transmission, usually with dedicated clock
  - Encoders contain a controller (logic) and a PHY (signal)
  - Pixel data is expected to be bit-perfect *(but noise still exists)*
- Digital interfaces can be **parallelized**:
  - One signal per color bit (e.g. 24 signals for 24-bit RGB), clock and sync
  - One clock cycle for one pixel (low clock rate)
- Or they can be **serialized**:
  - Pixel data is sent over physical lanes (one or more)
  - One clock cycle for one bit on each lane (high clock rate)

Types of display interfaces (illustrated)



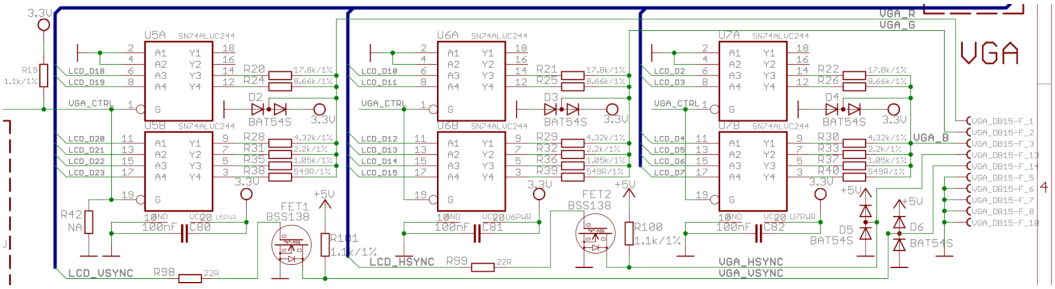
VGA display interface

- **Video Graphics Array** (VGA), since 1987 (IBM)
  - **Analog** pixel data on 3 pins (R-G-B), DAC encoder to 0.7 V peak-to-peak
  - **Per-channel pixel streaming** (voltage change), following mode timings
  - **Hsync** and **vsync** signals, I2C SDA and SDL **DDC** signals
  - Hotplug detection with R/G/B pins **current sensing**
  - Using a DB-15 connector for signals:



  - **Pixel**: Red, Green, Blue (1, 2, 3), Ground returns (6, 7, 8)
  - **Sync**: Hsync, Vsync (13, 14)
  - **Side**: DDC SDA, DDC SCL (12, 15)
  - **Power**: $+5$ V (9), Ground (10)

VGA display interface (illustrated)



- Very basic VGA encoder from parallel signals, DDC excluded
- Resistor ladder for digital-to-analog conversion
  using SN74ALVC244 voltage level shifters (1.8 V to 3.3 V), clamping diodes
- 6 most-significant bits only, 2 least-significant bits set to 0
  D0-D1, D8-D9 and D16-D17 are not routed

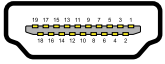
DVI display interface

- **Digital Visual Interface** (DVI), since 1999 (DDWG)
  - **DVI-A**: Analog only, comparable to VGA
  - **DVI-D**: Digital only, single-link (3 data lanes) or dual-link (6 data lanes)
  - **DVI-I**: Both analog and digital supported, single-link or dual-link
  - Digital serial link using **Transition-Minimized Differential Signaling** (TMDS)
  - Dedicated **DDC** and **HPD** signals
  - Using a subset or variation of the full **DVI-I connector** for signals:

  - **TMDS**: Data+ (2, 5, 10, 13, 18, 21), Data- (1, 4, 9, 12, 17, 20), Clock (23, 24)
  - **Analog pixel**: Red, Green, Blue (C1, C2, C3), Ground (C5)
  - **Analog sync**: Hsync, Vsync (C4, 8)
  - **Side**: DDC SDA, DDC SCL (7, 6), HPD (16)
  - **Power**: $+5$ V (14), Ground (15)

HDMI display interface

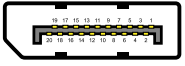
- **High-Definition Multimedia Interface** (HDMI), since 2002 (HDMI Forum)
  - Similar to **DVI-D**: no analog, 3 TMDS data lanes (R-G-B)
  - Adding the use of AVI infoframes for meta-data and audio
  - **High bandwidth** (≤ 48 *Gbit/s*) (2.1) and clock speeds (≤ 340 *MHz*)
  - **Extra features**: Audio, CEC (1.2), HDR (1.3), 4K (1.4), Stereoscopy (1.4), 8K-10K (2.1), DSC (2.1), HFR (120 *Hz*), per-frame HDR (2.1)
  - Using a dedicated (and proprietary) **HDMI connector** for signals:



  - **TMDS**: Data+ (1, 4, 7), Data- (3, 6, 9), Clock (10, 12)
  - **Side**: SDA, SCL (16, 15), HPD (19), CEC (13)
  - **Power**: $+5$ *V* (18), Ground (17)

DP/eDP display interface

- **DisplayPort** (DP), since 2008 (VESA)
  - Digital serial link with 4 data lanes using **Low-Voltage Differential Signaling** (LVDS) or TMDS for DP Dual-Mode (DP++), compatible with DVI-D and HDMI
  - Using **packets** for video/audio data and meta-data
  - Auxiliary channel encapsulating I2C DDC, CEC and more (e.g. USB)
  - **High bandwidth** (≤ 77.37 *Gbit/s*) (2.0)
  - **Extra features**: Audio, CEC, HDR (1.4), 4K (1.3), Stereoscopy (1.2), 8K (1.3-1.4), 10K-16K (2.0)
  - **Multi-Stream Transport** (MST) to chain displays
  - Using a dedicated (and proprietary) **DisplayPort connector** for signals:



  - **LVDS/TMDS**: ML+ (1, 4, 7, 10), ML- (3, 6, 9, 12)
  - **Side**: AUX+ (15), AUX- (17), HPD (18)
  - **Power**: $+3.3$ *V* (20), Ground (2, 5, 8, 11, 16)
- **Embedded DisplayPort** (eDP) for internal panels (without connector)

LVDS and DSI display interfaces

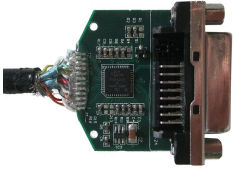
- **Low Voltage Differential Signaling** (LVDS)
  - Generic digital serial link with clock and data signals (3-4 lanes) using LVDS
  - Pixel data and control (vsync, hsync, display enable) sent as bits on lanes
  - Uses a fixed mode, no DDC, no packets
  - Specified with the JEIDA, LDI, DSIM and VESA specifications
  - For internal panels, exposed with specific connectors
  - Common for laptop panels
- **Display Serial Interface** (DSI), since 2006 (MIPI)
  - Digital serial link with clock and up to 4 data lanes using LVDS
  - Using **packets** for video data and meta-data
  - Commands for configuration can be issued with the **DSI Command Set** (DCS)
    Generic base with proprietary vendor-specific extensions
  - For internal panels, exposed with specific connectors
  - Common for mobile devices' panels

DPI display interface

- **Display Parallel Interface** (DPI)
  - Generic parallel digital interface, with 1 signal per color bit, clock and sync
  - Exists with different numbers of bits: 24 (8-8-8), 18 (6-6-6) or 16 (5-5-5)
    Dithering is required when using 16 or 18 bits
  - Sends pixel data bits following mode timings
  - Base signals: color data bits, vsync, hsync
  - Extra signals: display enable (DE)
  - Beware: sync and DE signals can be active-high or active-low
  - For internal panels, requires many signals

Bridges/transcoders

- Not every display interface is supported by the hardware at hand
- Bridges or transcoders are used to translate from one interface to another
- They are composed of a decoder and an encoder (in a single package)
- Usually standalone and transparent, often only replicate timings
  but some can have a side-bus for configuration and fine-tuning
- Example: VGA interfaces are usually bridged from digital interfaces nowadays



A DP to DVI bridge

Rendering Hardware Specifics

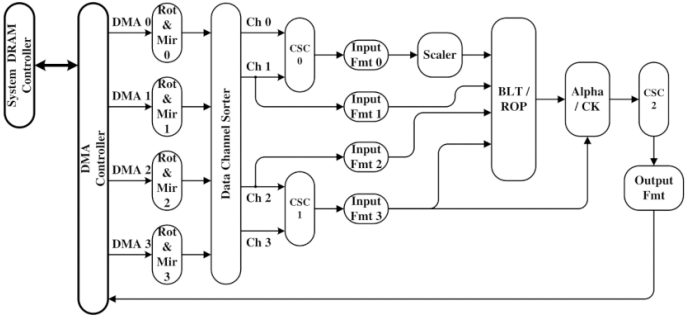
Digital Signal Processors

- Digital Signal Processors (DSPs) allow programmable image signal processing
  can also be used for implementing 2D rendering primitives
- Using a dedicated Instruction Set Architectures (ISA)
- Arithmetic implementations are either:
  - **fixed-point**: simple hardware implementation, fixed range (usually 16.16)
    16 bits for the integer part and 16 bits for the decimal part
  - **floating-point**: complex implementations, trade-off between range and precision
- Usually more power-efficient than general-purpose CPUs
- Depending on the DSP, the software can be:
  - A **standalone firmware**, usually developed from vendor libraries (C/C++/ASM)
  - A **real-time operating system** (RTOS) application (C/C++/...)
- Can be used standalone in a video pipeline or to offload a CPU
- Modern DSPs can be multi-core and feature various I/O controllers

Dedicated hardware accelerators

- Fixed-function hardware can be used for accelerating specific operations
  - Implemented as hardware circuits in Systems on a Chip (SoCs) or DSPs
  - Implemented as logic configuration bitstream in FPGAs
- Implement a configurable fixed pipeline for image operations
- Accessed and configured through specific registers exposed via a bus
  - Global configuration registers to build the pipeline between blocks
  - Configuration registers for each block
  - Kick and status registers
- Usually very power-efficient and very fast

Dedicated hardware accelerators (illustrated)



An example hardware pipeline for a 2D graphics block

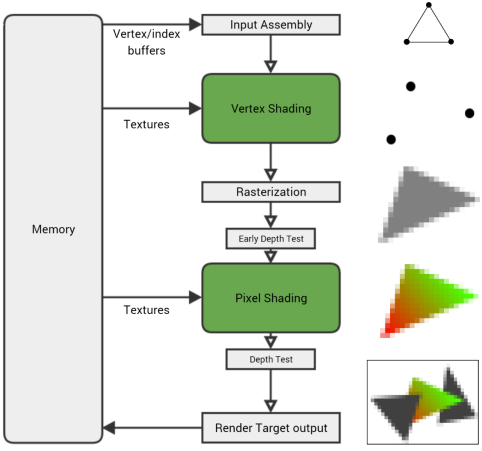
Graphics Processing Unit

- Graphics Processing Units (GPUs) are **3D rendering hardware** implementations
  the term no longer designates all graphics-processing hardware
- Operate on 3D graphics primitives: **points (vertices), lines and triangles**
- Generate a 2D view (viewport) from a given perspective
  - Objects are formed of interconnected triangles
  - A color can be applied to each vertex and interpolated
  - Textures can be applied to objects with texture element to vertex mapping
  - Lighting is applied from various sources
- Expected to render at **display scanout rate** (pseudo real-time)
  - Usually not photo-realistic methods as ray tracing or photon mapping
  - Extremely efficient compared to any general-purpose CPU
- GPUs are also used for general-purpose computing with GPGPU

Graphics Processing Unit architectures

- GPUs implement a pipeline of hacks accumulated since the 1970s
- GPU hardware architectures evolved over time:
  - From **fixed-function** configurable hardware block pipelines
  - To pipelines with both fixed blocks and specialized **programmable** processing units
- Shaders are programs that run at different steps of the pipeline:
  - **vertex shaders**: define the position, texture coordinates and lighting of each vertex
  - **geometry shaders**: generate new primitives from the provided ones
  - **tesselation shaders**: perform vertex sub-division (e.g. Catmull-Clark)
  - **fragment/pixel shaders**: perform rasterization for each output pixel
- Scenes can be rendered with multiples passes and multiple shaders

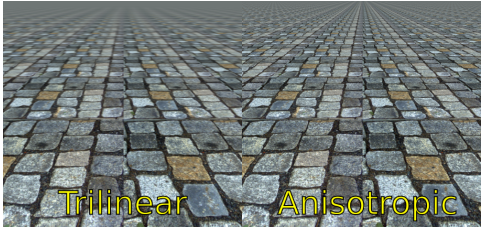
Graphics Processing Unit architectures (illustrated)



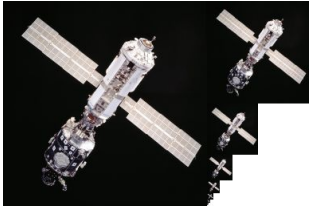
A programmable GPU pipeline

Graphics Processing Unit techniques

- GPUs are optimized for **performance** and **good-looking results**
- **Texture sampling** can easily cause aliasing (at a distance)
  - **Bilinear and trilinear filtering** is often used
  - **Anisotropic filtering** provides best visual results
  - **Mip-maps** provide the same texture at different sizes
  - **Multi-Sample Anti-Aliasing** (MSAA) averages colors from multiple points
- **Texture compression** reduces memory pressure (e.g. S3TC, ASTC)
- **Normal mapping/bump maps** provide increased details with low vertex count *affects light path calculation*
- Avoiding **useless rendering** operations:
  - Occluded surfaces are not rendered (visible surface determination)
  - Using a **depth/z-buffer** to keep track of the z-order for each output pixel
  - Culling for early vertex removal: back-face, view frustum, z-buffer



Comparison of tri-linear and anisotropic anti-aliasing filtering



A mip-mapped texture representation

Graphics Processing Unit internals

- A **command stream** is parsed and used to configure the pipeline
- Shader cores have highly-specialized ISAs adapted for geometry:
  - Vector operations, SIMD
  - Interpolation operations
  - Trigonometric operations
  - Usually few conditionals
- Texture access is provided by a **Texture Mapping Unit** (TMU)
  - Caching is used to reduce memory pressure
- Modern GPUs sometimes have a **unified shader core**
  allows efficient hardware resources usage, with complex scheduling
- Shading cores are **duplicated** and work in parallel (especially rasterization)
- Some architectures implement tiled processing:
  - Output is divided in tiles (clipping areas) and distributed to cores
  - Each rasterized tile is written to the output framebuffer separately

System Integration, Memory and Performance

Graphics integration and memory

- Graphics devices integrated in larger systems need two main interfaces:
  - **Control interface** (low speed): to program the device from the main CPU
  - **Memory interface** (high speed): to read the source data and write their framebuffer
- Other usual required elements: clocks, interrupts, reset
- Both the graphics device and the CPU need to access the memory
- Different types of memory used by graphics hardware:
  - **graphics memory**: dedicated memory attached to the graphics device
    the memory is made available to the CPU through the memory interface
  - **dedicated system memory**: a reserved contiguous area of system memory
    required when the device has no mapping unit
  - **system memory pages**: any system memory page can be mapped for access
    for devices with a dedicated IOMMU and graphics address remapping table (GART)
- Since the two parties access the same memory, cache can become incoherent
- Cache must be **synchronized** before reading and after writing, or **disabled**

Shared graphics memory access

- Concurrent access to memory can lead to trouble:
  - Concurrent read-write accesses result in partially-updated data
  - Concurrent write-write accesses result in incoherent data
- Common issue with display hardware: tearing
  - The framebuffer is scanned out at a fixed rate (e.g. 60 *fps*)
  - Any modification during scan out will result in a **partial update**
  - Causes an unpleasant **visual glitch** effect
- Solved using (at least) **double-buffering**:
  - The displayed buffer (**front buffer**) is kept intact
  - Another buffer (**back buffer**) is used for drawing the next contents
  - Front and back buffer are exchanged with **page flipping**, during vertical blanking
  - Using more buffers is possible if rendering can be done in advance

Graphics shared memory access (illustrated tearing)

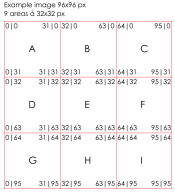
Tearing example: data is updated during scanout

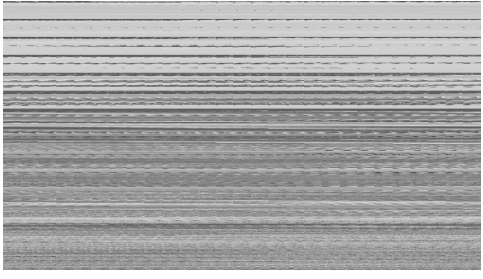
Graphics memory constraints and performance

- Fixed-pipeline 2D graphics hardware usually streams pixels through FIFOs
- A DMA engine fetches pixel data from framebuffer memory
- To simplify the logic and optimize memory access, memory constraints may apply:
  - The start address of each line needs to be aligned to 2^n
  - The byte size of each line needs to be aligned to 2^n
- The **stride** or **pitch** describes the line byte size
  - Calculated as: $ALIGN(width \times bpp/8,\ 2^n)$
  - The size of a framebuffer becomes (single-planar): $stride \times height$
    memory may be over-allocated to satisfy alignment constraints
- Pixel order in memory may not follow raster order:
  - Optimized depending on the hardware architecture
  - Optimized for efficient memory access
  - Tiled orders are frequent for parallel hardware
  - Framebuffer sizes are calculated with a specific formula

Tiled framebuffer format example

The Allwinner VPU tiled format





A tiled framebuffer read in raster order

The same framebuffer read properly

Offloading graphics to hardware

- Offloading graphics to hardware frees up significant CPU time
- For many use cases, it is crucially needed:
  - Video presentation at a given frame-rate, with format conversion and scaling
  - 3D scene rendering at display refresh rate
  - Windows and cursor composition at display refresh rate
- Offloading is not (always) a magical solution:
  - Fixed setup costs must be significantly lower than CPU processing time
    small operations are sometimes more efficient on-CPU
  - Asynchronous interrupts can introduce latency compared to active polling
    but blocking the whole system is not always an option
- 2D hardware is usually more efficient and adapted than bringing up the GPU
  the GPU is a power-hungry war machine that solves problems at a price

Graphics performance tips

- Making the most of hardware can help a lot
  e.g. camera controllers can often provide format conversion and scaling
- Generally reducing the number of devices in the graphics pipeline
- One major bottleneck in graphics pipelines is memory access:
  - Memory buffer copies must be avoided at all costs (zero-copy)
  - Chained elements in a pipeline should share the same buffer
    hardware constraints are usually compatible in SoCs
- Redrawing full frames should be avoided when possible
  - Local operations should be clipped
  - Buffer damage has to be accumulated in the multi-buffering case
- Graphics in operating systems is usually best-effort
- DSPs can usually provide real-time guarantees

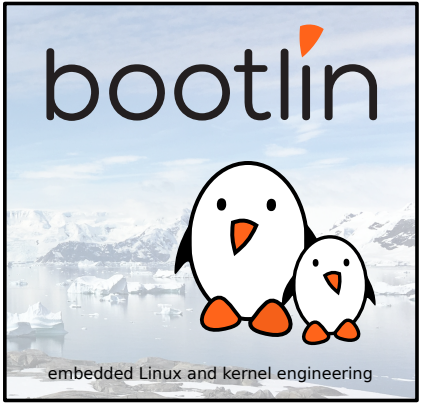
Graphics hardware online references

- Wikipedia (https://en.wikipedia.org/):
  - Graphics pipeline
  - Comparison of display technology
  - List of video connectors
  - Digital signal processor
  - Graphics processing unit
  - Tiled rendering
  - Multiple buffering

Graphics hardware illustrations attributions

- ATI Hercules Card 1986: Jörgen Nixdorf, CC BY-SA 3.0
- Intel Skylake 14nm: Fritzchens Fritz, CC0 1.0
- TN display closeup 300X: Akpch, CC BY-SA 3.0
- LCD monitor screen image: Koperczak, CC BY-SA 3.0
- CRT color enhanced unlabeled: Grm_wnr, CC BY-SA 3.0
- Dell TFT LCD: Szasz-Revai Endre, CC BY 2.5
- CeBIT 2011 3D Passport system: Bin im Garten, CC BY-SA 3.0
- Bouquin électronique iLiad en plein soleil: Mathieu Despont, public domain
- Kindle 3 E Ink screen: HorsePunchKid, CC BY-SA 3.0
- DP to DVI converter unmounted: Antifumo, CC BY-SA 3.0
- Anisotropic filtering: Thomas, Lampak, CC BY-SA 3.0
- MipMap Example STS101: Mike Hicks, NASA, CC BY-SA 3.0
- Big Bug Bunny: Blender Foundation, CC BY 3.0

Software Aspects



Display Stack Overview

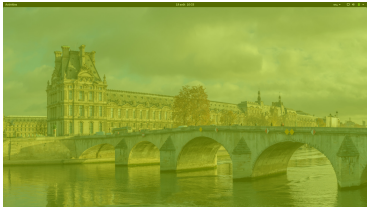
System-agnostic overview: kernel

- The kernel provides access to the hardware from userspace
- Handles clocks, power, register access, interrupts, etc
- Coordinates memory management with the rest of the system
- Exposes features to userspace through hardware-agnostic interfaces
  or at least, as much as possible
- Three aspects are usually involved:
  - **display**: from framebuffer to encoder
  - **render**: GPU and/or 2D accelerators
  - **input**: keyboard, mouse and other devices

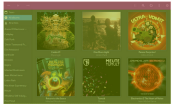
System-agnostic overview: display userspace

- The kernel provides **exclusive access** to display hardware
- Many applications need to show their buffers concurrently
- The **display server** is in charge of coordinating between applications:
  - Part of the core of the system, privileged
  - Applications (via libraries) contact the server to display pixel buffers
  - Dispatches input events to the concerned applications
  - Only the display server deals with the kernel display and input APIs
- The **compositor** merges pixel buffers from applications into the final buffer
- The **window manager** defines stacking order, focus, decorations, etc
- Both can be part of the display server or distinct components
- Serious security concerns: I/O isolation for applications, server privileges

System-agnostic overview: display userspace (illustrated)

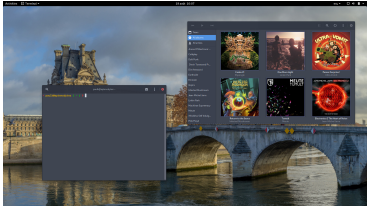


GNOME-Shell (green) displaying the top-bar and background



Lollipop (green) with window decorations (red)

GNOME-Terminal (green) with window decorations (red)

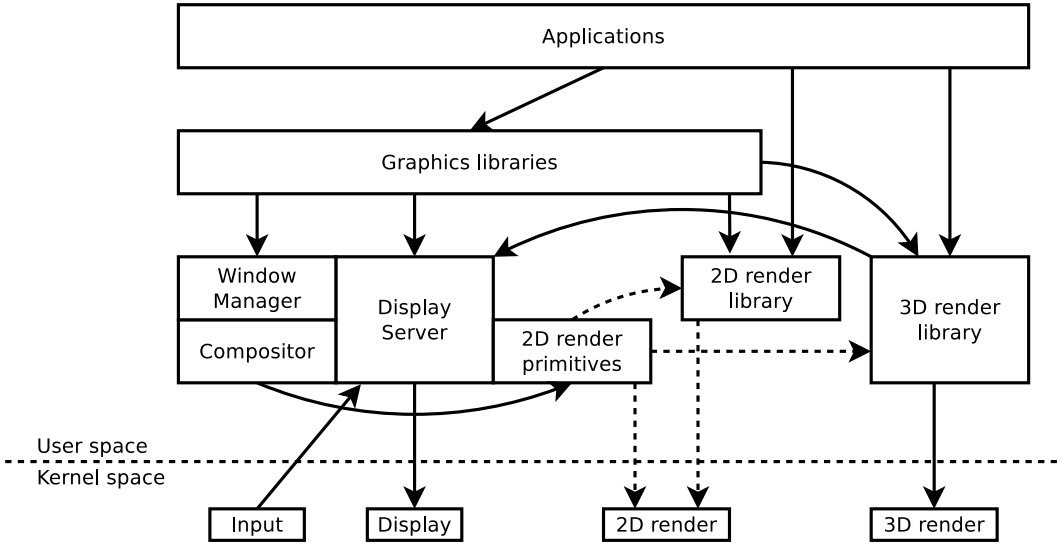


The composited result

System-agnostic overview: render userspace

- Graphics applications and libraries need to render visual elements
- Rendering can be a major **performance bottleneck**
- The system often provides **accelerated 2D primitives**:
  - Either in the display server
  - Either in dedicated libraries
- Their implementation can take different forms:
  - Using dedicated 2D hardware
  - Using 3D hardware in 2D setups ($z = 0$)
  - Using specific efficient CPU instructions (SIMD)
  - Using optimized generic algorithms
- **3D rendering** comes with its own interfaces and libraries
- Usually with generic interfaces and hardware-specific implementations

System-agnostic overview (illustrated)



Linux kernel overview

- **Input** subsystem
  - Supports devices such as mice, keyboards, joysticks, touchscreens
  - Legacy (unused) interfaces for keyboard, mice
  - Unified **evdev** event interface to simplify userspace
- **Framebuffer device** (fbdev) subsystem
  - **Legacy** interface for displaying pixel buffers on-screen
  - Very limited pipeline configuration, no hotplug support
  - Extended features added through driver-specific interfaces
- **Direct Rendering Manager** (DRM) subsystem
  - Unified display configuration interface: **Kernel Mode Setting** (KMS or DRM mode)
  - Allows synchronizing changes together (DRM atomic)
  - Exposes render devices through driver-specific interfaces (DRM render)
    Mostly for 3D rendering with GPUs, but a few 2D devices too
  - Provides memory management mechanisms (DRM GEM)

Linux-compatible low-level userspace overview

- **Input** low-level libraries
  - **libevdev** (C): Wrapper for evdev interface system calls
  - **libinput** (C): Input device management, abstraction and quirks, using libevdev
- **Display/render** low-level interface library
  - **libdrm** (C): Wrapper for DRM system calls
- **2D render** low-level libraries
  - **Pixman** (C): Optimized pixel-level operations
  - **Cairo** (C): Optimized vector drawing (can use 3D)
  - **Skia** (C): Optimized vector drawing from Google (can use 3D)
  - **Clutter** (C++): Accelerated UI animation (using 3D)
- **3D render** low-level libraries
  - **Mesa 3D** (C): Reference free software OpenGL implementation
  - **Proprietary vendor implementations** for specific hardware

X Window overview

- **X Window** overview
  - **X Window/X11** is the historical (and **legacy**) **display protocol**
  - Complemented by numerous protocol **extensions** for extra features
  - **X.org** is the reference X11 server **implementation**
  - Needs an external **window manager** to handle multiple applications
  - **Composition** by the server or the window manager (Composite extension)
- **Window manager** implementations examples
  - **Mutter**: GNOME accelerated compositing window manager
  - **i3**: Popular tiling window manager
  - **Compiz**: Popular 3D-enabled compositing window manager
- **Display client** libraries
  - **Xlib** (C): The legacy X11 client-side protocol library helper
  - **XCB** (C): The updated X11 client-side protocol library helper
  - Integrated in most higher-level graphics-oriented libraries

- **Wayland** overview
  - Wayland is a **display protocol** (with a core and extensions), not an implementation
  - Replaces X11 with a **less intrusive, more modern and minimal** paradigm
  - Compositors (server-side) handle **input, windows, composition and display**
- **Wayland compositor** implementations examples
  - Using the **libwayland-server** base protocol library
  - **Weston/libweston**: Reference implementation
  - **Sway/wlroots**: Tiling window manager and base library
  - **Mutter**: GNOME compositor
- **Display client** libraries
  - Using the **libwayland-client** base protocol library
  - Integrated in many higher-level graphics-oriented libraries

High-level graphics libraries and desktop environments overview

- Applications rarely to never use Wayland or X11 directly
- Drawing and managing a user interface is complex
- Widely-used high-level graphics libraries (aka toolkits)
  - **GTK** (C): Widget-based UI toolkit, drawing helpers (GDK)
  - **Qt** (C++): Widget-based UI toolkit, wide framework
  - **EFL** (C): Lightweight UI and application library
  - **SDL** (C): Drawing-oriented graphics library (used in games)

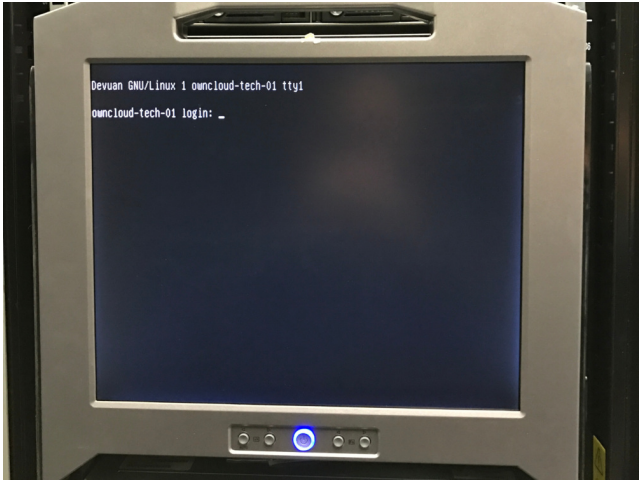
- A desktop environment groups related libraries and components *gives a consistent look and feel across the system*
- **Desktop environment** examples
  - **GNOME**: Using GTK, GNOME-Shell desktop
  - **KDE**: Using Qt, Plasma desktop
  - **Xfce**: Using GTK, lightweight
  - **Enlightenment**: Using EFL

TTY Kernel Aspects

Linux TTY subsystem introduction

- The **TTY** subsystem handles teletypewriters to send/receive characters
- Source code located at `drivers/tty` in Linux
- Supports physical instances (e.g. UART, RS-232) and virtual ones
- Virtual terminals/consoles (VTs/VCs) associate a distinct keyboard and display
  - Many VTs are created by Linux, available under: `/dev/tty*`
  - Only a single VT is active at a time, switched with `Ctrl + Alt + Fi`
  - Display grabbed using **fbcon** from the **fbdev** subsystem
  - Keyboard grabbed using the **input** subsystem
  - Can be used to show kernel messages (`console=tty1` in the cmdline)
  - Every program runs under a controlling tty (given by the `tty` command)
- Pseudo-terminals also exist, for software-based I/O only
  - Created by programs (e.g. terminal emulator) under: `/dev/pts/*`
  - Unrelated to graphics topics

Linux TTY (illustrated)



Getty running on `tty1` *of a GNU/Linux system*

Virtual terminals and graphics

- With VTs, the kernel is **already using** the display and keyboard!
- Display servers need to switch to **graphics mode** to release the display:

```
ret = ioctl(tty_fd, KDSETMODE, KD_GRAPHICS);
```

- And disable **keyboard support** on the standard input:

```
ret = ioctl(tty_fd, KDSKBMODE, K_OFF);
```

- The display device can then be used **exclusively**
- Input is no longer interpreted (e.g. `Ctrl-C` is ignored)
- Graphics and keyboard mode must be restored when leaving to keep the VT usable
- Current modes can be queried with:

```
short mode, kbmode;
ret = ioctl(tty_fd, KDGETMODE, &mode);
ret = ioctl(tty_fd, KDGKBMODE, &kbmode);
```

- More details in the `console_ioctl` man page

Virtual terminals switching and graphics

- However, the user might still want to switch VTs!
- So the display device must be **released/reacquired** for VT switching
- UNIX signals are used to notify the application, configured with:

```
struct vt_mode vt_mode = { 0 };
vt_mode.mode = VT_PROCESS;
vt_mode.relsig = SIGUSR1;
vt_mode.acqsig = SIGUSR2;
ret = ioctl(tty_fd, VT_SETMODE, &vt_mode);
```

- VT switching must be acknowledged for the other VT to take over:

```
ret = ioctl(tty_fd, VT_RELDISP, VT_ACKACQ); /* when entering VT */
ret = ioctl(tty_fd, VT_RELDISP, 1); /* when leaving VT */
```

- Failure to acknowledge will cause a **system hang**

Framebuffer Device Kernel Aspects

Fbdev overview

- Fbdev is the **historical and legacy** display subsystem in Linux
- Exposes a number of display-related features:
  - Framebuffer access (pre-allocated for one or two frames)
  - Operation primitives (bit blit, solid fill, area copy, etc)
  - Cursor drawing
  - Power management (blank/unblank, power down)
- Initial state can be configured with the `video` option of the `cmdline`
- Available to userspace via `/dev/fb*` nodes:
  - Generic base `ioctls` with driver-specific additions
  - Direct framebuffer memory mapping using `mmap`
  - Relatively simple and minimalistic interface
- Used by the kernel to provide a graphical console with **fbcon**

Fbdev basic operations

- **Fixed information** about the display is retrieved with:

```
struct fb_fix_screeninfo fix_screeninfo = { 0 };
ret = ioctl(fb_fd, FBIOGET_FSCREENINFO, &fix_screeninfo);
```

- **Variable information** (including mode) is retrieved and configured with:

```
struct fb_var_screeninfo var_screeninfo = { 0 };
ret = ioctl(fb_fd, FBIOGET_VSCREENINFO, &var_screeninfo);
...
ret = ioctl(fb_fd, FBIOPUT_VSCREENINFO, &var_screeninfo);
```

- **Power management** is operated with:

```
ret = ioctl(fb_fd, FBIOBLANK, FB_BLANK_POWERDOWN);
```

- **Double-buffering** is sometimes supported (within the same buffer):

```
var_screeninfo.yoffset = height;
ret = ioctl(fb_fd, FBIOPAN_DISPLAY, &var_screeninfo);
```

- Blocking until the next **vblank** is possible with:

```
int vsync = 0;
ret = ioctl(fb_fd, FBIO_WAITFORVSYNC, &vsync);
```

Fbdev limitations

- Fbdev does not expose or allow configuration of the display pipeline
- Output setup is mostly static (provided through the `cmdline`)
- Designed for simple cases (with a single output)
- Buffer allocation and management is not available
- No possibility of zero-copy import from other devices
- Limited page flipping with no associated synchronization mechanism
- Insufficient external synchronization interface (blocking wait)
- Mixes display, operation primitives and power management

Fbdev is mostly adapted to display from the 1990s and 2000s
please consider avoiding it at all costs!

DRM Kernel Aspects

DRM devices

- UNIX-style devices are identified with major/minor numbers
  - More details in the `makedev` manpage, using `dev_t` type
  - Minor/major can be retrieved with `stat/fstat`
  - DRM major in Linux is **226**
- Two types of DRM devices exist:
  - **Primary** nodes at `/dev/dri/card*` with minor < 128
    Used for display operations with the KMS (mode) interface
  - **Render** nodes at `/dev/dri/renderD*` with minor ≥ 128
    Used for render operations with a driver-specific interface
- DRM devices can also be used by the kernel directly (internal clients):
  - **fbdev** compatibility layer to provide `/dev/fb*` nodes
  - Used by **fbcon** to provide virtual consoles
- Userspace needs rights to open device nodes:
  - Usually allowed via the `video` **group** or **Access Control Lists** (ACLs)

DRM driver identification and capabilities

- Driver-specific **name and version** (major/minor/patchlevel) can be queried:

```
struct drm_version version = { ... };
ret = ioctl(drm_fd, DRM_IOCTL_VERSION, &version);
```

- Drivers expose specific **capabilities**, that can be queried:

```
struct drm_get_cap get_cap = { 0 };
get_cap.capability = DRM_CAP_DUMB_BUFFER;
ret = ioctl(drm_fd, DRM_IOCTL_GET_CAP, &get_cap);
```

- The kernel **must** be informed of client support for some features:

```
struct drm_set_client_cap client_cap = { 0 };
client_cap.capability = DRM_CLIENT_CAP_UNIVERSAL_PLANES;
client_cap.value = 1;
ret = ioctl(drm_fd, DRM_IOCTL_SET_CLIENT_CAP, &client_cap);
```

- Driver and client capabilities defined in Linux's `include/uapi/drm/drm.h`

DRM master, magic and authentication

- Multiple userspace clients can open the same primary device node
- Only the **master** client is allowed to configure display (KMS)
- Master is exclusive and can be **acquired** and **dropped** (VT switching):

```
ret = ioctl(drm_fd, DRM_IOCTL_SET_MASTER, NULL);
ret = ioctl(drm_fd, DRM_IOCTL_DROP_MASTER, NULL);
```

- Requires `CAP_SYS_ADMIN` Linux capability, see `capabilities` man page
  usually reserved to the root super user
- Some operations can be allowed on trusted clients with **magic authentication**:
  - Mostly used before render nodes or for allocating buffers on another process
  1. Client *foo* gets its client-specific magic:

```
struct drm_auth auth = { 0 };
ret = ioctl(drm_fd, DRM_IOCTL_GET_MAGIC, &auth);
```

  2. Client *foo* sends `auth.magic` to master client *bar* (via IPC)
  3. Master client *bar* authenticates client *foo*:

```
ret = ioctl(drm_fd, DRM_IOCTL_AUTH_MAGIC, &auth);
```

DRM memory management

- The **Graphics Execution Manager** (GEM) handles memory in DRM
- Used both by KMS and render drivers, with specific backends:
  - **CMA**: Contiguous Memory Allocator (reserved area at boot)
  - **Shmem**: Shared system memory (anonymous pages)
  - **Vram**: Video RAM, using the **Translation Table Manager** (TTM)
- Ensures buffers **coherency** on access (cache management)
- Allocated buffers are identified with a unique **handle** number
- In KMS, the **dumb buffer** API exposes memory operations:
  - For memory used for **scanout framebuffers**
  - Drivers calculate aligned pitch/stride and size based on dimensions and bpp
  - Sometimes too limiting (e.g. multi-planar formats)
- More details in the *drm-memory* man page
- Drivers sometimes expose extra `ioctls` for more advanced needs

DRM KMS dumb buffer API

- **Allocating** from `width`, `height` and `bpp`, returning `handle`, `pitch` and `size`:

```
struct drm_mode_create_dumb create_dumb = { ... };
ret = ioctl(drm_fd, DRM_IOCTL_MODE_CREATE_DUMB, &create_dumb);
```

- **Destroying** an allocated buffer:

```
struct drm_mode_destroy_dumb destroy_dumb = { .handle = ..., };
ret = ioctl(drm_fd, DRM_IOCTL_MODE_DESTROY_DUMB, &destroy_dumb);
```

- **Preparing a mapping** in user memory for a buffer, returning an `offset`:

```
struct drm_mode_map_dumb map_dumb = { .handle = ..., };
ret = ioctl(drm_fd, DRM_IOCTL_MODE_MAP_DUMB, &map_dumb);
```

- **Mapping memory** to userspace using the `offset`:

```
map = mmap(NULL, create_dumb.size, PROT_READ | PROT_WRITE, MAP_SHARED,
           drm_fd, map_dumb.offset);
```

- **Unmapping** memory after use:

```
munmap(map, create_dumb.size);
```

DRM FourCCs and modifiers

- DRM has its own representation of **pixel formats**, with FourCC codes (on 32 bits)
- Defined in the `include/uapi/drm/drm_fourcc.h` header
- They can specify up to 4 distinct **data planes** for color components
- Pixel formats are named "MSB-to-LSB" and specified in **little-endian** order
  LSB comes first in memory in little-endian
- For instance, `DRM_FORMAT_XRGB8888` has the `B` byte first in memory
  Memory order is independent from the CPU or hardware endianness
- A format **modifier** (on 64 bits) indicates the pixel order in memory
- `DRM_FORMAT_MOD_LINEAR` indicates raster order
  line-major left-to-right, top-to-bottom
- Other modifiers are usually hardware-specific, often tiled
  (e.g. `DRM_FORMAT_MOD_VIVANTE_TILED`)

DRM KMS resources probing

- KMS hardware resources are exposed through the following entities:
  - Connectors
  - Encoders
  - CRTCs
  - Planes: primary, overlay and cursor
  - Framebuffers
- Each resource instance is identified with a unique **identification number**
- The list of resource ids is retrieved with:

```
struct drm_mode_card_res res = { ... };
ret = ioctl(drm_fd, DRM_IOCTL_MODE_GETRESOURCES, &res);
```

- Plane ids (that were introduced later) are retrieved with:

```
struct drm_mode_get_plane_res res = { ... };
ret = ioctl(drm_fd, DRM_IOCTL_MODE_GETPLANERESOURCES, &res);
```

- Resource ids are used with subsequent resource-specific calls

DRM KMS connector probing

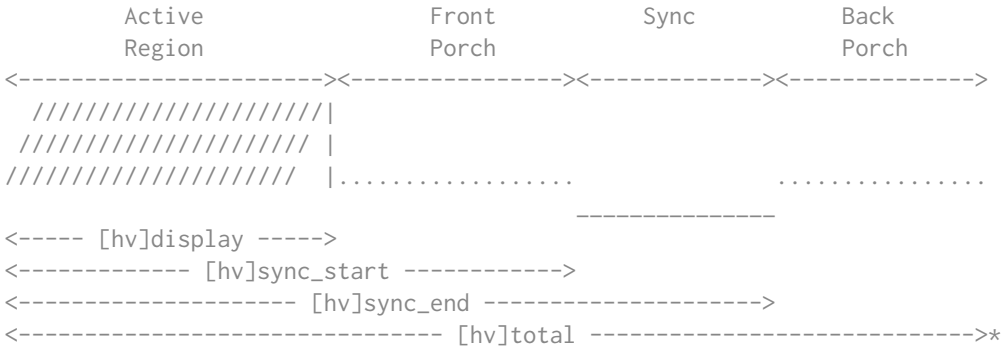
- The starting point to configure a KMS pipeline is the connector
- Current connector state is probed with:

```
struct drm_mode_get_connector get_connector = { .connector_id = ... };
ret = ioctl(drm_fd, DRM_IOCTL_MODE_GETCONNECTOR, &get_connector);
```

- `struct drm_mode_get_connector` exposes various information:
  - Connector type and connection state
  - Possible encoders, currently-attached encoder
  - Available modes and physical monitor size
- Probing modes triggers EDID read: optional and usually quite slow

DRM KMS modes

- A display mode is represented as a `struct drm_mode_modeinfo` in DRM
- Members: `clock`, `[hv]display`, `[hv]sync_start`, `[hv]sync_end`, `[hv]total` and `flags` for signal-specific details (polarities)
- Diagram from `include/drm/drm_modes.h`:



DRM KMS encoder probing

- The next step is to find which CRTC id can be used with the connector
- The encoder is the link between the connector and CRTC
- Current encoder state can be probed with:

```
struct drm_mode_get_encoder get_encoder = { .encoder_id = ... };
ret = ioctl(drm_fd, DRM_IOCTL_MODE_GETENCODER, &get_encoder);
```

- `struct drm_mode_get_connector` exposes some information:
  - Encoder type
  - Possible CRTCs, currently-attached CRTC
- This allows selecting the CRTC to use for the connector!

DRM KMS framebuffer management

- Framebuffers in DRM are described with a number of parameters:
  - Picture-wide: `width`, `height`, `pixel_format`
  - Plane-specific: GEM `handle`, `pitch`, `offset` and `modifier`
- Up to 4 memory planes are supported (depending on the format)
- Allows supporting a wide range of possible configurations
- Flags are passed to indicate that modifiers or interlaced scan are used
- Framebuffers are registered from their parameters, returning a `fb_id`:

```
struct drm_mode_fb_cmd2 fb_cmd2 = { ... };
ret = ioctl(drm_fd, DRM_IOCTL_MODE_ADDFB2, &fb_cmd2);
```

- They are destroyed using the `fb_id`:

```
unsigned int fb_id = fb_cmd2.fb_id;
ret = ioctl(drm_fd, DRM_IOCTL_MODE_RMFB, &fb_id);
```

DRM KMS CRTC configuration (legacy)

- The pipeline can then be configured with the connector and the CRTC
- The current CRTC configuration can be retrieved with:

```
struct drm_mode_crtc crtc = { .crtc_id = ... };
ret = ioctl(drm_fd, DRM_IOCTL_MODE_GETCRTC, &crtc);
```

- The CRTC is configured with the connector id

```
struct drm_mode_crtc crtc = { .crtc_id = ... };
ret = ioctl(drm_fd, DRM_IOCTL_MODE_SETCRTC, &crtc);
```

- A mode and a framebuffer can be set (previous setup used otherwise)
  mandatory if the CRTC was unused before
- The kernel will automatically select the best encoder for the connector and CRTC
- **Legacy and deprecated** way to do modesetting: only concerns the primary plane

DRM KMS page flipping (legacy)

- Page flipping is the action of switching the CRTC to another framebuffer
  only concerns the primary plane
- An event can be requested when the flip happens
- Can be scheduled at different times (specified with `flags`):
  - At a specified vblank target (absolute or relative) to avoid tearing
  - As soon as possible (asynchronously) if supported

```
struct drm_mode_crtc_page_flip page_flip = { .crtc_id = ..., .fb_id = ... };
ret = ioctl(drm_fd, DRM_IOCTL_MODE_PAGE_FLIP, &page_flip);
```

- **Legacy and deprecated**: limited to the primary plane

DRM KMS overlay plane configuration (legacy)

- Overlay planes are configured separately from the CRTC main plane
- The current state of a plane can be retrieved with:

```
struct drm_mode_get_plane get_plane = { .plane_id = ... };
ret = ioctl(drm_fd, DRM_IOCTL_MODE_GETPLANE, &get_plane);
```

- Provides possible CRTCs, current framebuffer and supported formats
- Planes are configured with source and destination parameters:
  - `crtc_[xywh]`: On-CRTC position and dimensions
  - `src_[xywh]`: In-framebuffer position and dimensions (source clipping area)
- Configuration takes place with:

```
struct drm_mode_set_plane set_plane = { .plane_id = ... };
ret = ioctl(drm_fd, DRM_IOCTL_MODE_SETPLANE, &set_plane);
```

- **Legacy and deprecated**: not synchronized to vblank or page flip

DRM KMS cursor configuration and position (legacy)

- Cursor planes have a separate dedicated legacy API
- Configured per-CRTC with a GEM `handle` and dimensions (`width`, `height`)
  a zero GEM `handle` *deconfigures and removes the cursor*
- Only supports the `DRM_FORMAT_ARGB8888` format (not configurable)
- Using a single `ioctl` with the `flags` field for the operation

```
struct drm_mode_cursor cursor = { .flags = DRM_MODE_CURSOR_BO,
                                  .crtc_id = ...};
ret = ioctl(drm_fd, DRM_IOCTL_MODE_CURSOR, &cursor);
```

- Once configured, the cursor can be moved to `x`, `y` on-CRTC coordinates

```
struct drm_mode_cursor cursor = { .flags = DRM_MODE_CURSOR_MOVE,
                                  .crtc_id = ... };
ret = ioctl(drm_fd, DRM_IOCTL_MODE_CURSOR, &cursor);
```

- `DRM_IOCTL_MODE_CURSOR2` variant provides cursor hotspot for virtual machines

DRM event notification and wait

- DRM provides an event notification mechanism for **vblank** and **page flip done**
- Available through the primary (KMS) file descriptor
- Can be used with `poll` and `select` (integrated in main loop)
- Events with a `struct drm_event` base are read using `read`
- Expand to `struct drm_event_vblank` for vblank and page flip done events
  only complete events are returned, so the buffer must be large enough
- Events can be requested at page flip time or explicitly:

```
union drm_wait_vblank wait_vblank = { .request = ... };
ret = ioctl(drm_fd, DRM_IOCTL_WAIT_VBLANK, &wait_vblank);
```

- A blocking wait for an absolute or relative vblank sequence can also be requested *using the same* `ioctl` *and dedicated* `request.type` *values*

DRM KMS object properties

- KMS objects expose generic (or driver-specific) properties with names and values *concerns **connectors**, **CRTCs** and **planes***
  - **Range** properties: limits for the value (signed or unsigned)
  - **Enum** properties: fixed values with associated names for the values
  - **Blob** properties: raw data with a given length
- Properties have a **unique identifier across objects**, details can be queried:

```
struct drm_mode_obj_get_property get_property = { .prop_id = ... }
ret = ioctl(drm_fd, DRM_IOCTL_MODE_GETPROPERTY, &get_property);
```

- Registered properties of an object can be retrieved using:

```
struct drm_mode_obj_get_properties get_properties = { .obj_id = ... }
ret = ioctl(drm_fd, DRM_IOCTL_MODE_OBJ_GETPROPERTIES, &get_properties);
```

- The `value` of a property can be assigned with:

```
struct drm_mode_obj_set_property set_property = { .obj_id = ..., .prop_id = ... }
ret = ioctl(drm_fd, DRM_IOCTL_MODE_OBJ_SETPROPERTY, &set_property);
```

- Blob properties need to be created and destroyed (with their own identifier)

DRM KMS atomic

- The legacy API comes with major design issues:
  - Overlay and cursor plane updates are applied instantly (tearing)
  - Plane updates cannot be synchronized together (intermediate states)
  - No way to check that setup is valid before applying it
- The atomic API lifts these restrictions with a new paradigm:
  - Objects are configured based on their KMS properties
    values are affected to each changed property
  - Property changes of different objects are grouped in an **atomic commit**
  - Planes are handled regardless of their type (primary, overlay, cursor)
  - Commits can be marked for test only: checked but not applied
  - Changes are applied at next vblank, unless marked asynchronous

```
struct drm_mode_atomic atomic = { ... }
ret = ioctl(drm_fd, DRM_IOCTL_MODE_ATOMIC, &atomic);
```

- Unless marked non-blocking, the `ioctl` returns when changes are applied
- A page flip event can also be requested

DRM KMS atomic common properties

- Common properties used to configure **connectors**:
  - `CRTC_ID`: id of the CRTC to bind with the connector
- Common properties used to configure **CRTCs**:
  - `ACTIVE`: whether the CRTC is in use
  - `MODE_ID`: id of the property blob with the `struct drm_mode_modeinfo` mode
- Common properties used to configure **planes**:
  - `FB_ID`: id of the framebuffer to bind with the plane
  - `CRTC_ID`: id of the CRTC to bind with the plane
  - `CRTC_[XYWH]`: on-CRTC position and dimensions of the plane
  - `SRC_[XYWH]`: in-framebuffer position and dimensions (source clipping area)
- Common properties used to probe **planes**:
  - `TYPE`: type of the plane (primary/overlay/cursor)
  - `IN_FORMATS`: list of supported formats/modifiers

DRM KMS atomic driver walkthrough

- A state-of-the-art DRM KMS driver: `vc4` at `drivers/gpu/drm/vc4/`
  integrates both DRM KMS and render
- Entry point at `drivers/gpu/drm/vc4/vc4_drv.c`
- Dedicated documentation:
  `https://dri.freedesktop.org/docs/drm/gpu/vc4.html`

DRM render generalities

- DRM render drivers have their own **driver-specific API**
  unlike KMS, render hardware abstraction is done in userspace
- Their API is exposed through custom `ioctls`
- Can be associated with a KMS driver (e.g. `vc4`) or separate (e.g. `v3d`)
- Drivers handle memory, job submission and scheduling, interrupts
- DRM has a common scheduler (from AMD) in `drivers/gpu/drm/scheduler/`
- Usual operations:
  - Managing buffer objects (BOs) of different types (create, destroy, mmap)
    using GEM under the hood
  - Submitting job data structures for programming the GPU (command lists)
    with a validation step to ensure its validity
  - Waiting for operations to complete
  - Exposing performance-related information

DRM render driver walkthrough

- A state-of-the-art DRM render driver: `v3d` at `drivers/gpu/drm/v3d/`
- Entry point at `drivers/gpu/drm/v3d/v3d_drv.c`
- Dedicated documentation: `https://dri.freedesktop.org/docs/drm/gpu/v3d.html`

DRM Prime zero-copy memory sharing (dma-buf)

- Memory buffers often need to be shared between different devices
  e.g. DRM KMS and DRM render but also concerns V4L2 for media devices
- The kernel-wide `dma-buf` API allows exporting and importing buffers
- Buffers are represented as **file descriptors** in userspace
  file descriptors can be shared between programs via IPC
- DRM exposes dma-buf via the **DRM Prime** API
- DRM prime exports a GEM `handle` to a returned `fd`:

```
struct drm_prime_handle prime_handle = { .handle = ... }
ret = ioctl(drm_fd, DRM_IOCTL_PRIME_HANDLE_TO_FD, &prime_handle);
```

- And vice-versa:

```
struct drm_prime_handle prime_handle = { .fd = ... }
ret = ioctl(drm_fd, DRM_IOCTL_PRIME_FD_TO_HANDLE, &prime_handle);
```

DRM sync object fencing

- In a multi-device pipeline with zero-copy, only scheduling is left to userspace
  each device signals completion and userspace moves on to the next
- Fences were introduced to avoid the extra roundtrip in userspace:
  - The flow of buffers between devices is usually known in advance
  - The kernel can coordinate internally and trigger the next device
  - Requires submitting all commands in advance with fences attached
- DRM exposes fences via the **Sync object** API
- Sync objects contain one fence, exposed as a file descriptor
- The KMS atomic API and some render driver APIs take input fence fds

DRM sync object fencing

- Sync objects are created and destroyed with a `handle`:

```
struct drm_syncobj_create syncobj_create = { 0 }
ret = ioctl(drm_fd, DRM_IOCTL_SYNCOBJ_CREATE, &syncobj_create);
struct drm_syncobj_destroy syncobj_destroy = { .handle = syncobj_create.handle }
ret = ioctl(drm_fd, DRM_IOCTL_SYNCOBJ_DESTROY, &syncobj_destroy);
```

- An output fence's `fd` is exported from a device's sync object with:

```
struct drm_syncobj_handle syncobj_handle = { .handle = handle, ... }
ret = ioctl(drm_fd, DRM_IOCTL_SYNCOBJ_HANDLE_TO_FD, &syncobj_handle);
```

- An input fence's `fd` is imported to a device's sync object with:

```
struct drm_syncobj_handle syncobj_handle = { .handle = handle, .fd = fd }
ret = ioctl(drm_fd, DRM_IOCTL_SYNCOBJ_FD_TO_HANDLE, &syncobj_handle);
```

- Quite a recent feature, not yet available in V4L2 (media)

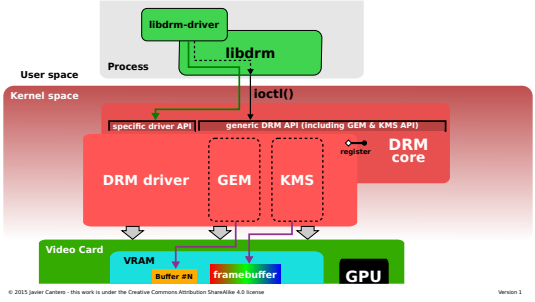
DRM debug and documentation

- **Debug message** using the `drm.debug` kernel cmdline argument:
  - Detailed in the `include/drm/drm_print.h` header
  - `drm.debug=0x17` for core, KMS, driver and atomic debug messages
- Current **state debug** in debugfs: `cat /sys/kernel/debug/dri/0/state`
- Drivers expose **specific debugfs entries**
- **Debug utility**: `modetest` from `libdrm`
- **Community** contact:
  - Mailing list: `dri-devel@lists.freedesktop.org`
  - IRC channel: `#dri-devel` on the OFTC network
- **Documentation** resources:
  - Linux GPU Driver Developer's Guide: https://www.kernel.org/doc/html/latest/gpu/index.html
  - Man pages about userspace aspects: `drm`, `drm-kms`, `drm-memory`

DRM Userspace Aspects

libdrm wrapper

- Userspace access to DRM devices is wrapped with **libdrm**
- Exposes convenience wrappers, helpers and some data structures around `ioctls`
  - For KMS support in the `libdrm.so` library
  - For hardware-specific render drivers in dedicated libraries (e.g. `libdrm_nouveau.so`)
- Used by almost every userspace project dealing with DRM:
  weston, *mutter*, *Xorg*, *mesa*, *etc*



X Window Userspace Aspects

X11 protocol and architecture

- X11 core protocol implemented by Xorg:
  - Asynchronous packet-based system with different types: `Request`, `Reply`, `Event` and `Error` packets
  - Can be used locally (UNIX socket) or over network (TCP/IP)
- Exposes **drawables** for clients to transfer or draw pixel data to the server:
  - **Windows**: area of the display buffer owned by the application *without backing storage, must be redrawn when occluded*
  - **Pixmaps**: off-display backing storage that can be copied to windows
- Windows are represented as a tree:
  - Starting with the root window created by X
  - Top-level windows and sub-windows created by clients
- A graphics context (GC) allows requesting basic drawing and font rendering
- The server provides **input events** to concerned clients:
  - Mouse movements relative to window coordinates
  - Translated key symbols a from raw keycodes

X11 protocol extensions

- X11 has evolved over time through **extensions** to its main protocol
  - Additional interfaces for X clients, matching **new hardware features**
- **XKB**: complex keyboard layouts
- **Xinput2**: touchpad, touchscreen and multi-touch support
- **XSHM**: shared client/server memory, avoiding extra transfers/copies
  not possible to operate via the network
- **XRandR**: monitor configuration and hotplugging without server restart
- **Composite**: delegates window composition to compositing window managers
- **XRender**: 2D rendering API with with alpha composition, rasterization, transformations, filtering
- **Xv**: video output format conversion and scaling offload in-DDX
  involves buffer copies and lacks synchronization with window position

Xorg architecture and acceleration

- Xorg is divided between generic and hardware-specific parts
- **Device-Independent-X** (DIX) concerns:
  - X11 protocol implementation, client coordination
  - Main event loop and event dispatching
  - Graphics operations logic, boilerplate and fallback implementations
- **Device-Dependent-X** (DDX) concerns:
  - Input drivers (`xf86-input-...`) to grab events from the kernel
  - Video drivers (`xf86-video-...`) to provide mode setting and 2D acceleration
- **EXA** provides a 2D acceleration architecture between DIX and DDX
  - Efficient way for drivers to expose accelerated 2D operation primitives
  - Replaced the XFree86 Acceleration Architecture (XAA)
  - Reduces driver boilerplate
- **Glamor** provides 2D acceleration for the DDX using OpenGL 3D rendering

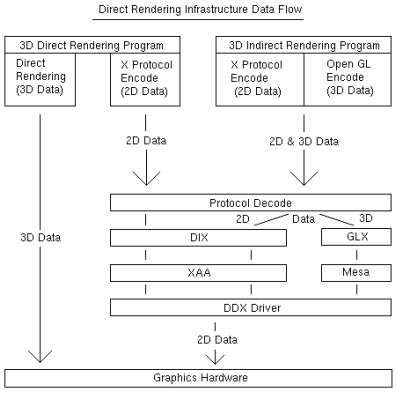
Xorg drivers overview

- Generic Xorg **input** drivers:
  - `xf86-input-libinput`: using `libinput` to get input events
  - `xf86-input-evdev`: using the `evdev` kernel interface directly (**deprecated**)
- Specific Xorg **input** drivers:
  - `xf86-input-synaptics`: for laptop touchpads
  - `xf86-input-wacom`: for Wacom drawing tablets
  - Specific drivers are **deprecated** in favor of `xf86-input-libinput`
- Generic Xorg **display** drivers:
  - `xf86-video-modesetting`: for **DRM KMS**, can be accelerated using **glamor**
  - `xf86-video-fbdev`: for the **fbdev** interface, without acceleration (legacy)
  - `xf86-video-vesa`: for the **Video BIOS Extension** (VBE) framebuffer (x86)
- Specific Xorg **display** drivers:
  - `xf86-video-[intel,nouveau,amdgpu]`: profiting from 2D acceleration blocks
  - Specific drivers are deprecated in favor of `xf86-video-modesetting` and **glamor**
    the trend is to accelerate everything via 3D rendering instead of 2D accelerators

X11 and OpenGL acceleration: GLX and DRI2

- Before DRM render nodes, there was a single device for KMS and render
  correlates with the idea of a graphics card mixing both aspects
- The X server owns the graphics device **exclusively**
- Clients using OpenGL need to access the device for rendering
- The **GLX** API was introduced to perform **indirect rendering**:
  1. Integrating OpenGL with the X Window API
  2. Forwarding GL calls to the GL implementation via the X server (AIGLX)
     introducing latency and performance issues
- **The Direct Rendering Infrastructure** (DRI/DRI2) was introduced next
  - The X server allowed access through DRM magic/auth
  - Buffers were shared via *GEM flinks*
  - Now using the standalone **render node** and **dma-buf** instead
  - Still in place for coordination between render and the display server
- GLX remained as a GL windowing API for X11 (deprecated by EGL)

X11 and OpenGL acceleration: GLX and DRI2 (illustrated)

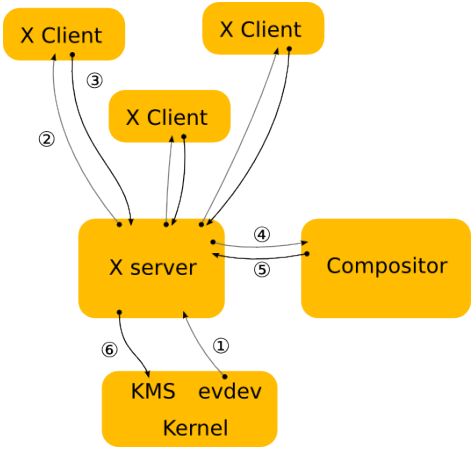


Data flow in X11 for different types of clients

Xorg usage, integration and configuration

- Xorg can be started with the `startx` command (wrapping `xinit`)
  - Executes server script from `/etc/X11/xinit/xserverrc` or `$HOME/.xserverrc`
  - Executes client script from `/etc/X11/xinit/xinitrc` or `$HOME/.xinitrc`
- An X **display manager** offers a login interface (e.g. KDM, LightDM)
  - Runs under a Xorg server, with its own dedicated user
  - Starts Xorg for authenticated users from session files in `/usr/share/xsessions/`
- Used to require **running the server as root** to access graphics devices *in particular, necessary to become DRM master*
  - The `systemd-logind` login manager lifts the restriction
  - Opens the DRM KMS fd privileged and passes it to Xorg via IPC
  - Xorg can then drop privileges: details in the `Xorg.wrap` man page
- Xorg is **configured** (both DIX and DDX) from `/etc/X11/xorg.conf`
- The `DISPLAY` environment variable indicates which server connection to use
  - Already set for X client applications and inherited
  - `export DISPLAY=:0` useful to launch programs from other TTYs

Xorg architecture: input to display roundtrip



1. An input event is read from the kernel by the server
2. The affected client is determined and receives the event
3. The client changes something and issues a rendering request
4. The server performs rendering (DDX) and notifies the compositor
5. The compositor updates the damaged regions in the back-buffer
6. The server updates the display buffer from the compositor buffer (page flip)

Major issues with X11

- The X11 core protocol and paradigm soon caused **various issues**:
  - Based on buffer **copies, transfers** and frequent **redraws**
    solved with XSHM and DRI2 extensions
  - Immediate-mode drawing, with intermediate states scanned out
    solved by drawing everything client-side instead
  - Lack of synchronization/feedback interface
    specified with the DRI3 and Present extensions
  - Everything's a window with X... but not in practice (screensavers, popups)
    specified with the DRI3 and Present extensions
  - Heavy packet-based protocol causing **latency issues**
  - **Security** concerns regarding client input/output isolation
- Because the core protocol did not evolve, **extensions proliferated**:
  - Complicated server aspects got **delegated** through extensions
  - Working around major design issues, not solving them in depth
  - In the end, the server mostly coordinates between other components
- **Client-side rendering** became more common (raster, operations, fonts, etc)

Xorg code structure and walkthrough

- Xorg source code available at: https://gitlab.freedesktop.org/xorg/xserver
- DDX components:
  - Code specific to the Linux kernel under: `hw/xfree86/os-support/linux/`
  - Modesetting DRM KMS driver under: `hw/xfree86/drivers/modesetting/`
  - fbdev core library under: `hw/xfree86/fbdevhw/`
  - Glamor implementation under: `glamor/`
- DIX components:
  - System-level helpers under: `os/`
  - Common framebuffer operations abstraction under: `fb/`
  - EXA abstraction under: `exa/`
- DRI2 components:
  - DRI2 common code under: `hw/xfree86/dri2`
  - Modesetting DRI2 glue under: `hw/xfree86/drivers/modesetting/dri2.c`
  - GLX support under: `glx/`

Xorg debug and documentation

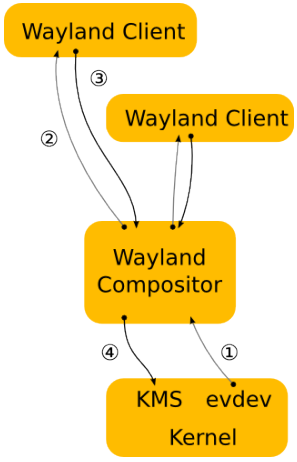
- Xorg has a **logging system** for all its components:
  - Written to a file at `/var/log/Xorg.0.log`, `-logfile` option
  - Verbosity can be set with the `-logverbose` option (log level)
  - Printed on the standard output (`stdout`)
- Xorg can be bound to any VT with the `vt` command line option
  useful for remote debugging, with a virtual controlling terminal
- **Community** contact:
  - Mailing list: `xorg@lists.freedesktop.org`
  - IRC channel: `#xorg` and `#xorg-devel` on the OFTC network
- **Documentation** resources:
  - Online wiki of the project: `https://www.x.org/wiki/Documentation/`
  - Man pages: `X`, `Xserver`, `Xorg`, `xorg.conf`, `xinit` and more!
  - Extensions specification documents

Wayland Userspace Aspects

Wayland overview and paradigm

- Wayland was started in 2008 as a modern replacement for the X Window system *solving issues in-depth with a clean implementation from scratch*
- Drastic **simplification** of the stack and paradigm shift:
  - The server and compositor are **unified** as the same component
  - **Clients** are expected to do all the rendering
  - Buffers are **shared** between client and server, no transfers
  - Window decorations can be added by the client or the server
- Improves **security** aspects:
  - Isolates the input and output of each client
  - Only the compositor can access display buffers *(and provide screenshots)*
  - Avoids running the compositor as root (using `systemd-logind`)
- No network support (can be implemented by compositors)
- **Weston** is the reference **Wayland** compositor

Wayland architecture: input to display roundtrip



1. An input event is read from the kernel by the compositor
2. The affected client is determined and receives the event
3. The client changes something, performs rendering and notifies the compositor
4. The compositor updates the damaged regions in the back-buffer and performs page flip

Wayland protocol and architecture

- Wayland provides a client-server **low-level API**:
  - Wayland **display connections** happen through local IPC
  - Display is identified with the `WAYLAND_DISPLAY` **environment variable**
  - **Asynchronous** and **object-oriented** protocol
  - **Objects** represent **resources** from the compositor
  - Objects implement specific **interfaces**, with requests and events
    - **Requests** are messages sent by the client,
    - **Events** are messages received from the server
      errors are only specific types of events
- Some **implementation** details:
  - IPC is a regular UNIX socket (allows passing file descriptors for zero-copy)
  - A proxy provides client-side object representations and message translation
  - Messages are serialized (marshalling) to the wire format
  - Messages are buffered and flushed/dispatched when asked
- Client-server protocol is implemented in **libwayland-client** and **libwayland-server**
- These libraries do not provide any interface implementation

Wayland core protocol: global object interfaces

- **Global** core object interfaces:
  - `wl_display`: manages server connection, exposes the registry
  - `wl_registry`: exposes available global object interfaces
  - `wl_output`: describes the output properties (mode, geometry)
  - `wl_seat`: exposes input device object capabilities and interfaces
  - `wl_compositor`: provides surfaces and regions for composition
  - `wl_subcompositor`: provides sub-surfaces for in-surface compositing
  - `wl_shm`: exposes a shared memory interface
  - `wl_data_device_manager`: support for copy/paste between clients
- Global object interfaces are bound with the registry before use
  - Using global `struct wl_interface` definitions
  - `wl_registry_bind` returns a pointer to a proxy object
- Global objects give access to other **specific objects**

Wayland core protocol: specific object interfaces

- **Input-related** (`wl_seat`) specific core object interfaces:
  - `wl_pointer`: exposes mice events and cursor
  - `wl_keyboard`: exposes keyboard events and information
  - `wl_touch`: exposes touchscreen events
- **Compositor-related** (`wl_compositor`) specific core object interfaces:
  - `wl_region`: specifies any area
  - `wl_surface`: rectangular on-screen pixel area
    - contents set to a `wl_buffer` (can be transformed)
    - configured with a `wl_region` for input
    - configured with a `wl_region` for area-based opacity
    - updated with buffer damage regions
  - `wl_subsurface`: converts `wl_surface` objects to positioned sub-surfaces
- **Shared memory-related** (`wl_shm`) specific core object interfaces:
  - `wl_shm_pool`: allows creating shared-memory buffers
- **Memory-related** specific core object interfaces:
  - `wl_buffer`: generic object for a `wl_surface`

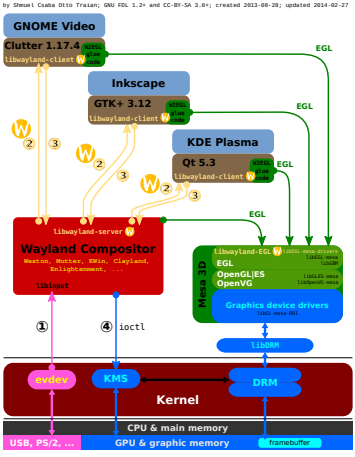
Wayland extra protocols

- Extra protocols (object interfaces) can be **exposed by the compositor**
  - Protocols (including the core) are described as XML files
  - The `wayland-scanner` tool produces client and server C code and headers
  - Accepted additional protocol descriptions are available at: `https://gitlab.freedesktop.org/wayland/wayland-protocols`
  - Some are considered stable and many unstable
- Some widely-used **protocol extensions**:
  - **XDG-Shell**: desktop shell integration
    - Turns `wl_surfaces` to `xdg_surfaces` that can be resized, minimized, etc
    - Provides a popup/menu interface with `xdg_popup`
  - **IVI-Shell**: In-vehicle shell with surface layers
  - **Presentation time**: Precise timing and event feedback

Wayland OpenGL integration

- Wayland supports EGL for windowing integration with OpenGL
- `eglGetDisplay` is called with a `struct wl_display`
  mesa's `_eglNativePlatformDetectNativeDisplay` *figures it out*
- Mesa 3D implements Wayland EGL interface for OpenGL integration
  - Needs to implement DRI2 for DRM authentication
  - `wl_drm` interface between the wayland EGL client and the compositor
  - Both sides are actually implemented in mesa
  - The interface is bound to the compositor with `eglBindWaylandDisplayWL`
    using the compositor's EGL context as entry-point to mesa
  - Allows sharing DRM GEM buffers with the compositor
- Regular `wl_surfaces` can be bound to EGL:
  - Converted to a `wl_egl_window` with `wl_egl_window_create`
  - Then converted to an `EGLSurface` with `eglCreateWindowSurface`

Wayland OpenGL integration (illustrated)

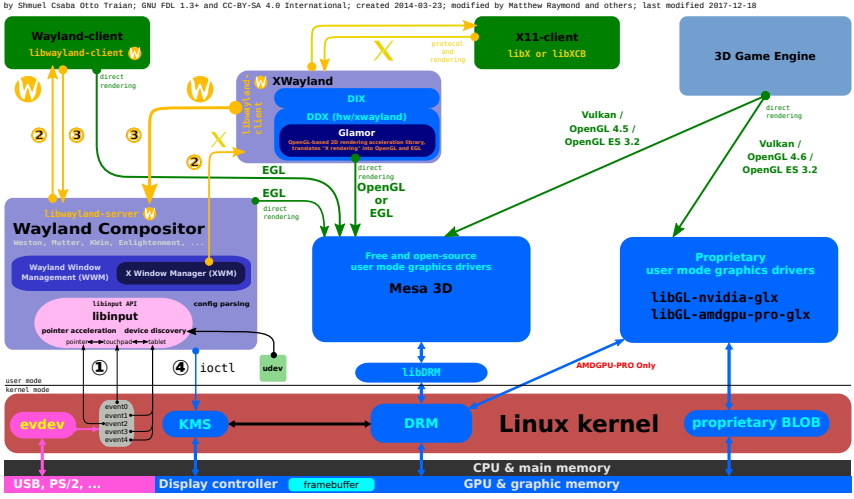


Wayland integration with EGL

Wayland status and adoption

- Wayland is now quite mature, robust, efficient and widely used
  - Most toolkits have support for it: GTK 3, Qt 5, SDL, EFL
  - Major desktop environments support it: GNOME 3, KDE Plasma 5
  - Integrated sessions with login managers from `/usr/share/wayland-sessions`
  - Runs with user privileges with `systemd-logind`
- X11 applications can be integrated using **XWayland**
  - X server implementation registered as a Wayland client
  - Wayland composer acts as X compositing window manager
  - Creates a `wl_surface` for each window, redirects input/output
- Diverse compositor implementations have emerged
  - Sometimes tied to a desktop environment: mutter, kwin
  - **libinput** was created to help with input aspects
  - **libweston** emerged from the **Weston** compositor core
  - **wlroots** emerged from the **Sway** compositor core
  - Support **DRM KMS** display, **EGL** rendering (**pixman** supported by libweston)

Wayland stack overview (illustrated)



The graphics stack with Wayland

Wayland debug and documentation

- **Debugging** tips:
  - Supported global object interfaces can be listed with `weston-info`
  - The `WAYLAND_DEBUG` environment variable enables protocol tracing
- **Weston debugging**:
  - Debug arguments: `--debug`, `--log=file.log`
  - Grabbing a different TTY argument: `--tty 1`
  - Wireframe keystroke: `mod + shift + space + F`
  - Timeline recording (to a JSON file) keystroke: `mod + shift + space + d`
    can produce a graph with https://github.com/ppaalanen/wesgr
- **Community** contact:
  - Mailing list: `wayland-devel@lists.freedesktop.org`
  - IRC channel: `#wayland` on the OFTC network
- **Documentation** resources:
  - Online wiki of the project: https://wayland.freedesktop.org/
  - Online documentation: https://wayland.freedesktop.org/docs/html/

Mesa 3D Userspace Aspects

Standardized 3D rendering APIs: OpenGL

- 3D rendering API, designed for GPU **hardware acceleration**
  - Generic API but **hardware-specific implementations**
  - Started by **Silicon Graphics** in 1992, now managed by the **Khronos Group**
- OpenGL provides a **high-level approach** to 3D graphics
  - Compromise between complexity and fine-grained control
  - Efficient abstraction, adapted to the hardware
  - Leaves most memory management to the implementation
- **Stateful** and **context-based** programming model

- OpenGL versions evolved with hardware features
  - Version 1 targeted fixed-function pipeline GPUs
  - Version 2 and up allow programming **vertex and fragment shaders**
  - More shaders supported with new versions (*geometry*, *tesselation*)
- OpenGL comes with the **GL Shading Language** (GLSL)
  - Source code language for OpenGL shaders
  - C-like syntax with intrinsic functions (e.g. texture access)
  - Compiled on-the-fly by the GL implementation
- Supports **extensions** that can be queried, for extra features

Standardized 3D rendering APIs: OpenGL ES and EGL

- **OpenGL ES** was introduced as a simplified version for embedded devices
- OpenGL ES versions are loosely following OpenGL versions:
  - Version 1 targets **fixed-function** GPUs
  - Version 2 and up target **programmable** GPUs
- Uses GLSL shaders and the same programming model as OpenGL
- **EGL** was introduced as standardized window integration API
  - Connects with the native system display server
  - Replaces GLX for X11 and adopted as default by Wayland
- Supports **extensions** that can be queried, for extra platform-specific features

Standardized 3D rendering APIs: Vulkan

- **Vulkan** is a **low-level** generic API for GPU access
  - Started by the Khronos group in 2016 and widely adopted
- Suitable for both **3D rendering** and **compute**
- Uses **Standard Portable Intermediate Representation** (SPIR-V) shaders
  - Unified intermediate representation from (adapted) GLSL/HLSL sources
  - Compiled with the program instead of on-the-fly (less overhead)
  - Translated to native GPU operations by implementations
- **Direct programming** model, with low-level **memory management**
- The API provides **window system integration** (WSI) for many platforms
  e.g. for Wayland: `vkCreateWaylandSurfaceKHR`

Mesa 3D overview

- Mesa is the reference free software **3D graphics implementation**
  - Started back in 1993, evolved with GPU implementations
  - Project works with the Khronos Group and develops extensions
- Implements support for **rendering** APIs:
  - **OpenGL** (up to 4.6) and **OpenGL ES** (up to 3.2)
  - **Vulkan** (up to 1.1) with translation to OpenGL via **Zink**
  - **Direct 3D** (version 9 only)
- Implements **windowing system** integration:
  - **EGL** for Wayland, X11, Android, native DRM (GBM) and surface-less
  - **Vulkan WSI** for Wayland and X11 (XCB/Xlib)
  - **GLX** for X11
- Also supports other **GPU-related features**:
  - **Video decoding** acceleration via VDPAU, VAAPI, OMX
  - **Compute** (GPGPU) support via OpenCL (`clover`)

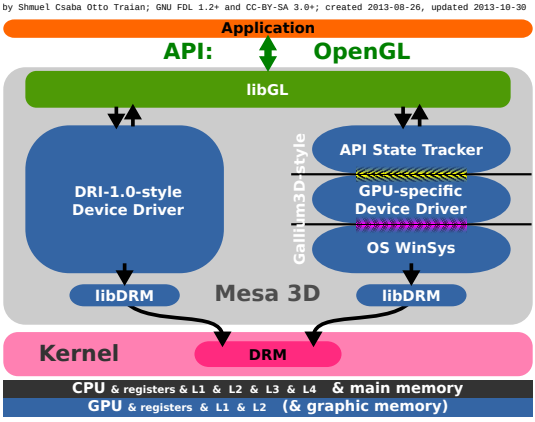
Mesa 3D implementation highlights

- Unlike other devices, 3D hardware is **abstracted in userspace**
  - 3D rendering is a very bad fit for in-kernel abstraction
  - Kernel drivers are much less complicated than GL implementations
- Mesa implements driver-specific **DRM render** support
  - Manages memory with the GEM and Prime DRM APIs
  - Manages DRI2 to allow direct rendering
- **Virtual** drivers are also supported (for virtual machines):
  - **vmwgfx**: VMware bridge (proprietary virtual hardware implementation)
  - **virgl**: Virtio bridge (standard for Linux and QEMU)
- Also provides **software backends**:
  - **softpipe**: Generic reference software renderer
  - **swr**: OpenSWR renderer (for x86 by Intel)
  - **llvmpipe**: LLVM-based renderer (high-performance)

Mesa 3D internals: Gallium 3D

- Classic mesa drivers have significant **code duplication**:
  - API state tracking
  - Compiler implementation
- The Gallium 3D interface splits things up instead:
  - **API State trackers**: maintain the current state for the API in use
  - **Drivers**: implement shader compilation and hardware configuration
  - **Winsys**: implement low-level kernel interfaces
- Gallium drivers implement a pipe interface:
  - `struct pipe_screen`: textures, buffers and sync management
  - `struct pipe_state`: pipeline configuration and resources state
  - `struct pipe_context`: rendering operation functions
- Pipe loaders (DRM or software) select the right pipe driver

Mesa 3D internals: Gallium 3D (illustrated)

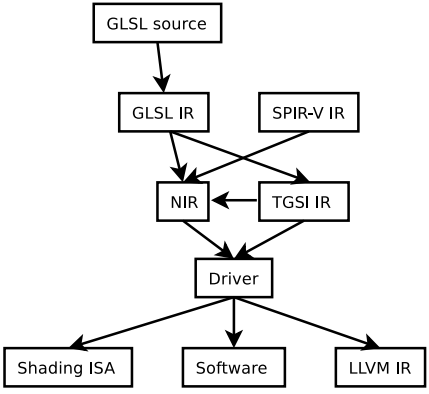


Driver integration in Mesa 3D

Mesa 3D internals: intermediate representations

- Mesa is in charge of **compiling shaders** to the native shading ISA
- **Intermediate representations** (IRs) are used for translation
- **Input-level** IRs:
  - **GLSL IR**: Internal GLSL shader representation
  - **SPIR-V**: Khronos' Standard Portable Intermediate Representation
- **Internal** IRs:
  - **TGSI IR**: Tungsten Graphics Shader Interface representation
  - **NIR**: New efficient internal representation for Mesa
  - **Mesa**: Historical implementation (deprecated)
- **External** IR:
  - **LLVM IR**: Used for LLVM interaction (e.g. with llvmpipe)
- Drivers emit **native instructions** from an internal IR and lowering
- Once ready, compiled shaders are **submitted to the GPU**

Mesa 3D internals: intermediate representations (illustrated)



The Mesa 3D internal representation pipeline

Mesa 3D Generic Buffer Management

- Mesa provides a Generic Buffer Management (GBM) interface:
  - Buffer creation/destruction (supporting modifiers)
  - Buffer information (bpp, dimensions, planes, stride, modifiers)
  - Buffer mapping/unmapping
  - Buffer dma-buf import/unimport
- Compatible with the EGL API:
  - `struct gbm_device` as `EGLNativeDisplayType`
  - `struct gbm_surface` as `EGLNativeWindowType`
  - `struct gbm_bo` as `EGLNativePixmapType`
- Provided with DRM KMS fd for DRI2 (used internally for most operations)

```
struct gbm_device *device = gbm_create_device(drm_fd);
```

- Useful when using bare-metal DRM KMS

Mesa 3D hardware support status: desktop

- Updated Mesa per-driver support at https://mesamatrix.net
- **Intel HD/Iris Graphics**
  - Platforms: Intel only
  - Mesa driver: i965 (classic), iris (Gallium)
  - DRM driver: i915
  - Status: state-of-the art (i965/iris)
- **Nvidia pre-NV110**
  - Platforms: Tegra, any PCI-e compatible
  - Mesa driver: nouveau (Gallium)
  - DRM driver: nouveau
  - Status: reverse engineered, advanced

Mesa 3D hardware support status: desktop

- **AMD Radeon GCN-ish**
  - Platforms: AMD, any PCI-e compatible
  - Mesa driver: radeonsi (Gallium)
  - DRM driver: amdgpu
  - Status: state-of-the art
- **AMD Radeon R600+**
  - Platform: AMD, any PCI-e compatible
  - Driver: r600 (Gallium)
  - DRM driver: radeon
  - Status: advanced

Mesa 3D hardware support status: embedded

- **Qualcomm Adreno**
  - Platforms: Qualcomm Snapdragon
  - Mesa driver: freedreno (Gallium)
  - DRM driver: freedreno
  - Status: reverse engineered, advanced
- **Vivante GCx000**
  - Platforms: i.MX6, i.MX8, i.MX8M
  - Driver: etnaviv (Gallium)
  - DRM driver: etnaviv
  - Status: vastly usable

Mesa 3D hardware support status: embedded

- **ARM Mali Utgard**
  - Platforms: Exynos, Allwinner, Amlogic
  - Mesa driver: lima (Gallium)
  - DRM driver: lima
  - Status: reverse engineered, usable
- **ARM Mali Midgard/Bifrost**
  - Platforms: Rockchip, Exynos, Mediatek, Allwinner
  - Mesa driver: panfrost (Gallium) / PanVK (Vulkan)
  - DRM driver: panfrost
  - Status: advanced
- **Imagination PowerVR Rogue**
  - Platforms: Mediatek
  - Mesa driver: imagination
  - DRM driver: imagination
  - Status: work in progress

Mesa 3D versus proprietary implementations

- 3D support is one of the most challenging parts of hardware integration
- Proprietary implementations easily lead to various practical issues:
  - Lack of support outside of prescribed environments
  - Lack of specific features or APIs
  - Lack of maintenance and updates
  - No adaptation possibility
- Mesa provides a collectively-maintained base
  - Constantly updated and improved by the community
  - Easier to manage: works out of the box with distributions
- Mesa support is complex and often takes some time to bring performance *especially for drivers based on reverse-engineering*

Mesa 3D code structure and walkthrough

- Mesa source code available at: https://gitlab.freedesktop.org/mesa/mesa
- **Gallium 3D** components:
  - Drivers under: `src/gallium/drivers/`
  - Winsys under: `src/gallium/winsys/`
  - API state trackers under: `src/gallium/state_trackers/`
  - Pipe loaders under: `src/gallium/auxiliary/pipe-loader/`
- **Compilation and IR** components:
  - IR compiler support under: `src/compiler/{glsl,nir,spirv}`
  - TGSI support under: `src/gallium/auxiliary/tgsi/`
  - State tracking between IRs under: `src/mesa/state_tracker`
- **Windowing and DRI2** components:
  - EGL support under: `src/egl/drivers/dri2/`
  - Vulkan WSI support under: `src/vulkan/wsi/`
- **Classic drivers** (DRI-1-style) under: `src/mesa/drivers/dri/`
- **GBM** support under: `src/gbm/`

Mesa 3D hardware support: debug and documentation

- Mesa is debugged with numerous **environment variables**
  - Generic and per-driver, see https://www.mesa3d.org/envvars.html
  - Shader-related, see https://www.mesa3d.org/shading.html
  - `LIBGL_DEBUG=verbose` for OpenGL, `EGL_LOG_LEVEL=debug` for EGL
- `eglinfo` and `glxinfo` show information about the implementation
- **Community** contact:
  - Mailing list: `dri-devel@lists.freedesktop.org`
  - IRC channel: `#dri-devel` on the OFTC network
- **Documentation** resources:
  - Online website: https://www.mesa3d.org/
  - Gallium 3D wiki: https://www.freedesktop.org/wiki/Software/gallium/
  - Gallium 3D documentation: https://gallium.readthedocs.io/
  - Source code reference: https://elixir.bootlin.com/mesa/latest/source

Graphics software online references

- Linux man pages
- Wikipedia (https://en.wikipedia.org/):
  - X Window System
  - X.Org Server
  - Wayland (display server protocol)
  - Mesa (computer graphics)
  - OpenGL
  - Vulkan (API)
- Freedesktop.org (https://freedesktop.org/):
  - Direct Rendering Infrastructure
  - DRM
  - Wayland
  - X.org wiki
- Khronos (https://khronos.org/):
  - OpenGL
  - OpenGL Wiki
  - EGL
  - Vulkan

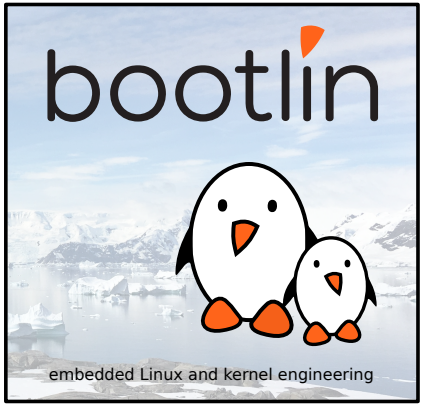
Graphics software illustrations attributions

- X11 logo: public domain
- Wayland logo: Kristian Høgsberg
- GTK logo: Andreas Nilsson, CC BY-SA 3.0
- Qt logo: Qt Project, public domain
- SDL logo: Arne Claus / SDL Project, public domain
- GNOME logo: GNOME Foundation, GNU LGPL 2.1+
- KDE logo: KDE e.V., GNU LGPL 2.1+
- XFCE logo: Xfce Team, GNU LGPL 2.1+
- Enlightenment logo: Carsten Haitzler and various contributors, BSD

Graphics software illustrations attributions

- Devuan GNU-Linux - tty login - server rack: Francesco Magno, CC BY-SA 4.0
- DRM architecture: Javier Cantero CC BY-SA 4.0
- Wayland architecture: Wayland Developers, GNU GPL 2+
- X architecture: Wayland Developers, GNU GPL 2+
- Wayland display server protocol: Shmuel Csaba Otto Traian, CC BY-SA 3.0
- The Linux Graphics Stack and glamor: Shmuel Csaba Otto Traian, CC BY-SA 3.0
- Khronos logo pack
- Gallium3D vs DRI graphics driver model, CC BY-SA 3.0

Last slides



Thank you!

And may the Source be with you

Rights to copy

Document sources: https://github.com/bootlin/training-materials/

Benchmarking, testing and validation tools

- DRM kernel aspects (display and render):
  - **IGT GPU Tools** (IGT): main DRM test suite, used for CI
    https://gitlab.freedesktop.org/drm/igt-gpu-tools
- OpenGL aspects:
  - **drawElements Quality Program** (dEQP): OpenGL/OpenGL ES/Vulkan conformance tests
    https://android.googlesource.com/platform/external/deqp
  - **glmark2**: OpenGL 2.0 and ES 2.0 benchmark tool
    https://github.com/glmark2/glmark2
- Patch series continuous integration:
  - **EzBench**: a collection of tools to benchmark graphics-related patch-series
    https://github.com/freedesktop/ezbench
- General benchmarking (including graphics):
  - **Phoronix Test Suite**: automated benchmarking tool
    https://github.com/phoronix-test-suite/phoronix-test-suite