

Linux grafika od zahteve do prikaza

Pregled grafičnega sklada v Linux operacijskih sistemih

Matic Poženel

Predmet: Računalništvo v praksi

Mentor: izr. prof. dr. Jurij Mihelič

Julij 2024

Kazalo

Uvod	1
1 Grafični sklad	1
1.1 Prikazna plast	2
1.1.1 Strojna plast	2
1.1.2 Gonilniška plast	2
1.2 Predstavitevne plasti	2
1.2.1 Upodobitvena plast	2
1.2.2 Orodna plast	3
1.2.3 Kompozicijska plast	3
1.3 Medoperacijske plasti	3
1.3.1 Plast za nadzor oken	3
1.3.2 Plast za nadzor namizja	4
1.4 Aplikacijska plast	4
2 Strojna oprema	5
2.1 Fizični medpomnilnik okvirjev (<i>framebuffer device</i>)	5
2.2 Grafične procesne enote	5
2.3 Grafične kartice	6
3 Primer: uporaba virtualnega medpomnilnika okvirjev v Linuxu	7
4 Grafični sklad operacijskega sistema Linux	12
4.1 Linux jedro in gonilniki naprav	13
4.2 Grafični API in OpenGL	14
4.2.1 Mesa in grafični gonilniki	15
4.2.2 Povezava med upravljalci oken in OpenGL	16
4.2.3 Cairo in Pixman	16
4.3 DRM in DRI	16
4.3.1 DRM	18
4.4 Primer: uporaba DRM, KMS, GBM in OpenGL za strojno-pospešen grafični prikaz	20
4.5 Okenski sistemi	21
4.5.1 Upravljač oken	21
4.5.2 Okenski sistem X	22
4.5.3 Wayland	23
4.6 GUI orodja	24
4.6.1 GTK+	26
4.6.2 Qt	26
4.7 Ostali grafični vmesniki	26
Zaključek	27
Kam naprej?	27

Slike

1	Fizični medpomnilnik okvirjev <i>Sun sbus cgsix</i> . [11]	5
2	Generična shema grafične procesne enote (GPU) [30].	6
3	Grafična kartica Sapphire Radeon HD 5570 [9]	7
4	Grafični sklad Linux operacijskega sistema z arhitekturo okenskega sistema Wayland, ki z uporabo XWayland emulira delovanje okenskega sistema X. [36]	13
5	Shema odnosa implementacije Mesa (OpenGL API-ja) z nižjimi in višjimi nivoji. [1]	15
7	Shema infrastrukture neposrednega upodabljanja. [35]	18
8	Shema podsistema DRM. [7]	19
9	Upodobitev aplikacije <code>kmscube</code> . Lastno delo.	20
10	Celoten grafični sklad v Linux operacijskem sistemu z okenskim sistemom X	22
11	Primerjava X arhitekture (levo) in Wayland arhitekture (desno). [15]	23
12	Celoten grafični sklad v Linux operacijskem sistemu z Wayland arhitekturo [37]	24
13	Poglavitni deli okna (meniji), ki so tudi sami grafični pripomočki. [38]	25
14	Primer GUI-ja, zgrajenega z orodjem GTK+. [33]	26
15	Primer GUI-ja, zgrajenega z orodjem Qt. [8]	26

Povzetek

Pričajoče delo opisuje sestavo grafičnega sklada na operacijskem sistemu Linux. Opisuje posplošen model grafičnih skladov, nekaj osnov grafične strojne opreme – medpomnilnika okvirjev (angl. *framebuffer device*), grafične procesne enote (GPE) in grafičnih kartic – in pa posamezne komponente ter najpogosteje aplikacije, ki jih zasledimo na Linuxovih sistemih. To so (poleg jedra in gonilnikov) grafični API (OpenGL z implementacijo Mesa), podistema DRM in infrastruktura DRI, okenski sistemi in prikazni strežniki (okenski sistem X in Wayland Compositor), kompozitor in ostala višenivojska grafična orodja.

Cilj je približati osnove grafičnega programiranja na Linuxu začetnikom na tem področju. Za razumevanje spodnjih konceptov je potrebno razumeti le nekaj osnov delovanja in organizacije računalniških sistemov ter delovanja Linux operacijskega sistema (npr. sistemski klici). Navedeni primeri so napisani v jeziku C.

Pričajoče delo *se ne poglablja* v matematiko, ki opisuje računalniško grafiko. Prav tako delo ni priročnik za uporabo katerekoli navedene knjižnice; tu ni opisov, katere funkcije in strukture sestavljajo OpenGL, okenski sistem X ali podsisteme Linux jedra, niti ni navodil za uporabo visokonivojskih orodij, kot je GTK+. Pričajoče delo zgolj daje izhodišče za začetnika, ta pa se lahko po svojih željah nato poglobi v posamezne dele grafičnega sklada.

Uvod

Po predmetu Vgrajeni sistemi sem postal precej navdušen nad programiranjem mikroprocesorjev, predvsem zaradi enostavnosti programiranja. Pomnilniško preslikani naslovi omogočajo preprosto interakcijo z vhodno-izhodnimi napravami (V/I napravami) – potrebno je poznati le specifikacijo in pomen bitov v njenih registrih.

Ob programiraju na višjem nivoju (native aplikacije z grafičnim vmesnikom) se mi je zato začelo porajati vprašanje – zakaj je potrebno za vsak programske jezik mukotrpno iskati grafični vmesnik, ki bi zadovoljil potrebe programerja? Še več – ali bi morda lahko spisal svoj grafični vmesnik, ki bi bil prilagojen mojim potrebam? Konec koncev je za delovanje računalnika zadolžen procesor in ta more na tak ali drugačen način sporočiti zaslonu, naj na koordinati (X,Y) aktivira piksel z neko določeno barvo.

1 Grafični sklad

Grafični sklad[28] je nabor programskih komponent, ki nadzirajo delovanje naprav (npr. gonilniki) in nudijo vmesnik za programiranje (API) z namenom prikaza grafičnih elementov na izhodni napravi. Za grafični sklad sicer ni predpisane enotne oblike, a večinoma so plasti, ki ga sestavljajo, razporejene na sledeč način (nivo najbližje strojni opremi je na lestvici najnižje):

1. Aplikacijska plast (Application layer)
2. Interoperacijske plasti (Interoperation layer)
 - 2.1. Plast za nadzor namizja (Desktop Management Layer)
 - 2.2. Plast za nadzor oken (Window Management Layer)
3. Predstavitevne plasti (Presentation Layers)
 - 3.1. Kompozicijska plast (Compositing Layer)
 - 3.2. Orodna plast (Widget Toolkit Layer)
 - 3.3. Upodobitvena plast (Rendering Layer)
4. Prikazne plasti (Display Layers)
 - 4.1. Gonilniška plast (Device Driver Layer)
 - 4.2. Strojna plast (Hardware Layer)

Omenimo še, da se tisti grafični skladovi, ki slonijo na okenskemu sistemu X nekoliko razlikujejo od ostalih, saj X uporablja ločena grafična skladova za strežnika in odjemalca zaradi svoje strežniške arhitekture [28]. To na primer pomeni, da aplikacijska plast praktično v celoti pripada odjemalcu, interoperacijske plasti pa strežniku.

V sledečih poglavjih bomo omenili tudi vmesnike, ki jih lahko uporabimo pri posameznih plasteh, kolikor je to mogoče, podrobneje pa jih bomo spoznali v poglavju, ki opisuje grafični sklad v operacijskih sistemih Linux.

Opomba: pri opisu funkcionalnosti posameznih plasti je potrebno razumeti, da višje plasti večinoma predstavljajo le olajšavo našega dela¹. Obstajajo sicer plasti, ki prinesejo nove koncepte, kot je npr. plast za nadzor oken, a bi lahko kompleksno grafiko upodabljali tudi brez koncepta oken.

1.1 Prikazna plast

1.1.1 Strojna plast

V strojni plasti najdemo strojno opremo, torej fizične naprave, ki jih uporablja višje plasti.

Vmesnik za strojno plast je ponavadi kar pomnilniško preslikan vhod/izhod. Problem je, da moramo imeti dokumentacijo uporabljenje strojne opreme, npr. grafičnih kartic, do katere pa se je zelo težko dokopati.

1.1.2 Gonilniška plast

Gonilniška plast vsebuje gonilnike, ki omogočajo delovanje bodisi specifičnih prikaznih naprav (video pomnilnika, GPE, zaslona ipd.), bodisi z neko pomnožico le-teh.

To ne pomeni, da vse sloni na gonilnikih – standard VESA BIOS Extensions (VBE) definira vmesnik, ki ga programska oprema (v tem primeru je tu mišljen predvsem BIOS) lahko uporabi za dostop do medpomnilnika okvirjev grafične kartice.

Vmesnik za gonilniško plast v operacijskih sistemih Linux predstavlja kar jedro s sistemskimi kljici `ioctl` oz. knjižnice, ki jih ovijajo, npr. `libdrm`, ter ogrodja, kot je DRI.

1.2 Predstavitevne plasti

1.2.1 Upodobitvena plast

Spološno pravilo upodobitvene plasti je, da se po nekem algoritmu iz navodil, matematičnih funkcij in podobnih abstraktnih konstruktov pridobi slika (tj. neko grafično celoto), ki se bo izrisala na zaslon, in sicer prek posebnega programa – upodobitelja (angl. *renderer, rendering engine*) [47]. Tu nastane razkorak med 2D in 3D grafiko, saj se je potrebno, na primer, odločiti, s katerim principom

¹To se na primer razlikuje od plasti pri računalniških komunikacijah. Npr. transportna plast in omrežna plast imata dejansko različne namene in ju predstavljajo različni koncepti. Pri grafičnem skladu nam npr. upodobitvena plast le močno olajša delo – vso kompleksno matematiko bi namreč lahko sprogramirali sami v zbirniku in delali neposredno v strojni plasti, a tak podvig bi bil nepredstavljivo zahteven.

se bodo izrisovala okna na zaslon. Čeprav se morda zdi, da bi lahko za vse probleme uporabljali le 3D upodabljanje (saj lahko na 2D gledamo kot na poseben primer 3D), je to računsko veliko bolj intenzivno in se na starejši strojni opremi ne obnese tako dobro, kot klasično 2D upodabljanje.

Vmesnik upodobitvene plasti predstavlja grafični API. Zelo prepoznavna je specifikacija OpenGL oz. knjižnica Mesa – s tovrstnim orodjem povemo prikazni plasti (strojni in gonilniški plasti) naj za nas izračuna željene funkcije, ki nam vrnejo okvir. Vlogo upodobitelja lahko igrajo naši programi, če neposredno izvajajo upodabljanje na GPE; tudi prikazni strežnik igra vlogo upodobitelja.

1.2.2 Orodna plast

Orodna plast večinoma skrbi za izris grafičnih kontrolnih elementov (gumbi, meniji, vnosna polja itd.), ki jih uporablja nadzornik oken. Ta plast je večinoma kar združena z upodobitveno plastjo. Grafični kontrolni elementi pa morajo (večinoma) delovati s kompozitorjem.

Vmesnik orodne plasti so GUI orodja, kot sta GTK+ in Qt – ta v ozadju pogosto uporabljajo grafične API-je iz upodobitvene plasti. Ta plast je nemalokrat kar združena s kompozicijsko, saj morajo GUI pripomočki delovati s kompozitorjem.

1.2.3 Kompozicijska plast

V kompozicijski plasti se nahaja poseben program, imenovan kompozitor (kompozicijski upravljač oken), ki skrbi za ustrezno razporeditev (kompozicijo) grafičnih elementov iz spodnjih plasti na zaslonu. Za 2D grafiko to plast nemalokrat kar enači z upodobitveno (z upodobitvene plasti), pri 3D grafiki pa je skoraj vedno samostojna.

Vmesnika za kompozicijsko plast ni preprosto določiti, saj je ta precej abstraktna, lahko pa izbiramo iz precej širokega nabora kompozitorjev, ko sestavljamo svoj grafični sklad.

1.3 Medoperacijske plasti

Opomba: Medoperacijske plasti in predstavljene plasti so med seboj povezane in bi za nekatere posamezne plasti lahko rekli, da so na enakem nivoju – primer sta npr. kompozicijska plast in plast za nadzor okenj.

1.3.1 Plast za nadzor oken

Grafično procesiranje in prikaz , kot sta bila razložena do te točke, se odvijata tako rekoč na enem platnu, poimenovanem ”okno”. Upravljač oken jih lahko izriše več in določi, na katero bo pripadal kateremu programu. Zadolžen je tudi za razporejanje oken, morebitno prekrivanje ipd.

Plast lahko enačimo s konceptom okenskega sistema. Obstaja več arhitektur: v nekaterih ga sestavlja prikaznega strežnika in upravitelja oken (npr. X), v drugih je edina komponenta kompozitor (npr. Wayland).

Tu se že srečamo z nekoliko bolj poznanimi (tržnimi) imeni, kot so AwesomeWM, qtile, i3wm in drugimi.

Vmesnik plasti za nadzor oken so knjižnice, s katerimi dostopamo do okenskega sistema – npr. libwayland, Xlib, XCB in druge.

1.3.2 Plast za nadzor namizja

Upravljač namizja je večinoma tista komponenta, na katero pomislimo, ko govorimo o grafičnih uporabniških vmesnikih (GUI-jih). Tudi tu srečamo poznana imena, kot so KDE, Gnome, XFCE, Cinnamon, MATE in drugi.

Vmesnik plasti za nadzor namizja je zopet abstrakten – kot pri kompozicijski plasti lahko tudi tu izbiramo med številnimi namiznimi okolji, kot so navedena zgoraj (KDE itd.).

1.4 Aplikacijska plast

Aplikacijska plast je prepuščena programu, ki želi uporabljati računalniško grafično. Včasih se lahko vloge te plasti pomešajo z vlogami nižjih plasti – v okenskemu sistemu X se dandanes nemalokrat zgodi, da aplikacije že same poskrbijo za upodabljanje in grafične kontrolne elemente, nato pa samo sporočijo X strežniku, naj pridelane piksle izriše na okno.

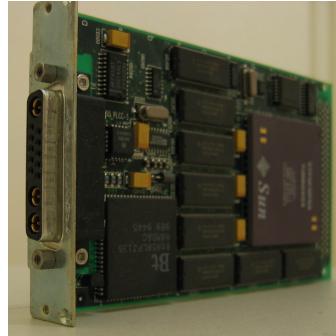
Vmesnik aplikacijske plasti je zelo širok pojem, saj lahko glede na potrebe našega programa uporabimo vmesnik katerekoli plasti. Čeprav se lahko omejimo na prikazne plasti in z velikimi mukami programiramo v zbirniku, ponavadi raje uporabimo visokonivojske knjižnice, ki služijo kot ogrodje za grafični API. Primer take knjižnice je GLFW [17].

2 Strojna oprema

Brez vhodno/izhodnih (V/I) naprav nam računalniki bore malo koristijo. Če poenostavimo: želimo si imeti nek način vnosa podatkov v pomnilnik, ter nek način pridobivanja podatkov iz pomnilnika tako, da bodo človeku berljivi – to pomeni, da želimo podatke pridobiti v grafični (besedilo, vizualizacije...), zvočni ali pa v kaki drugi, zaenkrat precej futuristični obliki. Ker se v pričujoči predstavitvi osredotočamo na grafično obliko, bomo izmed perifernih naprav zagotovo potrebovali zaslon. Sam način vhoda in izbira vhodnih perifernih naprav za potrebe te predstavitev ni bistvena (razen v določenih delih). Vprašanje se torej glasi – kako pripravimo procesor, da nam na zaslon izriše nekaj pikslov? Še bolje: kako to ukažemo Linux jedru [28]?

2.1 Fizični medpomnilnik okvirjev (*framebuffer device*)

Če preskočimo prikaze, osnovane na znanovnih celicah², so Linuxovi sistemi v začetku podpirali medpomnilnike okvirjev (MO, angl. *framebuffer*). Sprva je bila to fizična naprava, ki je, kot je moč razbrati iz imena, hranila naslednji okvir (sliko oz. polje pikslov), ki se bo izrisal na zaslon v naslednji iteraciji. Poleg tega je nudila tudi vmesnik v obliki pomnilniško preslikanih registrov, prek katerega je bilo mogoče nastavljanje načina delovanja naprave oz. njen mode (ang. mode-setting). Tako kot sodobne grafične kartice je bil na napravi vgrajen vmesnik, na katerega je bil preko ustreznegra kabla priključen zaslon. Za primer: medpomnilniku okvirjev je bilo mogoče prek pomnilniško preslikanih registrov sporočiti, da bo zaslon, ki je priključen na vmesnik MO, deloval z resolucijo 1920×1080 in frekvenco osveževanja 60Hz. To pomeni, da je bilo moč prek medpomnilnika okvirjev zaslon poganjati izven njegovih zmožnosti, kar je povzročilo fizično škodo na njem [42, Poglavlje Display Modes] [34].



Slika 1: Fizični medpomnilnik okvirjev *Sun sbus cgsix*. [11]

2.2 Grafične procesne enote

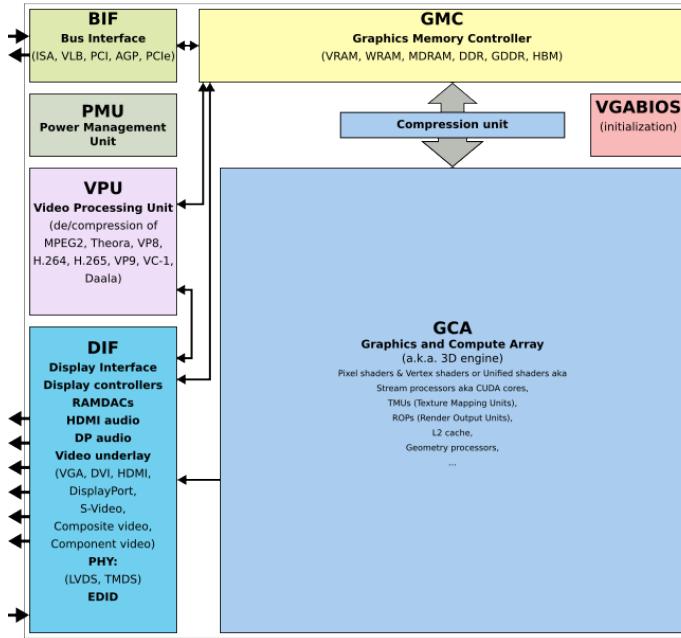
Uporaba MO pa je bila poleg teh varnostnih hib precej neučinkovit pristop za upodabljanje 3D grafike. Upodabljanje, ki sloni skoraj izključno na linearni

²Teh danes večinoma ni več, saj so nastali zaradi dragega spomina in dragih fizičnih medpomnilnikov okvirjev [49]. Danes gre vse prej ali slej skozi medpomnilnik okvirjev. Več o tem v nadaljevanju.

algebri (matričnih operacijah), je bilo še vedno prepuščeno procesorju. Tako so se začele pojavljati prve grafične procesne enote (GPE, angl. GPU).

Grafične procesne enote sestojijo iz enakih komponent, kot centralne procesne enote: oboje imajo aritmetično-logične enote, nadzorne enote in več nivojev predpomnilnikov. [42]

Razlikujeta pa se v številu in zmogljivosti posameznih komponent, in nenazadnje seveda v samem namenu. CPE je splošnonamenski – s serijskim izvajanjem lahko storí vse, a morda nekoliko počasneje (npr. množenje matrik). GPE je namenjen predvsem za linearno algebro in paralelno računanje. Sestavljen je iz velikega števila jader, ki računajo vzporedno druga z drugo. Hitrost grafičnih procesnih enot merimo v številu operacij v plavajoči vejici na sekundo (FLOPS, floating point operations per second), dandanes navadno v teraflopsih.



Slika 2: Generična shema grafične procesne enote (GPU) [30].

2.3 Grafične kartice

GPE je lahko integriran (na istem vezju kot CPE), ali pa je na voljo kot samostojna naprava, ki ji pravimo grafična kartica. Integriran GPE si deli pomnilnik s CPE-jem, grafična kartica pa vsebuje svoj pomnilnik, imenovan video pomnilnik (VRAM), ki je namenjen izključno GPE in se po zasnovi nekoliko razlikuje od SDRAM-a (pogosto je to SGRAM oz. GDDR SDRAM, Graphics double data rate synchronous dynamic random-access memory). Na grafični kartici

najdemo še vmesnik, na katerega lahko priključimo zaslon – ta je bil doslej povezan bodisi z MO-jem ali pa matično ploščo.

GPE na VRAM-u shranjuje različne elemente grafičnega upodabljanja (npr. teksture) [54] in pa medpomnilnik okvirjev ter FIFO vrsta ukazov. MO pošilja okvirje na vmesnik za zaslon, uporabniške aplikacije pa izstavljajo ukaze na FIFO vrsto in konfigurirajo napravo.

Na tej točki v zgodovini je še vedno obstajal problem neposrednega dostopa do grafične strojne opreme s strani programov. Lahko se je zgodilo, da v MO piše več aplikacij naenkrat; enako velja za konfiguracijo naprave [48]. Nalogo reševanja tega problema nosi jedro operacijskega sistema.



Slika 3: Grafična kartica Sapphire Radeon HD 5570 [9]

3 *Primer: uporaba virtualnega medpomnilnika okvirjev v Linuxu*

Ker je MO zastarel in ni več v uporabi, ga dandanes na sistemih Linux ne uporabljamo več na enak način (tj. vsaj ne z napravo `fbdev` [29]) [10].

Klub temu obstaja abstrakcija medpomnilnika okvirjev, oziroma kar virtualni³ medpomnilnik okvirjev (VMO), ki je še vedno uporabljen v današnjih Linux sistemih. Predstavlja MO neke grafične naprave (npr. GPE) [23]. S perspektive programerja gre za navadno datoteko oz napravo `/dev/fb*`, kjer znak * ponazarja številko VMO, če imamo poleg vgrajene strojen opreme še grafično kartico [23]. Tega lahko tudi kot programerji neposredno uporabimo. Poglejmo

³”Virtualni”, ker programi ne razlikujejo, ali gre za bodisi fizični MO, bodisi za hrambo okvirjev nekje v pomnilniku računalnika, bodisi za hrambo okvirjev nekje v video pomnilniku (VRAM-u) grafične kartice.

si primer na omenjenem operacijskem sistemu. Tukaj delo upodabljanja opravi CPE; z drugo besedo – `/dev/fb0` ne podpira strojnega pospeševanja oziroma uporabe GPE [23]. Predpogoj za uporabo `fb0` je, da se iz grafičnega vmesnika prestavimo v enega izmed teleprinterjev (teletypewriter)⁴. Namizno okolje (v mojem primeru se nahaja na `tty7`) namreč ne uporablja naprav `fb*` neposredno – edini dostop do njih ima uporavljač oken, zaradi česar sprememb ni mogoče videti.

Po opravljenem preklopu v na primer `tty2` lahko pričnemo z neposrednim pišanjem v medpomnilnik okvirjev. Privzeli bomo, da je to `/dev/fb0`

Najpreprostejši primer uporablja le lupino; poženemo sledeči ukaz:

```
cp /dev/random /dev/fb0
```

`/dev/random` sproti generira naključne vrednosti, zato se bo ukaz prej ali slej zaključil z napako, da je na napravi zmanjaklo prostora (popisali smo celoten ekran).

Poizkusimo izrisati 2^{20} pikslov modro-zelene barve (RGB: 0, 255, 255) v medpomnilnik okvirjev z uporabo sistemskoga klica `write()`. Prevedemo in poženemo naslednjo kodo: [32, predelana koda]

```

1 #include <fcntl.h>
2 #include <linux/fb.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <sys/ioctl.h>
6 #include <unistd.h>
7
8 int main() {
9     int fbfd = 0;
10    struct fb_var_screeninfo vinfo; // spremenljive lastnosti
11    struct fb_fix_screeninfo finfo; // nespremenljive lastnosti
12
13    fbfd = open("/dev/fb0", O_RDWR);
14    if (fbfd == -1) exit(1);
15    if (ioctl(fbfd, FBIOGET_FSCREENINFO, &finfo) == -1) exit(2);
16    if (ioctl(fbfd, FBIOGET_VSCREENINFO, &vinfo) == -1) exit(3);
17
18    printf("%dx%d, %dbpp\n", vinfo.xres, vinfo.yres,
19           vinfo.bits_per_pixel);
20
21    // assume 32bpp
22    // V pomnilniku: B, G, R, prosojnost.
23    unsigned char pixel[4] = {255, 255, 0, 0};
24    int write_size = 4;
```

⁴Prevod pridobljen 13. 7. 2024 s portala PONS (sl.pons.com). Verjetno bi bil ustrezен prevod tudi "teletipkalnik".

```

25     if (vinfo.bits_per_pixel != 32) { // assume 16bpp
26         // V pomnilniku: [B, G], [R, prosojnost].
27         pixel[1] = 0; // {255, 0}
28         int write_size = 2;
29     }
30
31     // izpišimo 2^20 pikslov modro-zelene barve (0, 255, 255)
32     for (int i = 0; i < 1<<20; i++) {
33         write(fbfd, pixel, write_size);
34     }
35
36     printf("Press enter to exit...\n");
37     getchar(); // Čakamo na izhod.
38     close(fbfd);
39     return 0;
40 }
```

Sedaj lahko poskusimo še nekoliko kompleksnejši primer, kjer izrišemo kvdarat. Preprosto prevedemo in poženemo naslednjo kodo: [32]

```

1 #include <fcntl.h>
2 #include <linux/fb.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <sys/ioctl.h>
6 #include <sys/mman.h>
7 #include <unistd.h>
8
9 void get_screeninfo(int *fbfd, struct fb_var_screeninfo *vinfo,
10                     struct fb_fix_screeninfo *finfo) {
11     // Odpre datoteko, ki predstavlja virtualni MO v načinu
12     // read/write
13     *fbfd = open("/dev/fb0", O_RDWR);
14
15     if (*fbfd == -1) {
16         perror("Error: cannot open framebuffer device");
17         exit(1);
18     }
19     printf("The framebuffer device was opened "
20           "successfully.\n");
21
22     // Get fixed screen information
23     if (ioctl(*fbfd, FBIOWGET_FSCREENINFO, finfo) == -1) {
24         perror("Error reading fixed information");
25         exit(2);
26     }
27
28     // Get variable screen information
```

```

29     if (ioctl(*fbfd, FBIOGET_VSCREENINFO, vinfo) == -1) {
30         perror("Error reading variable information");
31         exit(3);
32     }
33 }
34
35 int main() {
36     int fbfd = 0;
37     struct fb_var_screeninfo vinfo; // spremenljive lastnosti #to/do
38     struct fb_fix_screeninfo finfo; // nespremenljive lastnosti
39     long int screensize = 0, location = 0;
40     unsigned char *fbp = 0;
41     int x = 0,
42         y = 0; // Where we are going to put the pixel
43
44     get_screeninfo(&fbfd, &vinfo, &finfo);
45
46     printf("%dx%d, %dbpp\n", vinfo.xres, vinfo.yres,
47            vinfo.bits_per_pixel);
48
49     // Figure out the size of the screen in bytes (that's
50     // why we divide by 8).
51     screensize = vinfo.xres * vinfo.yres * vinfo.bits_per_pixel / 8;
52
53     // Map the device to memory
54     fbp = (unsigned char *) mmap(
55         0, screensize, PROT_READ | PROT_WRITE, MAP_SHARED, fbfd, 0);
56
57     if ((int) fbp == -1) {
58         perror("Error: failed to map framebuffer device to "
59                 "memory");
60         exit(4);
61     }
62     printf("The framebuffer device was mapped to memory "
63           "successfully.\n");
64
65     int x_border = 300, y_border = 100;
66
67     // Figure out where in memory to put the pixel
68     for (y = 150; y < y_border; y++) {
69         for (x = 150; x < x_border; x++) {
70             location =
71                 (x + vinfo.xoffset) * (vinfo.bits_per_pixel / 8) +
72                 (y + vinfo.yoffset) * finfo.line_length;
73
74             if (vinfo.bits_per_pixel == 32) {
75                 *(fbp + location) = 100; // blue
76                 *(fbp + location + 1) = 15+(x-100)/2; // green
77                 *(fbp + location + 2) = 200-(y-100)/5; // red

```

```

78             *(fbp + location + 3) = 0;           // No
79             // transparency
80         } else {                         // assume 16bpp
81             int b = 10;
82             int g = (x-100)/6;           // green
83             int r = 31-(y-100)/16;       // red
84
85             unsigned short int t = r << 11 | g << 5 | b;
86             *((unsigned short int *) (fbp + location)) = t;
87         }
88     }
89
90     printf("Press enter to exit...\n");
91     getchar();    // Čakamo na izhod.
92     munmap(fbp, screensize);
93     close(fbfd);
94     return 0;
95 }
```

Strukturi `fb_fix_screeninfo` in `fb_var_screeninfo` [24] nosita fiksne in spremenljive informacije o zaslomu. Fiksne morda niso tako zanimive s perspektive programiranja grafike (so pa morda zanimive s perspektive razvoja operacijskih sistemov⁵), razen polja `line_length`, ki nosi število pikslov v eni vrstici. Druga struktura je veliko bolj zanimiva:

```

1 struct fb_var_screeninfo {
2     __u32 xres;                  // visible resolution
3     __u32 yres;                  // virtual resolution
4     __u32 xres_virtual;          // virtual resolution
5     __u32 yres_virtual;          // visible resolution
6     __u32 xoffset;               // offset from virtual to
7     __u32 yoffset;               // visible resolution
8
9     __u32 bits_per_pixel;        // guess what
10    __u32 grayscale;             // 0 = color, 1 = grayscale,
11        // >1 = FOURCC
12    struct fb_bitfield red;      // bitfield in fb mem if
13    struct fb_bitfield green;    // true color, else only
14    struct fb_bitfield blue;     // length is significant
15    struct fb_bitfield transp;   // transparency
16
17    __u32 nonstd;                // != 0 Non standard pixel format
18
19    __u32 activate;              // see FB_ACTIVATE_*
```

⁵V fiksnih na primer najdemo polji `unsigned long mmio_start` in `__u32 mmio_len` – prvo hrani začetek pomnilniško preslikanega V/I, slednje pa njegovo dolžino.

```

20     __u32 height;           // height of picture in mm
21     __u32 width;           // width of picture in mm
22
23     __u32 accel_flags;      // (OBSOLETE) see fb_info.flags
24
25
26     // Timing: All values in pixclocks, except pixclock (of course)
27     __u32 pixclock;          // pixel clock in ps (pico seconds)
28     __u32 left_margin;       // time from sync to picture
29     __u32 right_margin;      // time from picture to sync
30     __u32 upper_margin;      // time from sync to picture
31     __u32 lower_margin;      // time from sync to picture
32     __u32 hsync_len;          // length of horizontal sync
33     __u32 vsync_len;          // length of vertical sync
34     __u32 sync;               // see FB_SYNC_*
35     __u32 vmode;              // see FB_VMODE_*
36     __u32 rotate;             // angle we rotate counter clockwise
37     __u32 colorspace;         // colorspace for FOURCC-based modes
38     __u32 reserved[4];        // Reserved for future compatibility
39 };

```

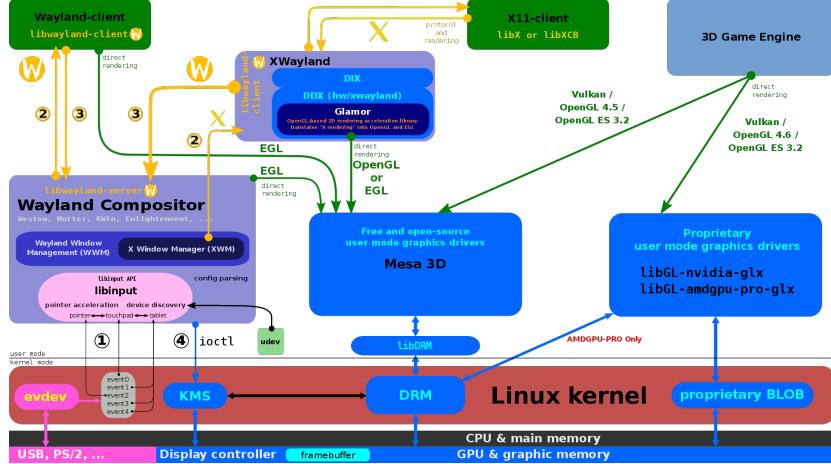
Telo funkcije `main()` torej pridobi fiksne in spremenljive podatke. Slednji vsebujejo tudi `x` in `y` resolucijo zaslona, odmik na `x` in `y` ter število bitov na piksel. Iz tega lahko izračunamo velikost enega okvirja zaslona v bajtih in določimo pomnilniški naslov (v VMO!), kamor bomo v i -ti iteraciji postavili i -ti piksel. Seveda moramo prej VMO še preslikati v pomnilniški prostor CPE, kar storimo s sistemskim klicem `mmap`. Sedaj lahko VMO tretiramo kot binarno datoteko – podobno, kot bi tretirali BMP datoteko. V dvojni for zanki se pomikamo piksel za pikslom in vsakič sprememimo količino rdeče, zelene in modre barve (ter prosojnost). Na koncu le čakamo na vnos nekega znaka, da se nam program ne zapre prehitro.

Če si ogledamo še funkcijo `get_screeninfo()`, ki pridobi podatke, lahko opazimo, da to stori prek sistemskih klicev `ioctl()`. Ti bodo ponovno prišli v upoštev v poglavju DRM in DRI. Za argumente vzamejo datotečni deskrptor, strojno-odvisen ukaz in dodatne argumente (v tem primeru strukturo, kamor se zapišejo pridobljeni podatki.)

4 Grafični sklad operacijskega sistema Linux

Jedro Linux operacijskega sistema nima orodij, ki bi omogočala izris kompleksnih grafičnih elementov na zaslonu, kot so okna, animacije in podobno. To pomeni, da se implementacija grafike razlikuje od distribucije do distribucije. V mnogih, kot so Ubuntu, Mint, openSUSE in Fedora, je določena implementacija grafičnega sklada že vgrajena v samo distribucijo, v lažjih ("lightweight") distribucijah, kot sta Arch in Gentoo, pa mora uporabnik sam dodati gonilnike,

upravljalca oken, kompozitor in druge gradnike.



Slika 4: Grafični sklad Linux operacijskega sistema z arhitekturo okenskega sistema Wayland, ki z uporabo XWayland emulira delovanje okenskega sistema X. [36]

Najpogostejše komponente (nekatere so neizogibne) operacijskega sistema Linux so: [1]

1. jedro in gonilniki naprav;
2. grafični API (OpenGL in konkretna implementacija, npr. Mesa);
3. upravljalec neposrednega upodabljanja (DRM) in infrastruktura neposrednega upodabljanja (DRI);
4. okenski sistem (ponavadi X) ali pa zgolj prikazni strežnik (arhitektura Wayland);
5. kompozitor;
6. Ostale komponente in orodja za razvoj (npr. Qt)

4.1 Linux jedro in gonilniki naprav

Ko govorimo o operacijskih sistemih Linux, najprej naletimo na jedro, saj je to najosnovnejši in najpomembnejši del vsake Linux distribucije in skrbi za delovanje računalnika nasploh.

Linux jedro vsebuje vse od gonilnikov (npr. grafičnih kartic), do podsistemov, ki uporabljajo te gonilnike in dostopajo do strojne opreme (kot je DRI – infrastruktura za neposredno upodabljanje). Linux jedro (z vsebovanimi gonilniki)

tako predstavlja gonilniško plast, ta pa skupaj s strojno plastjo (katero tvori strojna oprema) tvori prikazno plast.

Na Linuxovem grafičnem skladu obstaja veliko prepletanja med njegovimi plastmi, zato je dvema pomembnejšima deloma grafičnih gonilnikov – grafičnemu API-ju in DRI-ju – namenjeno posebno poglavje.

Grafični API je namreč po eni strani namenjen uporabniku in z njim delamo na višjem nivoju, po drugi strani pa mora že sam izdelovalec grafičnih kartic (npr. Nvidia) na nivoju elektrotehnike GPE zasnovati tako, da bo omogočal vse, kar zapoveduje standard.

Če želimo spisati svoj odprtoden gonilnik s podporo OpenGL-a, kot je Nouveau za Nvidia grafične kartice, lahko uporabimo arhitekturo Gallium3D. Gre za množico vmesnikov in zbirko knjižnic, s katerimi lahko spišemo gonilnik na (večinoma) strojno-neodvisen način. Projekt Nouveau je primer zbirke odprtodnih gonilnikov za Nvidia grafične kartice izdelanih s pomočjo Gallium3D API-ja. To pomeni, da so vsi Nouveau gonilniki izdelani po OpenGL standardu. Nouveau je del projekta Mesa.

Poleg grafičnega API-ja, ki sega v upodobitveno plast, obstaja tudi infrastruktura neposrednega upodabljanja (DRI), ki je tesno povezana z okenskim sistemom X in tako povezuje gonilniško plast in interoperacijske plasti. O tem več v poglavju, namenjenemu infrastrukturi DRI in podsistemu DRM.

4.2 Grafični API in OpenGL

Grafični API je tisti del grafičnega sklada, ki nastopi takoj za prikazno plastjo. Namenjen je programerju, da ga uporabi za izrisovanje kompleksnejših grafičnih elementov na zaslonu (npr. trikotniki).

OpenGL (in njegova ”nadgradnja”Vulkan), je standard oz. specifikacija [19] grafičnega API-ja. V prvi vrsti namenjen izdelovalcem grafičnih kartic, saj po teh navodilih izdelajo gonilnike za svojo opremo, za katero nato rečemo, da je kompatibilna z OpenGL standardom. Na tej točki še ne govorimo o (programskej) implementaciji OpenGL-a – ta se pojavi v obliki gonilnika grafičnega procesorja (oz. grafične kartice).

Če povzamemo: niti OpenGL-a niti Vulkana ne moremo opredeliti kot ”knjižnjici”, temveč kot šeznam navodil”, katerim zadošča več gonilnikov in implementacij⁶ za odjemalca⁷.

⁶Podobno, kot je v kontekstu podatkovnih baz SQL le specifikacija jezika, obstaja pa več implementacij

⁷Arhitektura okenskega sistema X (ki je najbolj pogost sistem na operacijskih sistemih Linux), tudi klient; Prevod pridobljen 13. 7. 2024 s portala PONS (sl.pons.com).

4.2.1 Mesa in grafični gonilniki

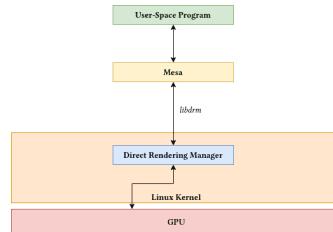
Mesa (oz. Mesa3D) [25] je ena izmed bolj poznanih implementacij OpenGL (in Vulkan) standarda za odjemalca. Gre za dejansko grafično knjižnico, ki nastopa v obliki gonilnika. To pomeni, da projekt Mesa:

1. nudi gonilnik za neko specifično grafično kartico (primer: že omenjeni Nouveau za Nvidia grafične kartice);
2. nudi implementacijo knjižnice `libdrm`, ta pa nudi ovojne funkcije sistemskih klicev `ioctl`, s katerimi komuniciramo z grafično kartico.
3. nudi implementacijo specifikacije OpenGL, torej vse funkcije, ki jih predpostavlja specifikacija tega API-ja.

Gonilnik grafične kartice je torej zadolžen za prevedbo visokonivojskih ukazov, ki so na voljo programerju, v strojno kodo, razumljivo grafičnemu procesorju. Do funkcionalnosti, ki jih omogočata grafična kartica in njen gonilnik, lahko dostopamo samo preko jedra, in sicer prek pod sistema DRM, ki ga bomo spoznali v sledečem poglavju. Projekt Mesa nudi knjižnico `libdrm` – ta je del podistema Linux jedra, imenovanega DRM (Direct Rendering Manager, upravljalec neposrednega upodabljanja)⁸. Sam razvoj podistema DRM iz zgodovinskih razlogov v okrilju projekta Mesa [41], kar je razlog za tako povezavo med jedrom in omenjeno knjižnico.

Kako pa deluje knjižnica, ki implementira specifikacijo OpenGL? Deklaracije funkcij, ki jih predpisuje API, so zapisane v zagavnih datotekah (npr. `gl.h`, `egl.h`, `eglext.h` itd.⁹). Pri prevajanju naših programov moramo povezati ustrezno deljeno knjižnico (npr. za zaglavno datoteko `gl.h` moramo pri prevajanju z `gcc` uporabiti zastavico `-fPIC`, da povežemo deljeno knjižnico `libgl.so`). S tem dejansko uporabimo implementacijo OpenGL.

V isto kategorijo kot Mesa spada tudi Direct3D, na katerega naletimo na Windows operacijskih sistemih. Omenimo še, da zaprtokodni gonilniki, ki jih proizvajalec za neko grafično kartico zamenjajo celotno hierarhijo od samega gonilnika do implementacije standarda.



Slika 5: Shema odnosa implementacije Mesa (OpenGL API-ja) z nižjimi in višjimi nivoji. [1]

⁸Glej poglavje *DRM in DRI*

⁹Podrobnosti strukture knjižnice OpenGL ni predmet te naloge.

4.2.2 Povezava med upravljalci oken in OpenGL

Standard OpenGL je namenjen upodabljanju 2D in 3D grafike in tako ne nudi funkcij za stvarjenje oken. Kljub temu predpostavlja, da na sistemu deluje nekakšen upravitelj oken – v Linuxu je to povečini X. Zato moramo uporabiti še GLX podaljšek (GLX extension) okenskega sistema X, s katerim lahko posamezno OpenGL sceno vežemo na neko X okno in tako omogočimo X odjemalcem, da rišejo na vzpostavljena okna [44]. Na Windows sistemih lahko namesto GLX uporabimo vmesnik WGL.

Ker odvisnost od upravljalca oken oteži programiranje prenosljivih aplikacij, so avtorji standarda OpenGL zasnovali še vmesnik EGL, ki povezuje poljubnega upravljalca oken z upodobitvenim API-jem. EGL deluje tudi, če upravljalca oken ni.

4.2.3 Cairo in Pixman

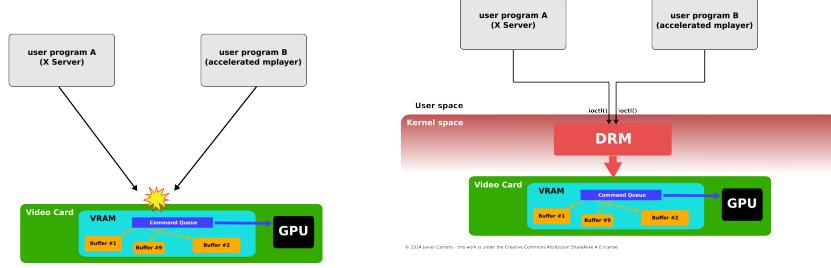
Omenimo še knjižnico Cairo. Skrbi izključno za izrisovanje vektorske grafike, v grafičnem skladu pa spada podoben nivo kot Mesa implementacija. Cairo uporablja različna zaledja (angl. *backends*), ki jih imenuje površine (angl. *surfaces*). Mednje spadajo Xlib, OpenGL, slikovne datoteke in SVG datoteke [2]. Če površina podpira strojno pospeševanje, ga uporabi [3]. Če Cairo za svoje zaledje torej uporablja OpenGL ali Xlib je torej upodabljanje strojno pospešeno; upodabljanje direktno na sliko, pa ni [4].

Pri izrisovanju na platna okenskega sistema X s knjižnico Xlib zaradi narave obeh knjižnic pri takem programiranju nastane veliko ponavljajoče se kode. Da bi rešili ta problem, je bila razvita deljena knjižnica Pixman, ki povezuje X in Cairo na nekoliko bolj uporabniku prijazen način. Med drugim Pixman nudi rasterizacijske algoritme in podporo za gradiente.

Cairo knjižnice večinoma ne uporabljamo neposredno, temveč prek orodij, kot je GTK+. Če je naš projekt enostavne narave, pa lahko za ozadje knjižnice uporabimo OpenGL.

4.3 DRM in DRI

Infrastruktura neposrednega upodabljanja (DRI, Direct Rendering Infrastructure) [40] je ogrodje, ki uporabniški programske opremi brez privilegijev omogoča izstavljanje ukazov grafični strojni opremi brez konfliktov. Če nekoliko poenostavimo: DRI razreši problem hkratnega pisanja programov v medpomnilnik okvirjev tako, da vsakemu dovoli alokacijo dela video pomnilnika na nadzorovan način.



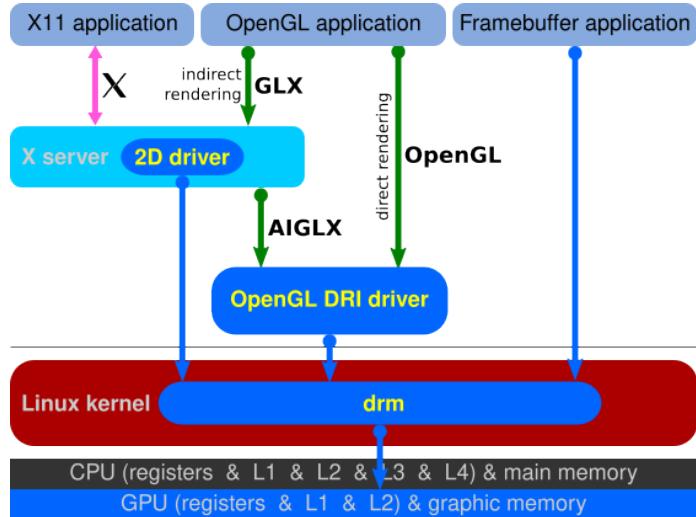
(b) Uporaba DRM odpravi problem hkratnega dostopa večih programov do video pomnilnika. [5]

(a) Konflikti pri dostopanju večih programov do video pomnilnika. [6]

Ime ogrodja izvira iz časov, ko je imel okenski sistem X edini pravico dostopanja do grafične strojne opreme (ta privilegij velja še danes), da bi se izognili konfliktom pri hkratnem dostopanju do nje. Tako so morali programi za izrisovanje najprej zahtevo predati X strežniku, od tod pa je bila zahteva posredovana grafični strojni opremi – uporabljanje je bilo *posredno*. Ob pojavu računskozahtevne 3D grafike je tovrstni model medprocesne komunikacije postal mnogo prepočasen. Iz tega razloga je bil razvit DRI, ki je omogočal *neposredno* upodabljanje, tj. predajo ukazov grafični strojni opremi brez komunikacije z okenskim sistemom. Obstajajo tri različice DRI (DRI1, DRI2 in DRI3) – sprva je vsak program pridobil edinstven nadzor nad grafično strojno opremo, z DRI2 pa jim je bila dana možnost alokacije video pomnilnika za upodabljanje. [40]

Infrastruktura DRI je torej tesno povezana z okenskim sistemom X. Sestavlja jo tri glavne komponente:

1. **DRI odjemalec** je vsak program, ki od grafične strojne opreme zahteva storitve. Za delovanje potrebuje DRI gonilnike, ki navadno nastopajo v obliki deljenih knjižnic.
2. **Okenski sistem X** nudi protokol X11, ki ga DRI odjemalci uporabijo za komunikacijo z okenskim sistemom – z strojno-neodvisnim gonilnikom X – in z strojno-odvisnim gonilnikom X. Slednji komunicira neposredno z grafično strojno opremo, zaradi česar se ga pogosto imenuje kar ”video gonilnik” oz. ”grafični gonilnik”.
3. **DRM (Upravljalec neposrednega upodabljanja)** je podsistem Linux jedra, ki nudi vmesnik GPE-jev in tako nadzira dostop do njih.



Slika 7: Shema infrastrukture neposrednega upodabljanja. [35]

Morda je nekoliko manj očitno, da je vsak X odjemalec lahko tudi DRI odjemalec, saj lahko na primer zahteva upodabljanje od GPE, ta pa vrne upodobljen okvir¹⁰. Temu postopku pravimo upodabljanje izven zaslona (off-screen rendering; GPE izvede upodabljanje, a ga ne izriše na zaslon).

Bolj očiten primer DRI odjemalca je X strežnik. Ta mora seveda uporabljati infrastrukturo, da lahko grafičnemu procesorju sporoči, kje se nahajajo posamezna okna. [40]

Sedaj pa lahko razumemo učinkovitost infrastrukture neposrednega upodabljanja: X strežnik bo za vsak program, ki želi risati na zaslon, alociral kos video pomnilnika. Sedaj okvirja, ki je nastal pri upodabljanju zunaj zaslona, DRI odjemalcu ni potrebnopošiljati še X strežniku, da ga ta pošlje v video pomnilnik, temveč lahko programi povsem samostojno pišejo v svoj kos video pomnilnika, ta pa se izriše na zaslon – tako se izognemo nepotrebni medprocesni komunikaciji. Če uporabnik spremeni velikost posameznega okna, bo X to preprosto sporočil ustreznemu odjemalcu in na novo alociral kos video pomnilnika. [40]

Če kljub temu okvir najprej pošljemo X strežniku, pa lahko nad okvirjem izvedemo post-procesiranje.

4.3.1 DRM

Upravljalec neposrednega upodabljanja (DRM, Direct Rendering Manager) [41] je podsistem Linux jedra, ki nudi vmesnik GPE-jev – ti se v tem kontekstu imenujejo *DRM naprave*. Vsako zaznano DRM napravo upravljalec razkrije

¹⁰Okvir si lahko predstavljamo kot polje pikslov

programom v obliki datotek¹¹ `/dev/dri/cardX`, kjer je X zaporedna številka GPE. DRM uporablja sistemski kljice `ioctl`, da alocira pomnilnik in nastavi parametre GPE. Sistemski kljici `ioctl` so namenjeni za vhodno-izhodne operacije za različne naprave – v tem primeru za GPE.

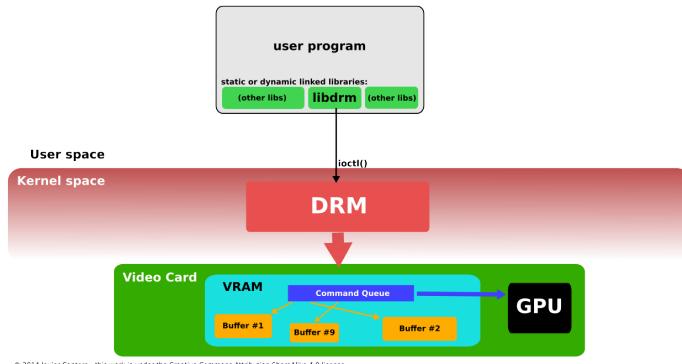
Definicija tega sistemskega kljica je sledeča:

```
int ioctl(int fd, unsigned long request, ...) [22]
```

kjer so:

1. `fd` odprt datotečni deskriptor naprave, ki jo naslavljamo;
2. `request` ukaz, poslan napravi;
3. ... kazalec na nek kos pomnilnika, ki je iz zgodovinskih razlogov ponavadi `char *argp`.

Da programerju ni potrebno ”na pamet” poznati ukazov, ki jih lahko pošlje grafičnim karticam (sploh, ker se ukazi od ene do druge kartice razlikujejo), nam projekt Mesa nudi knjižnico `libdrm` [26, poglavje *Software architecture*], ki se stoji iz ovojnih funkcij posameznih ukazov za grafično kartico [41]. Knjižnica ovija kljice `ioctl` tako gonilnika Nouveau kot DRM [13].



Slika 8: Shema podsistema DRM. [7]

DRM kot podsistem pa je sestavljen iz večih komponent; na kratko: [41]

1. **KMS (Kernel Mode Setting)** je komponenta, ki nam omogoča, da preko `libdrm` knjižnice neposredno konfiguriramo strojno opremo prek `ioctl` sistemskih kljicev (angl. *mode-setting*). KMS preprečuje, da bi programi sami neposredno nastavljali način delovanja grafične strojne opreme – dandanes je X strežnik edini, ki ima pravice za uporabo KMS (seveda ob predpostavki, da sistem uporablja X sistem oken).

¹¹princip *Everything is a file*

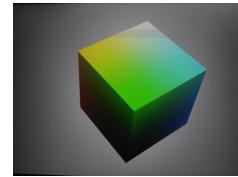
2. **GEM (Graphics Execution Manager)** je komponenta, ki programom nudi API za upravljanje z video pomnilnikom. Možno je ustvariti, rokovati in uničiti ti. GEM objekte (ti predstavljajo podatke, ki jih potrebuje GPU – npr. tekture). Programi si lahko isti GEM objekt med seboj delijo.
3. **Upodobitvena vozlišča (Render nodes)** so posledica arhitekture pravic podsistema DRM. Ker v privilegiranem načinu delujejo le programi, ki nadzirajo izris (X strežnik, Wayland kompozitor...), mora vsak neprivilegiran program za uporabo GPE (pa čeprav ni v grafične namene) zaprositi privilegirani program. Upodobitveno vozlišče poleg datoteke `/dev/dri/cardX` nudi še datoteko `/dev/dri/renderDY` (številki X in Y nista nujno enaki¹²), preko katere lahko vsak neprivilegiran program z nekoliko omejenim naborom ukazov prek sistema DRM izvaja računske operacije na GPE.

4.4 *Primer: uporaba DRM, KMS, GBM in OpenGL za strojno-pospešen grafični prikaz*

Z uporabo podsistema DRM in njegovih komponent KMS ter GBM (del projekta Mesa) je moč izrisovati na zaslon s strojnimi pospeševanjem. Poleg tega lahko uporabimo še funkcionalnosti, ki jih ponuja OpenGL in tako zagotovimo nekoliko zanimivejši rezultat.

Primer takega programa je na voljo na repozitoriju projekta Mesa in nosi naslov `kmscube` (gitlab.freedesktop.org/mesa/kmscube)^[27]. Zagon tega programa je enak kot v prejšnjem primeru – prestavimo se v teleprinter `tty*`, prevedemo projekt z uporabo orodij Meson in Ninja, ter poženemo prevedeno datoteko. Program je precej kompleksen zaradi svoje nizkonivojske narave, zaradi česar se vanj ne bomo spuščali. Omenimo pa nekaj osnovnih lastnosti programa:

1. Program uporablja knjižnico EGL [18], ki omogoča, da OpenGL uporabimo brez upravljalca oken, torej z neposrednim nadzorom nad medpomnilnikom okvirjev.
2. S podsistom KMS nastavimo delovanje grafične kartice.
3. S podsistom GBM [46], ki je abstrakcija funkcij za upravljanje s pomnilnikom grafične kartice, alociramo medpomnilnik za grafično upodabljanje.
4. OpenGL funkcionalnosti se kažejo predvsem v kompleksnem objektu, ki ga izrišemo – 3D kocka.



Slika 9: Upodobitev aplikacije `kmscube`. Lastno delo.

¹²Na sistemu avtorja obstajata `/dev/dri/card0` in `/dev/dri/renderD128`.

4.5 Okenski sistemi

Okenski sistem je del programske opreme, ki služi kot temelj grafičnega uporabniškega vmesnika tako, da realizira paradigmo WIMP [51, 31] – okna, ikone, meniji, kazalci (**windows, icons, menus, pointer**). Gre za t.i. grafične nadzorne elemente (angl. *graphical control element*) oz. grafične pripomočke (angl. *graphical widget*).

Okenski sistem je sestavljen iz dveh delov, ki pa nista nujno ločena:

1. **Prikazni strežnik** je osrednji del okenskega sistema. Vsak program z GUI-jem je njegov odjemalec. Njegova glavna naloga je nadzor vhodnih in izhodnih podatkov za odjemalce – od njih pridobi zahteve za risanje na zaslon, pošilja pa jim uporabnikove kretnje (podatki s tipkovnice, miške, mikrofona itd.). Prikazni strežnik je tisti, ki komunicira s strojno opremo preko jedra. [51, poglavje *Display server*]
2. **Upravljalec oken** nadzira postavitev in izgled oken na zaslonu. Ne komunicira s strojno opremo, temveč le izdaja navodila za izris prikaznemu strežniku. [50]. V X arhitekturi je upravljalec oken le še en odjemalec prikaznega strežnika [20].

4.5.1 Upravljalec oken

Poznamo več vrst upravljalcev oken: [50]

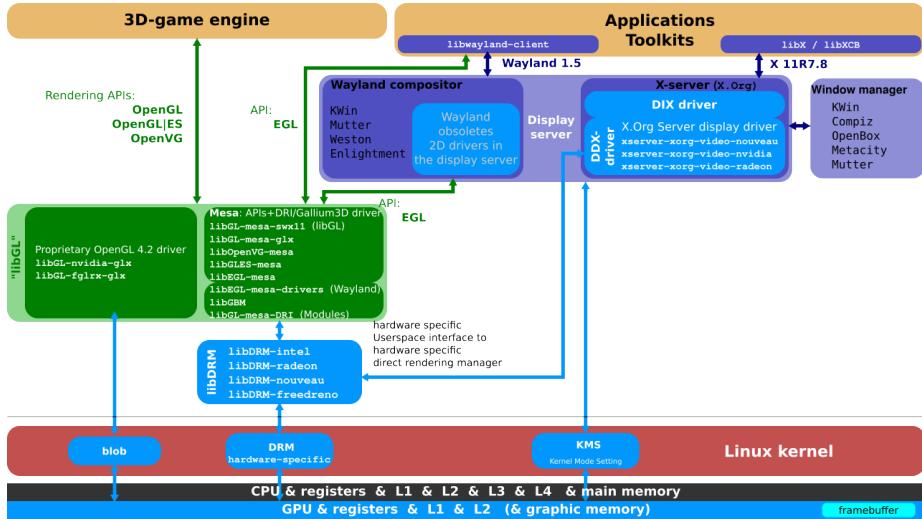
1. **Skladovni upravljalci oken** omogočajo lebdeča okna, ki se lahko prekrivajo. Poznamo različne algoritme (slikarski algoritem, obratni slikarski algoritem) za izrisovanje teh oken.
2. **Ploščični upravljalci oken** okna razporedijo v nekakšno polje, enega ob drugega. Tako ne pride do prekrivanaja.
3. **Dinamični upravljalci oken** lahko dinamično preklapljajo med skladovnim in ploščičnim načinom razporejanja oken.
4. **Kompozicijski upravljalci oken**, znani tudi kot **kompozitorji**, izrišejo vsa okna v izven-zaslonski pomnilnik (angl. *off-screen buffer*). To omogoča post-procesiranje in dodajanje raznih 3D efektov in animacij. [39]
5. **Namizna okolja** so posebna kategorija, saj poleg upravljalca oken nudijo še več orodij, kot so razne teme (angl. *themes*), GUI brskalniki datotek in podobno.

Namig: da ugotovimo, katerega upravljalca okenj uporablja naš sistem, lahko zaženemo ukaz `rmctrl -m`.

4.5.2 Okenski sistem X

Okenski sistem X (na kratko: X) je odprtokoden okenski sistem, najdemo pa ga na večini današnjih linux sistemov. Organiziran je v zvezo odjemalec-strežnik in tako omogoča tudi oddaljen dostop. X je eden izmed najbolj osnovnih gradnikov grafičnih uporabniških vmesnikov, zadolžen pa je tudi za rokovanje z dogodki (event handling) in za olepšave (visual decorations). [52, 20]

X ima za prikazni strežnik omisljeno svojo rešitev: X.org. Strežnik navadno zažene upravljač prikaza (angl. *display manager*)¹³ ali pa ga ročno zaženemo prek ukaznega poziva. [1, 53]



Slika 10: Celoten grafični sklad v Linux operacijskem sistemu z okenskim sistemom X

Za komunikacijo med X strežnikom in X odjemalcem se uporablja protokol X11 [1, 53]. Ta definira izmenjavo sporočil v strežniški arhitekturi sistema X. Če se strežnik in odjemalec nahajata na isti napravi, se sporočila izmenjajo z UNIX vtiči. X11 je razširljiv – nove funkcionalnosti lahko dodajamo brez spreminjanja protokola samega.

Xlib (X library) je knjižnica, namenjena implementaciji na strani odjemalca. Uporabljajo jo druga orodja, kot sta GTK+ in Qt za izdelavo grafičnih vmesnikov za aplikacije.

X.org Foundation dandanes odsvetuje razvoj aplikacij z neposredno uporabo X knjižnic, temveč priporoča različna orodja (dve omenjeni zgoraj). Za nizknivojski razvoj prav tako odsvetujejo uporabo **Xlib** knjižnice [12], priporočajo

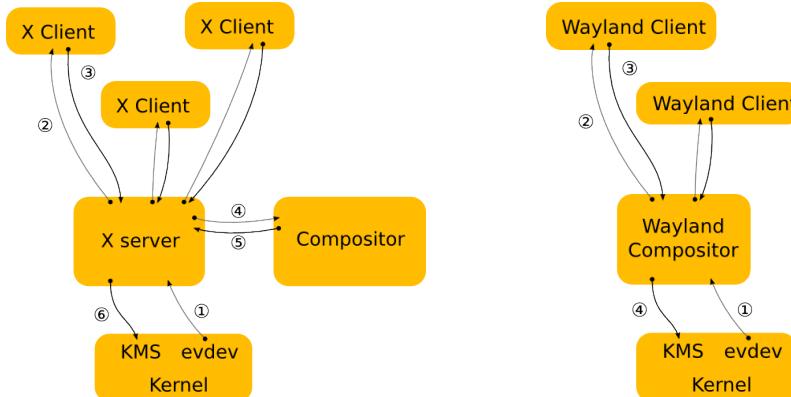
¹³Upravljač prikaza je program, ki omogoča izbiro med različnimi okenskimi sistemi ter nemalokrat nudi tudi prijavno okno.

pa njen "novejšo verzijo" oz. alternativo XCB (X C Binding). Razlogi so manjša velikost, večja odpornost na napake programerja, boljša podpora za multithreading, več razširitev in nižje latence [14].

4.5.3 Wayland

Okenski sistem X je že precej star (prva verzija je zaživila leta 1984) in posledično postaja neroden za uporabo, predvsem zaradi pojavitev novih konceptov, kot so kompozitorji – v arhitekturi sistema X se strežnik sploh ne zaveda lokacije oken na zaslonu in drugih specifik. Odjemalci morajo torej za izrisovanje zahtevno najprej poslati prikaznemu strežniku, ta jo odpošlje kompozitorju, od tod gre obdelana nazaj v strežnik in napisled do strojne opreme. Celoten proces se ponovi v obratnem vrstnem redu, ko uporabnik pritisne na nek grafični pripomoček, kot je gumb.

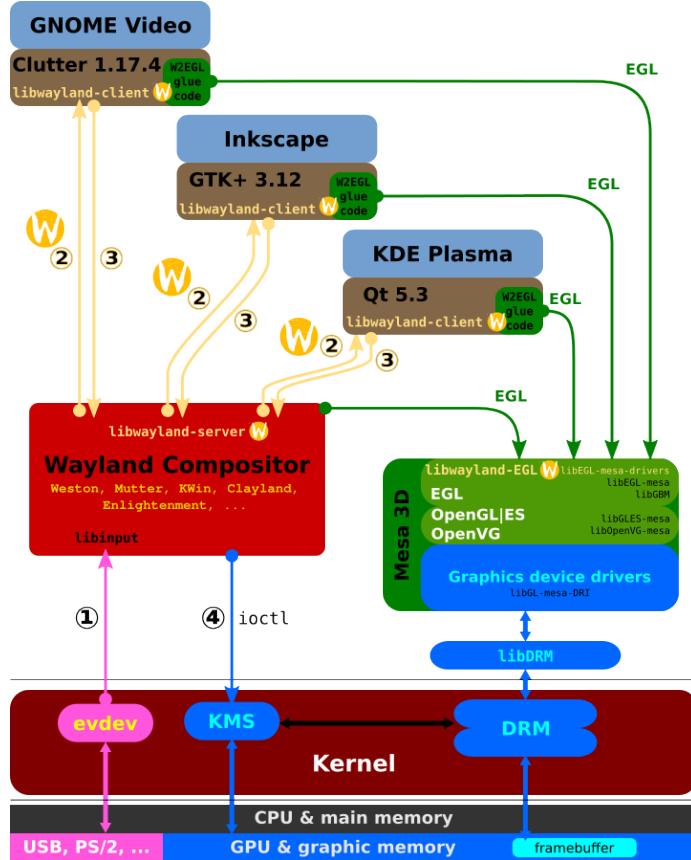
Poleg tega dandanes porvprečen uporabnik ne potrebuje arhitekture z okenskim strežnikom, na katerega bi se povezano več naprav, ki potrebujejo uporabljanje; ta zasnova pa zapovrh ni varna, saj ne kriptira oz. ne ščiti komunikacij med odjemalcem in strežnikom, kar omogoča prisluškovanie in potencialno razkritje zaupnih informacij.



Slika 11: Primerjava X arhitekture (levo) in Wayland arhitekture (desno). [15]

Wayland je naslednik okenskega sistema X. Nadomešča tako protokol X11 s protokolom Wayland, kot arhitekturo. Namesto strežnika postavlja kar kompozitor (ta mora biti kompatibilen z Wayland sistemom), s čimer se izogne nepotrebni medprocesni komunikaciji. Zahteve odjemalcev gredo zdaj zgolj čez kompozitor.

Pomemben gradnik Wayland arhitekture je knjižnica za medprocesno komunikacijo libwayland, ki kodira in dekodira sporočila ter jih pretvarja XML definicije sporočil v API, spisan v jeziku C [16].



Slika 12: Celoten grafični sklad v Linux operacijskem sistemu z Wayland arhitekturo [37]

Kot je že moč razbrati, Wayland ponuja preprosto nadaljno vzdrževanje, razvijanje in razširjevanje. Kljub teoretično odlični arhitekturi je Wayland v času pisanja tega dela v povojih in zaradi številnih pomanjkljivosti, hroščev in napak povzroča številne preglavice, zaradi česar je X še vedno bolj popularna opcija.

Zaradi lažje kompatibilnosti v času, ko še vedno prevlada prikazni strežnik X, lahko uporabimo orodje XWayland, ki pod implementacijo Wayland arhitekture vrine še X strežnik in tako omogoči, da novejši protokoli uporabijo tudi aplikacije, ki niso namenjene zanj.

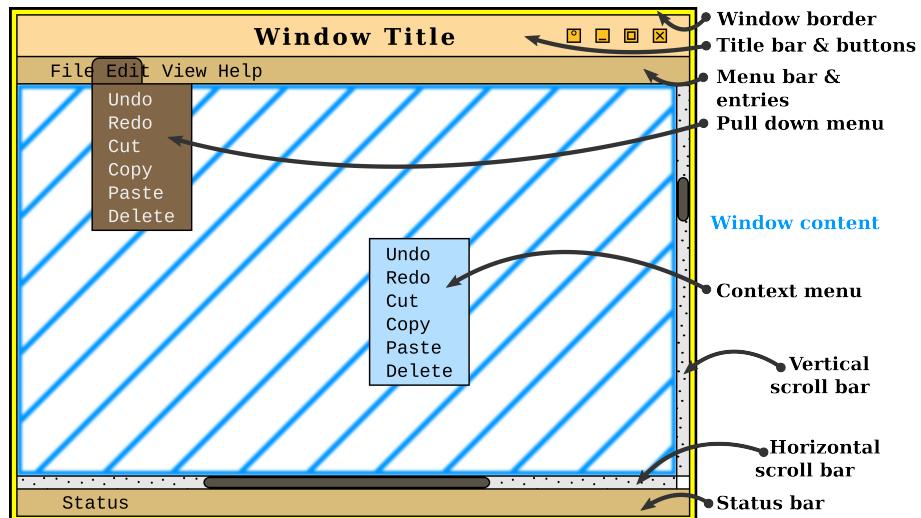
4.6 GUI orodja

GUI orodje (tudi GUI knjižnica) vsebuje vse potrebne funkcionalnosti za izdelavo grafičnih uporabniških vmesnikov (GUI – Graphical User Interface), kot so

rokovalniki dogodkov, prizori (scenes) in grafični pripomočki (angl. *widgets*)¹⁴.

Vsi sestavni elementi paradigmе WIMP [51, 31] (okna, ikone, meniji, kazalci – windows, icons, menus, pointer) predstavljajo grafične pripomočke, ki služijo interakciji med uporabnikom in računalnikom; uporabnik s klikom na gume, premiki oken in izbiranjem v menijih sporoči okenskemu sistemu, da želi določenemu programu poslati določene informacije, ki jih s svojim grafičnim vmesnikom definira program. Če npr. kliknemo na pripomoček za zapiranje oken, bo okenski sistem o tem obvestil ustreznega odjemalca, ta pa se bo na dogodek odzval.

V večini primerov so GUI orodja le ovojne knjižnice nizkonivojskih knjižnic, kot sta Xlib in XCB. Nekateri imajo celo implementirane lastne označevalne jezike, sisteme za dogodke in končne avtomate. Poglejmo si dve izmed bolj popularnih GUI orodij

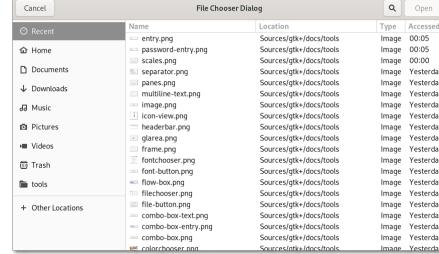


Slika 13: Poglavitni deli okna (meniji), ki so tudi sami grafični pripomočki. [38]

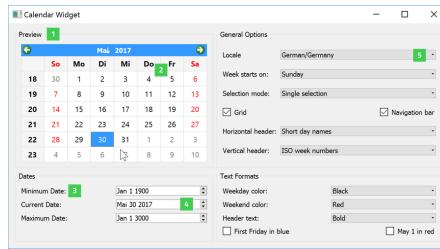
¹⁴Prevod pridobljen 13. 7. 2024 s portala PONS (sl.pons.com). Beseda *widget* izvira iz besede *gadget*, [45, Glej poglavje History] kar v prevodu pomeni pripomoček. Glede na to, da nam "widget-i" pomagajo pri interakciji z grafičnim vmesnimkom, jih lahko smatramo kot pripomoček in je posledično prevod razumen.

4.6.1 GTK+

GIMP Toolkit je GUI orodje za izdelavo večplatformskih aplikacij z uporabniškim grafičnim vmesnikom.



Slika 14: Primer GUI-ja, zgrajenega z orodjem GTK+. [33]



Slika 15: Primer GUI-ja, zgrajenega z orodjem Qt. [8]

4.6.2 Qt

Qt je GUI orodje za izdelavo večplatformskih aplikacij z uporabniškim vmesnikom, ki ponuja knjižnico interaktivnih elementov, za razliko od GTK+ orodja pa lahko za izdelavo vmesnikov uporabimo aplikacijo QtDesigner in integrirano razvojno okolje (IDE) QtCreator.

4.7 Ostali grafični vmesniki

Čeprav je na tej točki že razvidno, da na Linuxu ne obstaja namensko orodje ali knjižnica za izdelavo uporabniških grafičnih vmesnikov, lahko opazimo, da (prej ali slej) vsi grafični ukazi v Linuxovem grafičnem skladu uporabijo OpenGL. To pomeni, ga da lahko kot Linuxovo namensko orodje za grafiko. Primer programa, ki deluje na opisan način, je Blender, ki se ne zanaša na nizkonivojske knjižnice ali orodja, temveč ima svojo knjižnico, popolnoma osnovano na standardu OpenGL [21].

Razvijalec lahko na višjem nivoju uporabi tudi knjižnice, ki so prenosljive med operacijskimi sistemmi in prijazne za uporabo. Prier take knjižnice je GLFW (angl. *Graphics Library Framework*). Je majhna knjižnica za uporabo OpenGL API-ja ter omogoča stvarjenje oken in celo rokovanje z vhodom (npr. miška, tipkovnica in igralna palica) [43, 17].

Zaključek

Računalniška grafika ni enostavno področje. Odkili smo le površino tega, kar je zmožno z današnjo grafično strojno opremo in koncepti, ki jih za sabo prinese samo upodabljanje.

Kljub temu lahko odgovorimo na nekaj vprašanj, ki se lahko porodijo začetniku.

1. **Kako izrisati piksel na zaslon?** Uporabimo medpomnilnik okvirjev, ki se danes večinoma nahaja na grafični kartici. Do njega dostopamo prek mehanizmov Linux jedra, kot je `/dev/fb0`. To ni strojno-pospešen pristop, saj za upodabljanje uporabljamo procesor.
2. **Kako izrisati piksel na zaslon z grafičnim pospeševanjem?** Na Linux sistemih bo potrebno uporabiti OpenGL (ali drug API), DRM in DRI.
3. **Zakaj je treba za vsak programski jezik mukotrpno iskati grafični vmesnik?** V resnici ni potrebno; za prenosljivo aplikacijo bo dovolj uporabiti Qt ali pa GTK+.
4. **Kako naj moj C ali Rust program (ali pa program v kateremkoli jeziku) sporoči računalniku, da želi nekaj narisati na zaslon?** Potrebno je vedeti, kateri okenski sistem uporablja naš računalnik. Nato ga z ustreznimi knjižnicami, ki pridejo z okenskim sistemom, zaprosimo za okno, na katerega lahko nato rišemo. To je prvi korak pri programiranju na računalniku z okenskim sistemom.
5. **Kako sprogramiram svojo GUI knjižnico?** Za tak podvig bo skoraj vedno potrebno uporabiti OpenGL (ali drug API). Glede na to, katero plast bo predstavljalata knjižnica, bo potrebno uporabiti bodisi knjižnico okenskega sistema, bodisi sistemske klice ipd.

Kam naprej?

Vsak, ki želi razumeti matematične postopke upodabljanja, programiranje lastnega upodobitelja ipd. lahko razišče področje **grafičnega cevovoda** (angl. *graphics pipeline*).

Za ne-grafično programiranje GPE sta na voljo ogrodje CUDA podjetja Nvidia (za GPE-je tega podjetja) ali pa OpenCL (za splošno, odprtokodno ogrodje).

Literatura

- [1] Baeldung. *GUI Under Linux*. Opomba: večina poročila je narejena po vzorcu tega vira oz. gre za prevod vira. Nekateri deli so dopolnjeni. URL: <https://www.baeldung.com/linux/gui>. Dostopano: 5. 7. 2024.
- [2] Cairo Graphics. *Backends*. URL: <https://www.cairographics.org/backends/>. Dostopano: 18. 7. 2024.
- [3] Cairo Graphics. *cairographics.org*. URL: <https://www.cairographics.org/>. Dostopano: 18. 7. 2024.
- [4] Cairo Graphics. *Image Surfaces: Cairo: A Vector Graphics Library*. URL: <https://cairographics.org/manual/cairo-Image-Surfaces.html>. Dostopano: 18. 7. 2024.
- [5] Javier Cantero. *Access to video card with DRM.svg*. Licenca: CC BY-SA 4.0 (URL: <https://creativecommons.org/licenses/by-sa/4.0/>) preko Wikimedijine zbirke. URL: https://commons.wikimedia.org/wiki/File:Access_to_video_card_with_DRM.svg. Dostopano: 18. 7. 2024.
- [6] Javier Cantero. *Access to video card without DRM.svg*. Licenca: CC BY-SA 4.0 (URL: <https://creativecommons.org/licenses/by-sa/4.0/>) preko Wikimedijine zbirke. URL: https://commons.wikimedia.org/wiki/File:Access_to_video_card_without_DRM.svg. Dostopano: 18. 7. 2024.
- [7] Javier Cantero. *High level Overview of DRM.svg*. Licenca: CC BY-SA 4.0 (URL: <https://creativecommons.org/licenses/by-sa/4.0/>) preko Wikimedijine zbirke. URL: https://commons.wikimedia.org/wiki/File:High_level_Overview_of_DRM.svg. Dostopano: 18. 7. 2024.
- [8] The Qt Company. *Qt Widget Gallery / Qt Widgets 6.7.2*. URL: <https://doc.qt.io/qt-6/gallery.html>. Dostopano: 18. 7. 2024.
- [9] Evan-Amos. *Sapphire-Radeon-HD-5570-Video-Card.jpg*. Licenca: v javni domeni preko Wikimedijine zbirke. URL: <https://commons.wikimedia.org/wiki/File:Sapphire-Radeon-HD-5570-Video-Card.jpg>. Dostopano: 18. 7. 2024.
- [10] FFmpeg. *FFmpeg Devices Documentation*. Opomba: glej poglavje 3.7 fb-dev. URL: <https://ffmpeg.org/ffmpeg-devices.html#fbdev>. Dostopano: 5. 7. 2024.
- [11] Caroline Ford. *Sun sbus cgsix framebuffer2.jpg*. Licenca: CC BY-SA 3.0 (URL: <https://creativecommons.org/licenses/by-sa/3.0/>) preko Wikimedijine zbirke. URL: https://commons.wikimedia.org/wiki/File:Sun_sbus_cgsix_framebuffer2.jpg. Dostopano: 18. 7. 2024.
- [12] X.org Foundation. *Documentation*. URL: <https://www.x.org/wiki/Documentation/>. Dostopano: 5. 7. 2024.
- [13] freedesktop.org. *InstallNouveau · freedesktop.org*. URL: <https://nouveau.freedesktop.org/InstallNouveau.html>. Dostopano: 18. 7. 2024.
- [14] freedesktop.org. *tutorial*. URL: <https://xcb.freedesktop.org/tutorial/>. Dostopano: 5. 7. 2024.

- [15] freedesktop.org. *Wayland*. URL: {<https://wayland.freedesktop.org/architecture.html>}. Dostopano: 18. 7. 2024.
- [16] freedesktop.org. *Wayland*. URL: {<https://wayland.freedesktop.org/>}. Dostopano: 18. 7. 2024.
- [17] Marcus Geelnard in Camilla Löwy. *An OpenGL library / GLFW*. URL: {<https://www.glfw.org/>}. Dostopano: 18. 7. 2024.
- [18] Khronos Group. *EGL Overview – The Khronos Group Inc.* URL: {<https://www.khronos.org/egl/>}. Dostopano: 13. 7. 2024.
- [19] Khronos Group. *FAQ – OpenGL Wiki*. URL: {<https://www.khronos.org/opengl/wiki/FAQ>}. Dostopano: 5. 7. 2024.
- [20] Chuan Ji. *How X Window Managers Work, And How To Write One (Part I)*. URL: {<https://jichu4n.com/posts/how-x-window-managers-work-and-how-to-write-one-part-i/>}. Dostopano: 18. 7. 2024.
- [21] johnratius, LazyDodo in Developer Forum. *Blender GUI design and implementation – Archive – Developer Forum*. URL: {<https://devtalk.blender.org/t/blender-gui-design-and-implementation/13602>}. Dostopano: 13. 7. 2024.
- [22] Linux Kernel. *ioctl based interfaces — The Linux Kernel documentation*. URL: {<https://docs.kernel.org/driver-api/ioctl.html>}. Dostopano: 13. 7. 2024.
- [23] Linux Kernel. *The Frame Buffer Device*. URL: {<https://www.kernel.org/doc/Documentation/fb/framebuffer.txt>}. Dostopano: 5. 7. 2024.
- [24] Linux Kernel. *The Frame Buffer Device API*. URL: {<https://www.kernel.org/doc/html/latest/fb/api.html>}. Dostopano: 5. 7. 2024.
- [25] Mesa. *Introduction — The Mesa 3D Graphics Library latest documentation*. URL: {<https://docs.mesa3d.org/index.html>}. Dostopano: 5. 7. 2024.
- [26] Mesa. *Mesa / drm* · *GitLab*. URL: {<https://gitlab.freedesktop.org/mesa/drm>}. Dostopano: 5. 7. 2024.
- [27] Mesa. *Mesa / kmscube* · *GitLab*. URL: {<https://gitlab.freedesktop.org/mesa/kmscube>}. Dostopano: 13. 7. 2024.
- [28] OSDev.org. *Graphics stack*. URL: {https://wiki.osdev.org/Graphics_stack}. Dostopano: 5. 7. 2024.
- [29] Forbidden Projects. *How DRI and DRM Work*. URL: {<https://www.bitwiz.org.uk/s/how-dri-and-drm-work.html>}. Dostopano: 5. 7. 2024.
- [30] ScotXW. *Generic block diagram of a GPU.svg*. Licenca: CC0 preko Wikimedijine zbirke. URL: {https://commons.wikimedia.org/wiki/File:Generic_block_diagram_of_a_GPU.svg}. Dostopano: 18. 7. 2024.
- [31] Honinbo Shusaku in StackExchange. *What is a windowing system?* URL: {<https://unix.stackexchange.com/questions/109195/what-is-a-windowing-system>}. Dostopano: 18. 7. 2024.
- [32] StackOverflow in user3549833. *How to display something on screen through linux framebuffer?* URL: {<https://stackoverflow.com/questions/32082248/how-to-display-something-on-screen-through-linux-framebuffer>}. Dostopano: 5. 7. 2024.

- [33] GTK Development Team. *Gtk -- 4.0* (URL: <https://creativecommons.org/licenses/by-sa/4.0/>). *Widget Gallery*. URL: {https://docs.gtk.org/gtk4/visual_index.html}. Dostopano: 18. 7. 2024.
- [34] tldp.org. *Overdriving Your Monitor*. URL: {<https://tldp.org/HOWTO/XFree86-Video-Timings-HOWTO/overd.html>}. Dostopano: 13. 7. 2024.
- [35] Shmuel Csaba Otto Traian. *Linux graphics drivers DRI current.svg*. Licenca: CC BY-SA 3.0 (URL: <https://creativecommons.org/licenses/by-sa/3.0/>) preko Wikimedijine zbirke. URL: {https://commons.wikimedia.org/wiki/File:Linux_graphics_drivers_DRI_current.svg}. Dostopano: 18. 7. 2024.
- [36] Shmuel Csaba Otto Traian. *The Linux Graphics Stack and glamor.svg*. Licenca: CC BY-SA 4.0 (URL: <https://creativecommons.org/licenses/by-sa/4.0/>) preko Wikimedijine zbirke. URL: {https://commons.wikimedia.org/wiki/File:The_Linux_Graphics_Stack_and_glamor.svg}. Dostopano: 18. 7. 2024.
- [37] Shmuel Csaba Otto Traian. *Wayland display server protocol.svg*. Licenca: CC BY-SA 3.0 (URL: <https://creativecommons.org/licenses/by-sa/3.0/>) preko Wikimedijine zbirke. URL: {https://commons.wikimedia.org/wiki/File:Wayland_display_server_protocol.svg}. Dostopano: 18. 7. 2024.
- [38] Shmuel Csaba Otto Traian. *Window (windowing system).svg*. Licenca: CC BY-SA 3.0 (URL: <https://creativecommons.org/licenses/by-sa/3.0/>) preko Wikimedijine zbirke. URL: {[https://commons.wikimedia.org/wiki/File:Window_\(windowing_system\).svg](https://commons.wikimedia.org/wiki/File:Window_(windowing_system).svg)}. Dostopano: 18. 7. 2024.
- [39] Wikipedia contributors. *Compositing window manager — Wikipedia, The Free Encyclopedia*. URL: {https://en.wikipedia.org/wiki/Compositing_window_manager}. Dostopano: 18. 7. 2024.
- [40] Wikipedia contributors. *Direct Rendering Infrastructure — Wikipedia, The Free Encyclopedia*. URL: {https://en.wikipedia.org/wiki/Direct_Rendering_Infrastructure}. Dostopano: 13. 7. 2024.
- [41] Wikipedia contributors. *Direct Rendering Manager — Wikipedia, The Free Encyclopedia*. URL: {https://en.wikipedia.org/wiki/Direct_Rendering_Manager}. Dostopano: 5. 7. 2024.
- [42] Wikipedia contributors. *Framebuffer — Wikipedia, The Free Encyclopedia*. Opomba: glej poglavje Display modes. URL: {<https://en.wikipedia.org/wiki/Framebuffer>}. Dostopano: 5. 7. 2024.
- [43] Wikipedia contributors. *GLFW — Wikipedia, The Free Encyclopedia*. URL: {<https://en.wikipedia.org/wiki/GLFW>}. Dostopano: 18. 7. 2024.
- [44] Wikipedia contributors. *GLX — Wikipedia, The Free Encyclopedia*. URL: {<https://en.wikipedia.org/wiki/GLX>}. Dostopano: 18. 7. 2024.
- [45] Wikipedia contributors. *Graphical widget — Wikipedia, The Free Encyclopedia*. URL: {https://en.wikipedia.org/wiki/Graphical_widget}. Dostopano: 18. 7. 2024.

- [46] Wikipedia contributors. *Mesa (computer graphics)* — Wikipedia, The Free Encyclopedia. URL: [https://en.wikipedia.org/wiki/Mesa_\(computer_graphics\)](https://en.wikipedia.org/wiki/Mesa_(computer_graphics)). Dostopano: 13. 7. 2024.
- [47] Wikipedia contributors. *Rendering (computer graphics)* — Wikipedia, The Free Encyclopedia. URL: [https://en.wikipedia.org/wiki/Rendering_\(computer_graphics\)](https://en.wikipedia.org/wiki/Rendering_(computer_graphics)). Dostopano: 18. 7. 2024.
- [48] Wikipedia contributors. *Synchronous dynamic random-access memory* — Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/Synchronous_dynamic_random-access_memory. Dostopano: 5. 7. 2024.
- [49] Wikipedia contributors. *Text Mode* — Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/Text_mode. Dostopano: 5. 7. 2024.
- [50] Wikipedia contributors. *Window manager* — Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/Window_manager. Dostopano: 18. 7. 2024.
- [51] Wikipedia contributors. *Windowing system* — Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/Windowing_system. Dostopano: 18. 7. 2024.
- [52] Wikipedia contributors. *X Window System* — Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/X_Window_System. Dostopano: 18. 7. 2024.
- [53] Wikipedia contributors. *X.Org Server* — Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/X.Org_Server. Dostopano: 18. 7. 2024.
- [54] Cornell Virtual Workshop. *Design: GPU vs. CPU*. URL: <https://cvw.cac.cornell.edu/gpu-architecture/gpu-characteristics/design>. Dostopano: 5. 7. 2024.