Universitat Politècnica de Catalunya - BarcelonaTech

Facultat d'Informàtica de Barcelona

Grau en Enginyeria Informàtica

Especialitat d'enginyeria de computadors

Treball de fi de grau

Optimització de Weston mitjançant atomic mode-setting

Autor:

Sergi Granell Escalfet

Director:

Juan Jose Costa Prats

21 de juny de 2017

Agraïments

En primer lloc m'agradaria agrair al meu director del projecte, Juan Jose Costa Prats, per donar-me l'oportunitat de realitzar aquest projecte com a treball final de grau.

També voldria agrair a Daniel Stone, Pekka Paalanen i a tota la comunitat que hi ha al darrere del projecte *Wayland* en general per resoldre els meus dubtes, aconsellar-me i donar-me idees en els moments que ho he necessitat. Sens dubte, sense el seu suport m'hauria resultat molt més difícil realitzar el projecte amb èxit.

I per últim, però no menys important, vull agrair a tota la meva família i amics per animar-me i donar-me sempre suport en fer tot allò que m'ha agradat, i que m'han donat confiança a tirar endavant.

Resum

Després d'introduir els conceptes bàsics i explicar l'arquitectura dels diversos components d'un entorn gràfic de Linux, aquest treball mostra com s'han dut a terme certes optimitzacions i millores al compositor referència de *Wayland*: *Weston*.

En primer lloc es realitza un programa de prova, *atomictest*, que mostra com utilitzar el *Direct Rendering Manager* de Linux des d'una perspectiva d'usuari. Seguidament, es mostra el disseny i la implementació de les diverses millores realitzades a *Weston*, i per últim s'avaluen aquestes i se n'extreuen les conclusions oportunes.

Resumen

Después de introducir los conceptos básicos y explicar la arquitectura de los diversos componentes de un entorno gráfico de Linux, este trabajo muestra cómo se han llevado a cabo ciertas optimizaciones y mejoras al compositor referencia de *Wayland*: *Weston*.

En primer lugar se realiza un programa de prueba, *atomictest*, que muestra cómo utilizar el *Direct Rendering Manager* de Linux desde una perspectiva de usuario. Seguidamente, se muestra el diseño y la implementación de las diversas mejoras realizadas a *Weston*, y por último se evalúan estas y se extraen las conclusiones oportunas.

Abstract

After introducing the basics and explaining the architecture of the various components of a Linux graphic environment, this paper shows how some optimizations and improvements have been made to the *Wayland*'s reference compositor: *Weston*.

First of all, a test program called *atomictest* is developed to show how to use the *Direct Rendering Manager* of Linux from a user perspective. Secondly, the design and implementation of the various improvements made to *Weston* are shown, and finally these are evaluated and the appropriate conclusions are extracted.

Índex

Índex de figures

Índex de taules

Índex de figures

Índex de taules

1 Introducció

En aquest capítol s'introdueix el context sobre el qual se sustenta el projecte i es fa una introducció més detallada de l'objectiu principal d'aquest: optimitzar un dels components d'un servidor gràfic modern de Linux (anomenat *Weston*) mitjançant funcionalitats noves per accelerar les operacions gràfiques, afegides en les últimes versions de Linux.

1.1 Context

Des de ben els inicis de la computació, els humans ens hem interessat en la interfície humà-màquina com a mètode d'interacció i control d'aquestes. Inicialment, a l'era dels anys quaranta-seixanta, va aparèixer el que es coneix com a *batch processing/interface*, o interfície/processament per lots, és a dir, un tipus d'interfície (si és que es pot considerar realment com a tal) en la qual no hi havia cap mena d'interacció entre l'humà i la màquina. L'entrada més comuna a aquest tipus de processament eren les targetes perforades, que contenien instruccions per a realitzar un càlcul concret.

Més endavant, a mesura que la tecnologia va anar avançant i els computadors es van anar fent més complexos i alhora assequibles, les necessitats dels usuaris es van tornar més diverses; aleshores va començar a ser més necessari que els programes fossin més immersius i interactius amb l'usuari, i és quan van començar a aparèixer les *command-line interfaces* (*CLI*) o interfícies de línia d'ordres.

Les *CLI* permeten a l'usuari realitzar tasques molt precises de forma eficient (i més encara quan s'agrupen diverses ordres en un mateix *script*), per la qual cosa la majoria d'usuaris d'aquest tipus d'interfície són habitualment programadors o administradors de sistemes i usuaris domèstics avançats.

Va ser cap al final de la dècada dels anys seixanta quan uns investigadors del grup de recerca del *Stanford Research Institute* van crear *NLS* (*oN-Line System*)[1], que es podria considerar com el precursor de les interfícies gràfiques d'usuari (*GUI* o *Graphic User Interface*)[2]. L'*NLS* va ser el primer sistema a incorporar un sistema de *links* basats en hipervincles[1], un ratolí, un sistema de finestres, programes de presentació i altres conceptes que es consideren moderns en l'àmbit de la informàtica.

[1]Element de referència o de navegació en un document que condueix a una altra secció del mateix document o a qualsevol altre recurs.

Al voltant dels inicis dels vuitanta, un grup de docents i estudiants del grup de sistemes distribuïts de la universitat d'Standford, van començar a desenvolupar un sistema operatiu del tipus *microkernel*[2] anomenat *V-System*[3]. Aquest sistema operatiu s'utilitzava per fer recerca en el camp de les interfícies gràfiques i disponia d'un sistema de finestres anomenat *W Window System*[4]. Alhora, l'*MIT* (*Massachusetts Institute of Technology*) en conjunt amb *DEC* (*Digital Equipment Corporation*) i *IBM* (*International Business Machines Corporation*) van iniciar un projecte conjunt anomenat *Project Athena*[5]. Un dels objectius del *Project Athena* era facilitar als estudiants l'accés als recursos computacionals. En conseqüència, el projecte necessitava un sistema de gràfics independent de la plataforma, mitjançant el qual es pogués unir el sistema heterogeni *multiple-vendor*[3] sobre el qual se sostenia el mateix projecte. L'any 1984, un enginyer de l'*MIT* va portar l'*W Window System* a Unix[4], tot millorant-lo canviant les funcions síncrones per unes d'asíncrones equivalents, i va anomenar a aquest port *X Window System* (també conegut com a X)[6].

En els seus inicis, i aproximadament fins a l'aparició de les primeres *GPU*s[5], el servidor X era l'encarregat de dibuixar a petició dels diversos clients i canviar la configuració de la pantalla. A més a més, era el mateix X qui es comunicava amb el *hardware* directament, i per tant el procés que implementava el servidor de les X havia d'executar-se com a *root*. Això era necessari, ja que es volia que X fos el més agnòstic a la plataforma sobre la qual s'estava executant possible, d'aquesta forma el mateix codi servia per a tota la gamma de sistemes operatius *UNIX-like*[6].

Amb l'arribada dels gràfics 3D, es va voler evitar haver de passar totes les comandes gràfiques cap al X server i que aquest les enviés cap al *hardware* corresponent, amb el consegüent augment de la latència, ja que això suposaria una degradació del rendiment del sistema molt important. Per tant, va crear-se un nou enfocament de com permetre als clients realitzar gràfics 3D sense haver de delegar les comandes a un tercer, anomenat *direct rendering*. La implementació d'aquesta nova idea en el servidor X es va anomenar *Direct Rendering Infrastructure*, o simplement *DRI*[8].

[2]Petit kernel de sistema operatiu que proporciona les bases per ampliacions modulars.

[3]De múltiples proveïdors.

[4]Sistema operatiu de propòsit general, multi-usuari i interactiu desenvolupat als *Bell Labs*.

[5]Unitat de procés gràfic, de l'anglès *Graphic Processing Unit*.

[6]També conegut com a *UN*X* o **nix*, és aquell sistema operatiu que té un comportament similar al sistema Unix, tot i no conformar necessàriament o ser certificat per cap versió de la *Single UNIX Specification* [7].

Un dels components essencials de *DRI* és el *Direct Rendering Manager* (*DRM*)[9][10]. El *DRM* és la capa/subsistema del *kernel* de Linux que s'encarrega de gestionar tots aquells dispositius gràfics, *GPU*s i acceleradors, que es poden trobar a les targetes de vídeo. A més a més, el *DRM* també exposa certa funcionalitat cap a l'espai d'usuari del sistema operatiu.

Com passa amb tots els projectes de software que són actius, tota aquesta infraestructura ha anat evolucionant, i actualment (juny del 2017) l'última versió de l'especificació de les X és la 11 (coneguda com X11), sent X.Org la implementació referència d'aquesta[11]; *DRI* es troba en la versió 3 (DRI3), tot i que encara s'utilitza DRI2 per defecte a menys que l'usuari ho canviï; i en el *DRM* s'hi han anat introduint noves característiques a mesura que les targes gràfiques han anat evolucionant. Un dels canvis més recents i rellevants en el *DRM* ha sigut la inclusió del que se'n diu *atomic mode-setting*, que tracta d'actualitzar el *hardware* relacionat amb la pantalla de forma 'tot o res', tot minimitzant els possibles *glitch*s[7] visuals i reduint el consum energètic[12].

Tot i les millores, extensions i actualitzacions que s'han anat incorporant al servidor X per tal de millorar les seves funcionalitats, al final algunes d'aquestes han acabat caient en desús, tot i que és necessari que estiguin implementades per tal de poder executar aplicacions llegat, cosa que fa que moltes de les línies de codi que componen X al final no s'acabin executant, per tant es pot considerar X com a una aplicació 'inflada'. A més a més, el paradigma del *hardware* sobre el qual s'executarà X ha canviat molt des del moment de la seva concepció fins als temps actuals, fent que molts dels conceptes sobre els quals es basa X hagin quedat obsolets o desfasats pel maquinari modern, fet que evita que X pugui treure el màxim rendiment de la màquina.

Conscient de tots aquests obstacles que eviten que X sigui una arquitectura gràfica moderna i lleugera, l'any 2008 Kristian Høgsberg, llavors enginyer de l'empresa Red Hat, va fer públic un projecte en el qual havia estat treballant durant part del seu temps lliure: *Wayland*[13].

[7] Fallada molt curta que habitualment es soluciona sola, deguda principalment a una mala configuració breu del *hardware*.

Wayland es tracta d'un nou enfocament a la idea d'un gestor de finestres modern, on hi intervenen dues parts: el compositor (o servidor), encarregat de gestionar i dibuixar l'escena final a partir de les imatges dels clients, i els clients, que són independents l'un a l'altre i són els que generen les imatges. A més a més el projecte *Wayland* també conté un compositor referència anomenat *Weston*, que serveix com a exemple per mostrar com utilitzar *Wayland*, i a la vegada com un banc de proves per a idees avantguardistes relacionades amb el subsistema DRM de Linux.

1.2 Motivació

Com a usuari de Linux, sempre m'han interessat totes aquelles novetats relacionades amb aquest; ja siguin noves característiques o millores en el *kernel* o actualitzacions d'aplicacions a nivell d'usuari. Una de les coses que sempre he trobat fascinant és tota la infraestructura que hi al voltant del subsistema o *stack* (pila) que gestiona tot allò relacionat amb els gràfics: anant des dels *drivers* de la *GPU* a nivell del *kernel*, fins a com utilitzar una API[8] gràfica a nivell d'usuari, com per exemple OpenGL[9] o la més moderna i potent Vulkan[10], per tal de poder dibuixar en tres dimensions.

Així mateix, ja fa bastants anys que segueixo de molt a prop el projecte *Wayland* i tots els components relacionats amb aquest, fins al punt de seguir quasi diàriament la *mailing list*[11] que utilitzen els desenvolupadors per tal d'enviar *patches* i debatre sobre aquests.

A més a més, també li tinc molt de respecte a tota la comunitat *Open Source*, i sempre he desitjat poder contribuir a aquesta.

Consegüentment, he triat dur a terme la realització d'aquest projecte, ja que em permet, a més a més de l'objectiu principal, aprendre, contribuir i efectuar, entre d'altres, els següents reptes que es podrien considerar d'un caire més personal:

- Aprendre a utilitzar una *Raspberry Pi*[14], en concret el model 3, ja que utilitzaré aquest computador per tal de realitzar algunes execucions i proves de *Weston* al llarg del projecte.

[8] *Application Programming Interface*, és un conjunt de subrutines, funcions i procediments que ofereix certa llibreria per ser utilitzada per altre software com una capa d'abstracció.

[9] `https://www.khronos.org/opengl/`

[10] `https://www.khronos.org/vulkan/`

[11] Llista dels noms i adreces de les persones a les quals se li poden enviar per correu regular matèria de publicitat, informació, o un altre material.

- Aprendre a instal·lar un *cross-compiler* i a utilitzar-lo per a poder compilar binaris per una altra arquitectura (*Raspberry Pi 3*) que no sigui la del *host*.
- Aprendre a configurar i a compilar el *kernel* Linux, en particular utilitzant un *cross-compiler*.
- Aprendre a utilitzar *autotools*, conjunt d'eines de programació del projecte GNU que faciliten la tasca de compilació de codi font de manera portable, i és comunament utilitzat en gran part de les aplicacions *open source*.
- Contribuir en un projecte *open source*.
- Aprendre a utilitzar una *mailing list*, eina que s'utilitza en la majoria de projectes *open source* per tal de debatre canvis suggerits en el codi font.

1.3 Objectius

L'objectiu d'aquest projecte és aplicar diverses millores i optimitzacions al codi font de *Weston* de tal forma que aquest utilitzi les noves funcionalitats del subsistema DRM del sistema operatiu Linux, amb la finalitat de millorar l'eficiència i reduir el consum energètic del sistema, i de millorar-ne l'experiència de l'usuari quant a fluïdesa gràfica.

Finalment, un cop acabada la implementació de les optimitzacions, es realitzaran proves de rendiment comparant la versió original de *Weston* i la meva versió amb les optimitzacions aplicades, per tal de poder extreure'n uns resultats que mostrin si realment hi ha hagut una millora significativa i extreure'n les conclusions pertinents.

1.4 Actors implicats

Potencialment, tots aquells usuaris que utilitzin un *desktop environment* (*DE*) (entorn gràfic) a Linux es poden veure beneficiats pels resultats d'aquest projecte pel següent motiu: tot i que l'*scope* del treball està enfocat en optimitzar *Weston*, com ja he explicat anteriorment, *Weston* es pot considerar com banc de proves per a idees avantguardistes en l'àmbit del subsistema *DRM* del *kernel* de Linux. D'aquesta manera, altres *DE*s com per exemple *GNOME* o *KDE* podran aprofitar-se dels resultats d'aquest treball i extreure algunes de les idees utilitzades en les optimitzacions realitzades durant el transcurs del projecte per tal d'aconseguir les millores obtingudes.

Així doncs, podem dividir els possibles beneficiaris en tres grups:

- **Desenvolupadors de *Wayland*/*Weston*:** ja que l'objectiu del projecte no només és optimitzar *Weston* mitjançant les funcions atòmiques de *DRM*, sinó que per tal de dur a terme aquesta tasca, s'ha de fer un *refactoring* i simplificació del codi font, cosa que farà el projecte més mantenible, i per tant en facilitarà el futur desenvolupament.
- **Desenvolupadors que utilitzin *Weston***: com que l'optimització quedarà encapsulada dins *Weston*, l'usuari que utilitzi *Weston* com llibreria s'hi beneficiarà sense que hagi de modificar el seu codi.
- **Usuaris finals:** aquests els podem sub-dividir en dos grups:
  - usuaris que utilitzin *Weston* com a llibreria.
  - usuaris que utilitzin un altre compositor que hagi aplicat les optimitzacions resultants d'aquest projecte.

2 Estat de l'art

En aquest capítol es descriurà quin és l'estat de l'art en l'àmbit d'aquest projecte, és a dir, tots aquells components capdavanters que o bé hagin sortit fa relativament poc, o bé s'estiguin debatent per un llançament futur, i que puguin influenciar d'alguna forma el treball. Quasi tots els termes referenciats en aquest capítol formen part del subsistema *DRM* de Linux, i és que aquest és la base fonamental sobre la qual s'aguanta tota la infraestructura de *Wayland*.

En els últims anys hi ha hagut un esforç per modernitzar *DRM* cap a un *framework* modern, consolidant-se amb l'*atomic mode-setting*, que permet modificar el mode de com es mostra la pantalla i la configuració *hardware* gràfic d'una tacada i sincronitzadament amb l'objectiu d'evitar *glitches* visuals.

A més a més, també s'està intentant incorporar a la versió original (*upstream*) de Linux, totes aquelles modificacions sobre sistema operatiu per mòvils Android que Google va fer. Una d'aquestes és l'anomenat *explicit fencing*, que serveix per assegurar que hi hagi una sincronització correcta entre el productor i el consumidor d'imatges gràfiques. Aquest tipus de sincronització ha estat afegida satisfactòriament a la versió 4.10 de Linux[15], i el principal usuari d'aquesta és el compositor sobre *DRM* que fa servir Android anomenat *drm_hwcomposer2* (o *HWC2*)[16].

També s'està discutint la possibilitat d'afegir al *kernel* el que s'anomenen *writeback connectors*[17], que a part de facilitar la depuració d'aplicacions que utilitzin tota aquesta interfície moderna, també permeten poden gravar o realitzar captures de pantalla sense que hi hagi cap mena de penalització al rendiment del sistema.

Un altre sistema operatiu, també basat en Linux, que també fa ús d'*atomic mode-setting* i tota la infraestructura relacionada amb aquest és Tizen[18], desenvolupat per Samsung.

A part dels dos sistemes mencionats, en aquest moment (juny del 2017), no n'hi ha cap altre que utilitzi tecnologies tan noves relacionades amb l'àmbit de *DRM*, i per aquest motiu es poden considerar *state of art technologies*.

3 Abast i possibles obstacles

En aquest capítol hi podem trobar fins quins dels punts referenciats en el capítol anterior aquest projecte té intenció d'aprofundir, quins obstacles hi poden haver al llarg del desenvolupament d'aquest, i quines possibles solucions s'aplicaran.

3.1 Abast

Aquest és un projecte involucrat en molts dels conceptes nous que s'han anat introduint al subsistema *DRM*, no obstant això, i per la raó principal de la durada del projecte, és impossible arribar a aprofundir en tots. D'aquesta manera l'abast del projecte es limita a modernitzar l'ús de les funcions de DRM que utilitza *Weston* canviant-les per tal d'utilitzar les relacionades amb l'*atomic mode-setting*. La presa d'aquesta decisió no ha estat pas difícil: tota la resta de funcionalitats exposades en el capítol 2 fan implícitament el supòsit que s'utilitza *atomic mode-setting*.

3.2 Possibles obstacles

- **Canvi de la *ABI*[12] de *Weston*:** Pot passar si els desenvolupadors de *Wayland* treuen una nova versió de *Weston* amb una API diferent.

  Possibles solucions:

  - Utilitzar una versió antiga (que no hagi patit el trencament de la *ABI*). Tot i que en principi és una solució desaconsellable (habitualment és millor realitzar els canvis a l'última versió), ja que és possible que el canvi ajudi a realitzar la implementació del projecte, caldria fer una anàlisis.
  - Portar els canvis realitzats deguts a la implementació del projecte a l'última versió. En principi aquesta seria la solució aplicada, tot i que seria recomanable realitzar un anàlisis previ per assegurar que la solució triada és potencialment la millor.

- **Canvi de la *ABI* del *kernel* de Linux:** En el cas que passi, serà degut al fet que he actualitzat el *kernel* a una versió més nova, per tant és un problema similar a l'anterior.

[12] Interfície binària d'aplicació, és la interfície entre dos mòduls de programa, un dels quals és, habitualment, una llibreria o sistema operatiu, a nivell de llenguatge màquina.

Possibles solucions:

- Com que l'API de *libdrm* abstreu la interfície cap al subsistema *DRM* del kernel, una possible solució seria esperar que la implementació d'aquesta llibreria reflecteixi el canvi de Linux, en el cas que afecti. Seria la millor solució tot i que s'hauria de tenir en compte que l'actualització de *libdrm* pot no ser immediata.
- Utilitzar una versió més antiga de Linux. A menys que sigui imprescindible, rebutjar aquesta solució, ja que és possible que la nova versió de Linux millori alguns aspectes de *DRM*.

- **La *Raspberry Pi 3* es trenqui:** Poc probable, però s'ha de considerar la possibilitat.

  Possibles solucions:

  - La solució més òbvia seria comprar-ne una de nova gràcies al seu preu reduït. Molt possiblement aquesta seria la solució triada.
  - Utilitzar un altre computador que executi Linux i que el seu *hardware* suporti les funcions atòmiques, com per exemple els ordinadors de les aules de sistemes operatius de la facultat.

4 Metodologia

En aquest capítol es descriurà el mètode de treball que s'emprarà al llarg del transcurs del projecte. També es realitzarà una explicació de quin mètode de seguiment s'utilitzarà i el mètode de validació del treball.

4.1 Mètode de treball

Aquest projecte és potser un treball de fi de grau no massa ordinari, ja que el seu objectiu és de millorar una aplicació existent en el mercat, utilitzada per usuaris i empreses, i que està en continuo desenvolupament amb enginyers al seu darrere. Per aquest motiu la sincronització amb aquests desenvolupadors és crucial per poder dur a terme el projecte satisfactòriament.

D'aquesta manera, abans de poder fer cap optimització o millora a *Weston* haig de parlar amb els desenvolupadors d'aquest, per posar-nos d'acord i que quedi constància que duré a terme tal tasca.

A més a més, periòdicament hauré de contactar amb els mateixos desenvolupadors per informar sobre l'estat actual de la feina, i en particular per quan acabi una tasca, assignar-me'n una de nova.

Pel que fa al desenvolupament del projecte, aquest es realitzarà des d'un ordinador *host* (principalment el meu portàtil) i una *Raspberry Pi 3*. A l'ordinador portàtil és a on es durà a terme el desenvolupament i les execucions principals de *Weston*, tot i que de tant en tant també s'utilitzarà la *Raspberry Pi 3* per comprovar que els canvis realitzats funcionen correctament als dos dispositius, i que per tant aquests canvis no són específics a una sola plataforma. El sistema operatiu que s'utilitzarà en els dos dispositius és Linux.

Si és necessari, per a modificar el codi remotament, utilitzaré *sshfs*[19] i compilaré a través de *SSH*[13]. Una alternativa seria *cross-compilar* des de l'ordinador *host*, tot i que això te l'avantatge d'utilitzar una màquina més potent per compilar, i per tant reduir el temps, *Weston* té diverses dependències (a altres llibreries) i per tant es faria bastant molest.

Finalment, un cop cregui que els canvis realitzats són correctes, utilitzaria l'eina *GIT* com a sistema de control de versions d'aquesta i forma mantenir un registre organitzat de tots els canvis i modificacions realitzades.

[13] *Secure SHell*, serveix per accedir a màquines remotes a través de la xarxa.

Així doncs, pel que fa a la metodologia de treball, s'assemblaria molt a fer *refactoring* incloent millores, de la pràctica de desenvolupament *agile*.

4.2 Mètode de seguiment

L'eina de control de versions *GIT* obliga, en certa forma, mitjançant els comentaris dels *commits*, a realitzar un seguiment dels canvis que s'estan realitzant.

A més a més, per tal de dur a terme un control més exhaustiu del temps dedicat a cada tasca, anotaré les hores dedicades a aquestes.

També realitzaré reunions periòdiques amb el director del projecte, a més a més d'estar en constant contacte amb els desenvolupadors de *Weston* per assegurar-me que el rumb que segueixo és l'adequat.

4.3 Mètode de validació

A causa de la naturalesa del projecte, la forma de validar els canvis es basa fortament en *trial and error* (prova i error), ja que no hi ha cap mètode senzill ni de comprovar el resultat dels canvis, ni de realitzar proves que comprovin la correctesa de *Weston* de forma exhaustiva. Per tant el mètode de validació serà principalment comprovar que no hi hagi cap defecte visual a simple vista.

Per aquest motiu, i per facilitat la detecció de problemes, es faran proves contínuament després de cada canvi al codi font.

5 Planificació temporal

Si no hi han complicacions, el temps destinat a la realització del projecte és des de l'inici del curs de gestió de projectes (GEP, 20 de febrer del 2017) fins al primer dia dels torns de lectura (26 de juny del 2017), és a dir, aproximadament uns quatre mesos. Tot i això, per tenir cert marge de temps al final, consideraré que la part tècnica del projecte s'haurà de tenir acabada dues setmanes abans de l'inici dels torns de lectura, és a dir, al 12 de juny del 2017.

Així doncs, en aquest capítol es mostra el desglossament del projecte en tasques, les dependències entre aquestes (en cas que n'hi hagi), quins recursos seran necessaris i quines són les seves restriccions.

5.1 Descripció de les tasques

En aquest apartat es descriuen quines tasques formen el projecte, quins recursos es necessiten, i quantes hores s'hi dedicaran a cada una. A més a més cada tasca pertany a una etapa o activitat, on en cada activitat hi trobem les tasques relacionades entre si. A la següent taula es mostra a quina activitat pertany cada tasca:

Activitat	**Tasques**
Configuració de l'entorn	Configuració de l'ordinador Instal·lació *cross-compiler* Compilació del *kernel* Linux Configuració de la *Raspberry Pi 3*
Aprenentatge eines	Aprenentatge *autotools* Aprenentatge *libdrm*
Optimització de *Weston*	Familiarització codi Buscar pla d'optimització Realitzar l'optimització Proves de rendiment

Taula 1: Activitats i les seves tasques.

5.1.1 Gestió del projecte (GEP)

La primera tasca és la corresponent al curs de Gestió de projectes, que correspon a 3 dels 18 crèdits ECTS que conformen el projecte. Aquesta tasca engloba tots els sis entregables que s'han de lliurar a GEP. Segons la guia de l'assignatura, els entregables i hores dedicades tant d'aprenentatge dirigit com d'aprenentatge autònom a cada un són les següents:

- **Lliurable 1, Abast del projecte i contextualització**: 24.5 hores.
- **Lliurable 2, Planificació temporal**: 8.25 hores.
- **Lliurable 3, Gestió Econòmica i sostenibilitat**: 9.25 hores.
- **Lliurable 4, Presentació preliminar**: 6.25 hores.
- **Lliurable 5, Plec de condicions (Bloc Especialitats)**: 12.5 hores.
- **Lliurable 6, Presentació oral i document final**: 18.25 hores.

Els recursos principals necessaris per a la realització d'aquesta tasca són: un ordinador amb un editor de text i *pdflatex*, *git* com a eina de còpia de seguretat, un *software* de creació de presentacions (per exemple Google Docs) i un mòbil per gravar la presentació preliminar.

5.1.2 Configuració de l'ordinador

En primer lloc, abans de poder començar a realitzar el projecte, cal configurar l'ordinador en el que es durà a terme el treball. Aquesta tasca inclou l'instal·lació i configuració d'una distribució de Linux, *Arch Linux* en particular.

5.1.3 Instal·lació d'un *cross compiler* per a l'arquitectura ARM

Atès que part de les execucions i proves de rendiment es faran sobre la *Raspberry Pi 3*, cal instal·lar un *cross-compiler*, és a dir, un compilador que produeixi binaris diferents de l'arquitectura sobre la qual es compila, ja que la *Raspberry Pi 3* té una arquitectura *ARM* i no la trobada en els ordinadors habituals (*x86-64*). El *cross-compiler* es pot instal·lar utilitzant *crosstool-ng*[14] per exemple, però, ja que el meu entorn de treball consta de la distribució *Arch Linux*, utilitzaré el paquet *arm-linux-gnueabihf-gcc*[15] dels repositoris *AUR*.

[14] `http://crosstool-ng.org/`
[15] `https://aur.archlinux.org/packages/arm-linux-gnueabihf-gcc/`

5.1.4 Compilació del *kernel* Linux

Un cop instal·lat el *cross-compiler*, ja es poden generar binaris per la *Raspberry Pi 3*, per tant és un bon moment de familiaritzar-se amb la compilació del *kernel* de Linux, especialment quan es compila per una arquitectura diferent. La compilació de Linux serà necessària per poder tenir els últims canvis i actualitzacions que hi hagi en els controladors del xip gràfic de la *Raspberry Pi 3*.

5.1.5 Configuració de la *Raspberry Pi 3*

Una vegada obtingut el binari del *kernel* compilat, aquest s'ha de copiar a la targeta *microSD* i modificar l'arxiu de configuració de la *Raspberry Pi 3* per indicar-li quina imatge de *kernel* utilitzar quan arrenqui. A més a més també s'ha de copiar els possibles mòduls de Linux i el *DTB* (device tree blob) generats a l'etapa de compilació.

Arribats a aquest punt, la *Raspberry Pi 3* ja arrenca utilitzant el nostre *kernel* compilat, per tant ara és moment de configurar l'entorn: creació d'usuaris, instal·lar paquets necessaris, creació d'*scripts* que facilitin algunes feines, etc.

En particular, s'establirà una adreça *IP* estàtica i s'instal·larà un servidor *SSH*, amb el qual mitjançant l'eina *SSHFS*[16] es podrà accedir al sistema de fitxers de la *Raspberry Pi 3* remotament des del meu ordinador. També s'instal·laran els compiladors natius i eines necessàries per poder compilar *Weston* i les seves dependències manualment.

5.1.6 Aprenentatge de les eines *autotools* escrivint un programa de prova

Gran part dels projectes de codi lliure (incloent-hi *Wayland*, *Weston*, i la majoria de les seves dependències) utilitzen el conjunt d'eines de programació *autotools* per tal de facilitar la compilació d'aquests, així que considero una bona idea aprendre a fer-les anar, i per això escriuré un programa de prova senzill (un '*hello world*' per exemple) utilitzant-les.

[16] `https://github.com/libfuse/sshfs`

5.1.7 Aprenentatge de la llibreria *libdrm* escrivint un programa que faci ús de les funcions atòmiques

Weston es comunica amb el subsistema *DRM* de Linux a través de la llibreria *libdrm*, per tant és essencial entendre bé com funciona i quines funcionalitats té aquesta. Una bona forma d'aprendre a utilitzar-la és escriure un programa de prova que la utilitzi, en particular que faci ús de les funcions relacionades amb l'*atomic mode-setting*. A més a més, per a la compilació del programa de prova s'usaran els coneixements d'*autotools* apresos a la tasca 5.1.6. Aquest programa de prova també podrà ser útil per saber en quin estat es troba la implementació i si hi ha errors a la infraestructura atòmica dels controladores gràfics de diferents plataformes i *GPU*s.

5.1.8 Familiarització amb el codi de *Weston*

Abans de poder buscar un pla d'optimització, s'ha d'entendre bé el codi i l'organització de *Wayland* i les diferents components de *Weston* (*backends*, *renderers* i protocols). Aquesta tasca també inclou aprendre a compilar *Wayland* i *Weston* i veure quines opcions de compilació i característiques es poden activar i desactivar.

5.1.9 Buscar pla per tal d'optimitzar *Weston*

Aquesta tasca és clau per tal de dur a terme satisfactòriament l'optimització utilitzant les funcions atòmiques de *libdrm*. Un mal pla d'optimització pot resultar en un increment molt gran del temps que s'haurà de dedicar a la tasca 5.1.10. Per tant és vital que abans de començar a modificar el codi hi hagi un pla que digui quins arxius es tocaran, com canviaran les funcions i a on s'haurà de fer *refactoring* per poder utilitzar les funcions atòmiques.

5.1.10 Dur a terme l'optimització

Dur a terme el pla d'optimització realitzat a la tasca 5.1.9 es pot considerar la tasca principal i la més extensa de tot el projecte. Durant el transcurs d'aquesta tasca, s'aniran fent modificacions de mica en mica, segons el pla establert, i recompilant i provant que aquestes modificacions incrementals funcionen. A causa de les dimensions del projecte *Weston*, és inconcebible realitzar totes les optimitzacions d'una sola tacada amb èxit.

5.1.11 Realitzar proves de rendiment abans i després de la optimització

Un cop realitzada satisfactòriament l'optimització, és moment de dur a terme les proves de rendiment i de consum. No només es compararan les versions de *Weston* amb la millora de la tasca 5.1.10 i sense aquesta, sinó que també es realitzaran proves comparant el rendiment del servidor X, així doncs la comparació serà a tres bandes. Aquestes proves es realitzaran en una *Raspberry Pi 3* i també als ordinadors dels laboratoris de sistemes operatius de la FIB, que disposen[17] d'un processador *Intel Core i7 4770* que té una gràfica integrada, i està molt ben suportada per Linux.

5.1.12 Memòria final

La memòria del projecte és una part important d'aquest, ja que sosté tota la feina feta durant el transcurs del treball de fi de grau. Aquesta és l'única tasca que es realitzarà paral·lelament a les anteriors, considerant-ne l'inici un cop finalitzat el curs de gestió de projectes. També s'hi inclou la preparació de la defensa davant del tribunal.

5.2 Dependències entre tasques

A la Taula 2 s'hi poden observar quines dependències té cada tasca, és a dir, quines tasques s'han d'haver completat abans de poder realitzar la següent. Cal destacar que la tasca d'escriure la memòria del projecte es farà en paral·lel amb les altres un cop finalitzada la tasca de gestió del projecte.

[17] `http://www.fib.upc.edu/fib/serveis/informatiques.html`

Tasca	**Tasca predecessora**
Gestió del projecte	-
Configuració de l'ordinador	Gestió del projecte
Instal·lació cross-compiler	Configuració de l'ordinador
Compilació del kernel Linux	Instal·lació cross-compiler
Configuració de la *Raspberry Pi 3*	Compilació del kernel Linux
Aprenentatge *autotools*	Configuració de la *Raspberry Pi 3*
Aprenentatge *libdrm*	Aprenentatge *autotools*
Familiarització codi de *Weston*	Aprenentatge *libdrm*
Buscar pla per optimitzar *Weston*	Familiarització codi de *Weston*
Realitzar l'optimització de *Weston*	Buscar pla per optimitzar *Weston*
Proves de rendiment	Realitzar l'optimització de *Weston*
Memòria final	Gestió del projecte

Taula 2: Dependències entre les tasques del projecte.

5.3 Estimació del temps dedicat a cada tasca

A la Taula 3 hi veiem una estimació del temps que s'haurà de dedicar a cada tasca. El treball de fi de grau consta de 18 crèdits ECTS[18], dels quals 3 són destinats al curs de gestió de projectes (GEP). Cada crèdit ECTS equival a 25 hores[19] de treball, per tant el nombre total d'hores a dedicar al TFG és de: $18\ ECTS \cdot 25\ h/ECTS = 450\ hores$.

[18] `http://www.fib.upc.edu/fib/estudiar-enginyeria-informatica/treball-final-grau.html`

[19] `http://www.upc.edu/aprendre/estudis/graus/faqs_estudis_grau#credits_ects`

Tasca	**Temps dedicat (hores)**
Gestió del projecte	75
Configuració de l'ordinador	18
Instal·lació *cross-compiler*	12
Compilació del *kernel* Linux	12
Configuració de la *Raspberry Pi 3*	13
Aprenentatge *autotools*	10
Aprenentatge *libdrm*	36
Familiarització codi de *Weston*	22
Buscar pla per optimitzar *Weston*	22
Realitzar l'optimització de *Weston*	140
Proves de rendiment	55
Memòria final	35
Total	**450**

Taula 3: Estimació del temps dedicat a les tasques del projecte.

5.4 Diagrama de Gantt

Veure l'apèndix A.

5.5 Recursos

Durant el transcurs del projecte, s'utilitzaran diversos recursos, en els apartats següents es detallen quins són segons el tipus.

5.5.1 Recursos *hardware*

- **Ordinador portàtil**: Lenovo ThinkPad X230 amb Intel(R) Core(TM) i7-3520M CPU @ 2.90GHz i 8GB de memòria RAM.
- **Raspberry Pi 3 Model B**: 64-bit quad-core ARMv8 CPU @ 1.2GHz i 1GB de RAM.
- **Monitor**: Monitor TV LED LG M2631D de 26" i resolució 1920x1080, a on anirà connectada la *Raspberry Pi* a través d'*HDMI*.
- **Càmera de vídeo**: S'utilitzarà la incorporada en un mòbil Google LG Nexus 4, per a gravar la presentació preliminar de GEP.

5.5.2 Recursos *software*

- **Sistema operatiu de l'ordinador portàtil**: *Arch Linux*
- **Sistema operatiu de la *Raspberry Pi***: *Arch Linux*
- **Editor de text**: *Geany*
- **Compilador**: *GNU C Compiler* (*GCC*)
- ***Debugger***: *GNU Debugger* (*GDB*)
- **Eines de desenvolupament**: *make* i *autotools*
- **Sistema de control de versions**: *git*

5.5.3 Recursos humans

- **Desenvolupador**: Encarregat de dur a terme i desenvolupar el projecte.
- **Director**: Supervisa que el projecte avanci correctament.
- **Desenvolupadors de *Wayland* i *Weston***: Poden ajudar en cas de tenir dubtes, i encarregats de donar el vistiplau de les optimitzacions.

5.6 Valoració d'alternatives i pla d'acció

La tasca principal del projecte és la de dur a terme l'optimització, i com que excepte la tasca de realitzar el pla d'optimització, les altres anteriors no són estrictament necessàries per dur a terme l'optimització, si no aconseguís realitzar-les en el temps estimat, les deixaria de fer i passaria a la següent. Per tant es poden considerar tasques crítiques la configuració de la *Raspberry Pi*, la de realitzar el pla d'optimització i la de dur a terme aquest pla; la resta de tasques són molt útils per poder estar el màxim preparat possible, però no vitals.

6 Gestió econòmica

6.1 Estimació dels costos

6.1.1 Costos directes

Els costos directes són tots aquells que estan directament relacionats amb l'activitat i són clarament identificables, sense ells no s'haurien pogut obtenir els recursos o eines necessaris per dur a terme el projecte. Així doncs, els costos directes engloben principalment tot aquell *hardware* i *software* que s'ha hagut d'adquirir per tal de realitzar el projecte.

A les següents dues taules es mostra un desglossament dels costos directes, en funció de si són *software* o *hardware*, indicant la vida útil i l'amortització (relació del cost entre el temps) dels mateixos considerant la seva vida útil.

Software	**Preu**	**Unitats**	**Vida útil**	**Amortització**
Sistema operatiu (*Arch Linux*)	0	1	∞	0
Editor de text (*Geany*)	0	1	∞	0
Compilador (*GCC*)	0	1	∞	0
Debugger (*GDB*)	0	1	∞	0
Eines *dev.* (*make*, *autotools*)	0	1	∞	0
Control versions (*git*)	0	1	∞	0
Servidor *git* (`www.github.com`)	\$7[20]	4	1 mes	0.00972\$/h
Total	**26.24€**[21]			**0.0091(€/h)**

Taula 4: Cost del software.

Hardware	**Preu (€)**	**Unitats**	**Vida útil**	**Amortització**
ThinkPad X230	680	1	6 anys	0.014€/h
Raspberry Pi 3 Model B	33.95	1	2 anys	0.002€/h
TV LED LG M2631D	229	1	4 anys	0.007€/h
Google Nexus 4	199	1	5 anys	0.005€/h
Total	**1141.95**			**0.03(€/h)**

Taula 5: Cost del hardware.

[20] `https://github.com/pricing`
[21] Considerant 1€= \$1.07

Podem observar com a causa de la forta naturalesa *open-source* del treball, els costos del *software* són ínfims, ja que la majoria de components i eines utilitzades pertanyen al projecte *GNU*. Pel que fa als costos *hardware*, la majoria de components són les que un programador o fins i tot usuari normal tindria a casa, per tant en realitat només s'ha hagut d'adquirir explícitament la *Raspberry Pi 3*.

Un altre cost directe important a considerar és el cost de la mà d'obra, és a dir els salaris. En el projecte hi intervé el desenvolupador principal (jo) fent d'enginyer informàtic, a més a més dels responsables de *Wayland* i *Weston* que contribueixen a aquests sigui de forma personal o de forma laboral, i per tant s'assumeix que l'únic cost de recursos humans pel que fa a aquest treball de fi de grau ve donat per mi. També s'assumirà que un enginyer informàtic cobra 20 euros l'hora, i sent la duració del projecte de 450 hores, obtenim el següent cost directe en recursos humans: $20€/h \cdot 450h = 9000€$.

6.1.2 Costos indirectes

Per altra banda, els costos indirectes no tenen una vinculació tan òbvia amb el projecte en si, i poden ser compartits per altres activitats o projectes. Aquests costos inclouen principalment la tarifa elèctrica, la tarifa d'internet i possibles desplaçaments, siguin privats o amb transport públic. A la següent taula es mostren els costos indirectes associats a aquest projecte:

Cost indirecte	**Preu (€)**	**Unitats**	**Vida útil**	**Amortització (€/h)**
Tarifa elèctrica	55	4	1 mes	0.08
Tarifa internet	60	4	1 mes	0.09
T-Jove 4 zones	244[22]	1	3 mesos	0.12
Total	**704**			**0.28**

Taula 6: Costos indirectes del projecte.

6.2 Imprevistos i contingències

La realització del projecte s'ha dividit en tasques precises de forma detallada, i per tant si va tot bé no hi haurien d'haver imprevistos. Tot i així és important considerar els imprevistos com a una cosa possible, ja que podrien arribar a perjudicar greument l'èxit del mateix treball.

[22] `https://www.tmb.cat/ca/barcelona/tarifes-metro-bus/abonaments/t-jove`

Com ja s'ha mostrat anteriorment, el projecte està dividit en tres grups de tasques (activitats), i es pot considerar que cada una té un pes o importància diferent, així doncs el percentatge de contingència pels imprevistos sobre el total de l'activitat per cada una d'aquestes s'ha establert com es mostra a continuació:

Activitat	**Percentatge de contingència**
Configuració de l'entorn	10%
Aprenentatge de les eines	5%
Optimització de *Weston*	15%

Taula 7: Percentatges de contingència pels imprevistos en funció de l'activitat.

A més a més s'estima un factor de contingència del 10% global al projecte, que pot ajudar a tenir una mica més de marge en cas d'imprevistos, i com que el cost del projecte no és massa elevat gràcies al seu caire de *software* lliure, això no suposa un increment molt alt del preu d'aquest.

6.3 Control de gestió

Per tal de dur a terme un control de la gestió del projecte satisfactori, s'han d'establir bé quins indicadors es faran servir per a detectar desviacions, i com i quan s'utilitzaran. El principal indicador que s'utilitzarà és el de comptabilitzar el total d'hores dedicades a cada tasca, ja que a més a més d'útil, és molt senzill d'aplicar. Aquest és un indicador de desviacions típic utilitzat en la majoria de projectes informàtics, ja que la desviació més habitual que acostumen a tenir és la de no haver planificat correctament el temps que s'hauria de dedicar a cada tasca, i passar-se d'aquest temps un cop el projecte es posa en marxa.

Així doncs, es comptabilitzarà explícitament el temps que es dedica a cada una de les tasques de les diferents activitats, i en cas de trobar una desviació, que és molt probable que sigui la de passar-se del temps prèviament estimat, s'estudiarà per quins motius això ha passat i com repercutirà consegüents tasques del desenvolupament del projecte. En principi aquest desviament comportarà haver de reduir la durada d'una o més de les tasques posteriors, i s'avaluarà quines d'aquestes són les menys problemàtiques si s'escurça la seva durada.

En el cas que el desviament temporal sigui quedar-se curt (completar la tasca amb menys temps de l'estimat), s'haurà de comprovar que realment la tasca s'ha realitzat correctament, i si aquest és el cas, es podria ampliar el temps dedicat a alguna de les tasques posteriors, en principi tindrien preferència les de les activitats més crítiques (*Configuració de l'entorn* i *Optimització de Weston*).

A la següent taula es mostra el pressupost total del projecte, on cal destacar que els costos directes es calculen en funció de la seva amortització i durada, i pels imprevistos es tenen en compte els costos directes de cada activitat (grup de tasques) i les hores de l'activitat pels recursos humans:

	Amortització (€/h)	Durada (h)	Cost (€)
Gestió del projecte		**75**	**1.425**
ThinkPad X230	0.014		1.05
Google Nexus 4	0.005		0.375
Configuració de l'ordinador		**18**	**0.252**
ThinkPad X230	0.014		0.252
Instal·lació *cross-compiler*		**12**	**0.168**
ThinkPad X230	0.014		0.168
Compilació del *kernel* Linux		**12**	**0.276**
ThinkPad X230	0.014		0.168
Raspberry Pi 3 Model B	0.002		0.024
TV LED LG M2631D	0.007		0.084
Configuració de la *Raspberry Pi 3*		**13**	**0.117**
Raspberry Pi 3 Model B	0.002		0.026
TV LED LG M2631D	0.007		0.091
Aprenentatge autotools		**10**	**0.14**
ThinkPad X230	0.014		0.14
Aprenentatge libdrm		**36**	**0.828**
ThinkPad X230	0.014		0.504
Raspberry Pi 3 Model B	0.002		0.072
TV LED LG M2631D	0.007		0.252
Familiarització codi de Weston		**22**	**0.308**
ThinkPad X230	0.014		0.308
Buscar pla per optimitzar Weston		**22**	**0.308**
ThinkPad X230	0.014		0.308
Realitzar l'optimització de Weston		**140**	**3.22**
ThinkPad X230	0.014		1.96
Raspberry Pi 3 Model B	0.002		0.28
TV LED LG M2631D	0.007		0.98
Proves de rendiment		**55**	**1.265**
ThinkPad X230	0.014		0.77
Raspberry Pi 3 Model B	0.002		0.11
TV LED LG M2631D	0.007		0.385
Memòria final		**35**	**0.49**
ThinkPad X230	0.014		0.49
Imprevistos			**873.91**
Configuració de l'entorn	10% sobre 1100.81		110.09
Aprenentatge de les eines	5% sobre 920.968		46.05
Optimització de Weston	15% sobre 4785.101		717.77
Costos indirectes			**704**
Recursos humans			**9000**
Total acumulat			**10,586.704**
Contingència	10% sobre 10,586.704		**1058.68**
Total sense IVA			**11,645.38**
Total amb IVA	21% sobre 11,645.38		**14,091**

Taula 8: Pressupost total del projecte.

7 Sostenibilitat i compromís social

L'estudi de sostenibilitat d'aquest projecte es farà mitjançant l'enfocament de tres punts de vista diferents: l'econòmic, el social i l'ambiental. En aquest apartat es mostrarà la matriu de sostenibilitat del projecte, i tot seguit s'avaluaran aquestes tres dimensions.

7.1 Matriu de sostenibilitat

	Projecte en producció	**Vida útil**	**Riscos**
Ambiental	Consum del disseny	Empremta ecològica	Riscos ambientals
	9:10	20:20	0:0
Econòmic	Factura	Pla de viabilitat	Riscos econòmics
	7:10	15:20	0:0
Social	Impacte personal	Impacte social	Riscos socials
	10:10	17:20	0:0
Valoració total	26:30	52:60	0:0
	78:90		

Taula 9: Matriu de sostenibilitat del projecte.

7.2 Dimensió econòmica

A la secció 6 es mostra de forma detallada tots els recursos necessaris, ja siguin de *software*, *hardware* o humans per a la correcta realització del projecte.

Com que la naturalesa del projecte és la de realitzar una optimització d'un programa ja existent, no té massa sentit considerar la competitivitat a menys que el projecte en si tingués un caire competitiu, que no és el cas.

Tal com es pot observar quan s'exposen els recursos necessaris, la quantitat i el preu d'aquests és relativament baix, i gran part del pressupost necessari és degut al salari del desenvolupador (jo mateix), per tant és difícil pensar que es podrien reduir molt els recursos i que el projecte es pugui realitzar correctament.

Per altra banda, es dedica la major part del temps a les tasques que tenen més dificultat o extensió, com es pot observar per exemple en la tasca de *Realitzar l'optimització de Weston*.

Tot i que *Wayland* és un projecte de codi lliure, hi ha empreses que dediquen recursos i diners a millorar-lo, com per exemple *Collabora*[23], de tota manera, l'única relació amb aquestes empreses serà quant a revisió i acceptació de les optimitzacions sorgides d'aquest treball i no pas en l'àmbit lucratiu.

7.3 Dimensió social

Actualment Espanya, i en concret Catalunya té un dels creixements econòmics més elevats a la Unió Europea, tot i que això es deu principalment a la forta crisi econòmica que va començar fa uns anys. Pel que fa a Barcelona, es considera com una de les ciutats punteres en el sector tecnològic i de les *startups*[20]. Així doncs, crec que ara és un bon moment per tal de dur a terme no només el meu projecte, sinó qualsevol projecte de caràcter tecnològic en general, i per tant en cap cas la situació del país podria perjudicar el meu projecte, en tot cas podria millorar-lo.

En l'actualitat gairebé tothom utilitza un entorn gràfic quan utilitza un sistema informàtic amb una pantalla (sigui un ordinador personal o en un cotxe, un telèfon mòbil, etc.) i per tant crec que és necessari que aquesta experiència sigui la més satisfactòria possible, on a més a més del disseny de la *GUI*, això també inclou la fluïdesa i rendiment d'aquesta, per tant crec que aquest treball pot ajudar en aquest aspecte.

7.4 Dimensió ambiental

La majoria si no tots els recursos materials d'aquest projecte són altament reaprofitables tant en un entorn personal com en un altre projecte posteriorment.

Així mateix, un dels motius principals darrere l'optimització de *Weston* és la de fer que aquest aprofiti més els recursos *hardware* dels que disposa el sistema sobre el qual s'executa, i per tant fer que aquest consumeixi menys. Això és molt interessant sobretot per sistemes encastats (*embedded systems*), ja que aquests habitualment requereixen un baix consum energètic, i a més a més, els recursos utilitzats per realitzar el projecte són de baix consum energètic.

[23] `https://www.collabora.com/about-us/open-source/open-source-projects/wayland.html`

D'aquesta forma, el desenvolupament del projecte no generarà cap tipus de contaminació directament, a excepció del requeriment d'energia elèctrica necessari per fer funcionar l'ordinador de desenvolupament, la *Raspberry Pi 3* i la pantalla on anirà connectada, i com ja s'ha dit anteriorment, totes aquestes components es podran continuar fent servir un cop acabat el projecte, tot allargant-ne la vida útil.

8 Conceptes bàsics

En aquest capítol s'explicaran quins són i quin paper juguen tots aquells conceptes relacionats amb el desenvolupament del projecte. En particular, el capítol es dividirà en tres apartats: en primer lloc s'introduiran conceptes teòrics que tenen una rellevància important en la realització d'aquest projecte, en segon lloc es presentaran els components *hardware* necessaris perquè *Weston* (i qualsevol entorn gràfic en general) pugui funcionar, i per últim es mostrarà el vessant *software*, anant des del sistema operatiu fins al servidor X i *Wayland*, i acabant en *Weston*.

8.1 Conceptes teòrics

8.1.1 *Buffer*

Un *buffer* és una zona de memòria de dimensions definides utilitzada per guardar-hi dades.

Habitualment en un *buffer* s'hi emmagatzemen dades de caràcter temporal, i el seu principal ús és evitar que el programari o recurs que les requereix, sigui *hardware* o *software*, es quedi sense dades durant una transferència de dades d'entrada/sortida irregular a causa de diferència de la velocitat de processament entre les dues parts (productor i consumidor).

8.1.2 *Framebuffer*

Tal i com el seu nom indica, un *framebuffer* (o FB) és una porció de memòria (*buffer*) que conté les dades, en aquest cas píxels, que es mostren en pantalla, és a dir, un fotograma.

8.1.3 Renderitzar

S'anomena renderitzar o simplement dibuixar al procés que realitza un computador per generar les dades d'un *framebuffer*, és a dir, una imatge, que quan es mostra visualment té una interpretació vàlida per qui l'observa. Cada finestra d'un entorn gràfic renderitza cap al seu *framebuffer*.

8.1.4 Compositar

Se li diu a l'acció realitzada per un compositor quan aquest renderitza, és a dir, genera una nova imatge, a partir d'un o més *framebuffers* ja existents i informació que indica de quina manera ha de posicionar i transformar aquests.

8.1.5 *Mode-setting*

El *mode-setting*, o establiment de mode, és l'acció de canviar la configuració, habitualment la resolució, d'un dispositiu de visualització, com per exemple una pantalla o un monitor.

8.1.6 *Page-flipping*

El *page-flipping* consisteix a canviar el *framebuffer* que mostra una pantalla per un d'altre. Habitualment el sistema disposa de diversos *framebuffers*, només un dels quals es mostra per pantalla en un instant determinat (*front buffer*) mentre que s'està renderitzant cap a un d'ocult (*back buffer*). L'objectiu és evitar dibuixar cap a un *framebuffer* que s'estigui mostrant, ja que això podria comportar veure defectes visuals.

El nombre de *framebuffers* utilitzats més freqüentment són dos o tres, coneguts com a *double* i *triple buffering* respectivament.

8.2 Components *hardware*

Els components bàsics que permeten la visualització de gràfics en un computador són els quatre mostrats a la figura 1: la CPU i els perifèrics connectats, la GPU i el seu *display controller*, i la pantalla on es mostren els *framebuffers* generats. A les següents seccions és descriuen les tasques que realitzen aquests components.

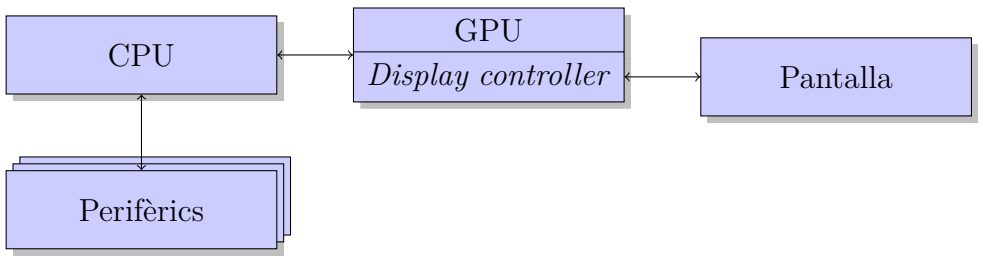


Figura 1: Visió general dels components que permeten la visualització de gràfics.

8.2.1 Unitat central de processament (CPU)

És el circuit electrònic d'un ordinador que s'encarrega d'executar les instruccions i processar les dades del programa que executa. A més a més també controla l'entrada sortida (*I/O*) ja sigui mitjançant ports (*port-mapped I/O*) o accessos a memòria a adreces d'entrada/sortida (*memory-mapped I/O*).

En el context del projecte, cal destacar que la CPU és qui executa el *kernel* de Linux i tots els controladors i mòduls que controlen els diversos perifèrics, i en particular la GPU i el seu controlador de vídeo.

8.2.2 Unitat de processament gràfic (GPU)

La tasca principal d'aquest circuit és la del processament de gràfics i operacions en coma flotant, sovint amb la intenció posterior de mostrar-les a un visualitzador o pantalla.

Si bé juguen una part important en la creació d'interfícies gràfiques i entorns d'escriptori, la part més rellevant pel que fa al treball és que les GPUs realitzen renderització accelerada pel seu *hardware* i que la majoria de GPUs modernes disposen d'un controlador de vídeo incorporat.

8.2.3 Controlador de vídeo (*display controller*)

El controlador de vídeo és el circuit integrat encarregat de generar el senyal de vídeo cap a una pantalla o televisor. Actualment, la majoria d'aquests controladors es troben integrats com a blocs funcionals en el mateix dau de la GPU, APU (CPU i GPU en un mateix xip) o SoC (*system on chip*), especialment en els sistemes encastats.

Aquest circuit juga un paper molt rellevant, ja que disposa del que s'anomena *hardware planes/overlays*, que permeten mostrar diverses imatges a la vegada sense haver de renderitzar-les manualment, fet que comporta un estalvi d'energia i recursos. A més a més, per tal d'evitar defectes visuals quan es mostra contingut, es deixa un lapse de temps anomenat interval vertical o *VBLANK* al final de cada *frame* en el que no es subministren dades d'imatge, per tant és el moment ideal per realitzar el *page-flipping*.

8.2.4 Pantalla

És el dispositiu de sortida que permet mostrar informació visual. Actualment les pantalles digitals proveeixen al sistema operatiu una estructura de dades anomenada EDID[24] (*Extended Display Identification Data*), que conté informació sobre totes les capacitats d'aquesta, com per exemple els temps de sincronització i resolucions suportades, els formats de colors, el nom que identifica el monitor, etc.

8.3 Components *software*

8.3.1 Linux *Kernel*

El *kernel* és el nucli del sistema operatiu, amb un control total sobre el sistema, encarregant-se de gestionar tots els dispositius d'entrada sortida (mitjançant el que s'anomenen controladors). També proporciona diverses funcionalitats al *user space* a través de punts d'entrada anomenats crides al sistema (*syscalls*). Una d'aquestes funcionalitats de les quals disposa tot espai d'usuari de qualsevol sistema operatiu és la comunicació entre processos o IPC (*inter-process communication*), que permet a un procés comunicar-se amb un altre, sense comprometre la seguretat del sistema (a vista del *kernel*). Per tal de facilitar la tasca de desenvolupament i gestió del codi, Linux està dividit en subsistemes, cada un encarregat d'una feina concreta.

El subsistema encarregat de gestionar els dispositius gràfics (GPUs i targetes de vídeo) s'anomena DRM (*Direct Rendering Manager*).

8.3.2 Mesa

Mesa[25] és la implementació *open-source* per excel·lència d'inicialment OpenGL i més endavant de moltes de les APIs del consorci industrial *Khronos Group*[26], com per exemple OpenVG, OpenGL ES (*Embedded Systems*), EGL i Vulkan. Mesa actua com una capa de traducció entre les APIs gràfiques (com per exemple OpenGL) i els *drivers* del sistema operatiu dels dispositius gràfics.

[24] `https://en.wikipedia.org/wiki/Extended_Display_Identification_Data`
[25] `https://www.mesa3d.org/`
[26] `https://www.khronos.org/developers`

8.3.3 *X server*

Actualment el servidor X (*X Window System* o simplement X11) és el sistema gràfic més utilitzat en Linux i altres sistemes Unix-like. X es basa en l'arquitectura de client/servidor, on els clients són les diverses aplicacions, i el servidor és l'encarregat de gestionar l'entrada/sortida.

Un entorn gràfic habitual que utilitzi el servidor X consta de tres components ben diferenciades, que estan en processos del sistema operatiu diferents: els clients, el servidor X, i un client especial anomenat *Window Manager* o WM, que s'encarrega principalment de gestionar les posicions de les finestres i dibuixar les seves decoracions. Normalment el WM també actua com a compositor.

8.3.4 *Wayland*

Wayland pretén ser un reemplaçament per a X, que sigui més senzill i fàcil de desenvolupar i mantenir. *Wayland* en si és tant el protocol base, asíncron i orientat a objectes, que utilitzen els clients i el servidor (també anomenat compositor), com una implementació en C d'aquest protocol (*libwayland*).

L'arquitectura de *Wayland* és similar a la de X (model client/servidor) però amb una diferència important: ara el servidor és un sol procés que actua a la vegada el compositor i el *Window Manager*.

8.3.5 *Weston*

Una part del projecte *Wayland* consta d'un compositor de referència de *Wayland* anomenat *Weston*, utilitzat com a prova de concepte. Aquest compositor és mínim (en el sentit a què no aspira a ser un entorn gràfic d'ús general complet) i ràpid, cosa que el fa adequat per a sistemes encastats i mòbils. La idea principal d'aquest és ensenyar com utilitzar *Wayland* i les seves llibreries per tal de facilitar l'adopció i la migració del servidor X cap a *Wayland* en els entorns d'escriptori més utilitzats.

9 Arquitectura de la pila gràfica de *Linux*

Ara que ja coneixem els conceptes bàsics més importants, procedim a explicar amb més detall com funcionen els diferents components de la pila gràfica que podem trobar en un sistema *Linux* habitual.

9.1 *Direct Rendering Manager* (DRM)

DRM exposa una API als programes d'usuari que permet enviar comandes i dades cap a les GPUs i controladores de vídeo i realitzar operacions com per exemple dur a terme el *mode-setting* i *page-flipping*.

DRM consisteix de dues parts: una part genèrica anomenada *DRM core* i una d'específica, anomenada *DRM driver*. El *DRM core* conté tota la infraestructura bàsica en la qual els diversos *DRM drivers* es poden registrar, a més a més de proveir a l'espai d'usuari d'una sèrie d'`ioctls` amb funcionalitats comunes i independents del *hardware*. Per altra banda, un *DRM driver* implementa la part de DRM que és dependent del *hardware*, específica a la GPU que suporta, i per tant també pot exposar `ioctls` addicionals específics.

Seguint la filosofia d'Unix de 'tot és un arxiu', DRM proporciona aquests serveis a través de crides al sistema `ioctl`, que es poden utilitzar sobre els descriptors de fitxer (*file descriptors*) oberts de `/dev/dri/cardX`, que representen les GPUs de les que disposa el nostre sistema.

La part de l'API de DRM que permet realitzar el *mode-setting* es coneix com a *Kernel Mode Setting* (KMS), tot i que a vegades DRM i KMS s'utilitzen indistintament.

9.1.1 *Kernel Mode Setting* (KMS)

KMS és una API genèrica, independent del dispositiu, que proporciona una forma de configurar el *pipeline* de visualització i *display controller* d'una targeta gràfica o sistema encastat, i intenta reemplaçar el ja antic i desfasat *fbdev*[27] (*Framebuffer Devices*).

[27] `https://www.kernel.org/doc/Documentation/fb/framebuffer.txt`

KMS ha estat dissenyat tenint en compte els components d'un controlador de pantalla modern, i aquest fet es reflecteix en la seva API. Els principals components d'aquesta API i la seva relació són els mostrats a la figura 2 i explicats posteriorment.

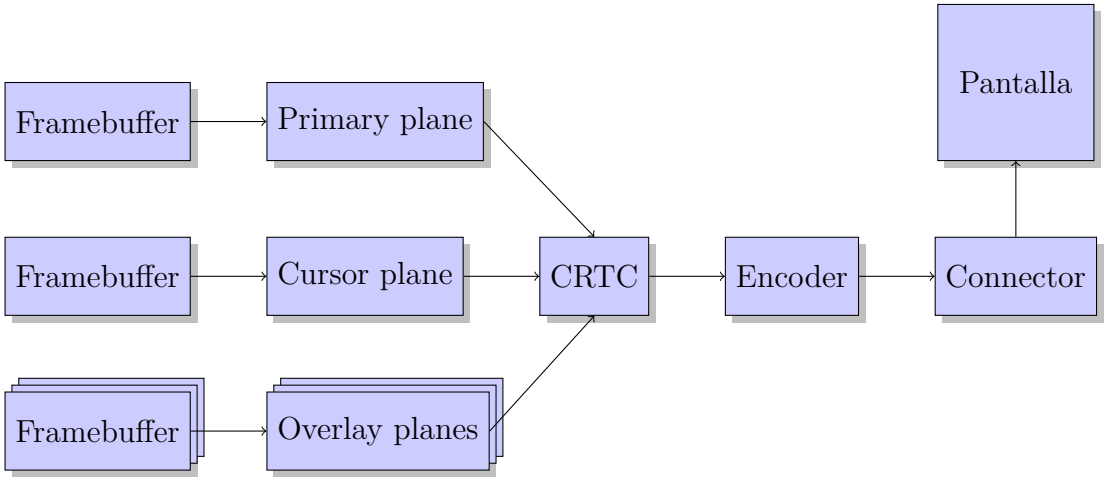


Figura 2: Visió general dels components de DRM/KMS.

Framebuffer Objecte estàndard que guarda referències a regions de memòria dels continguts que es volen mostrar, el format d'aquests continguts (mida, formats de píxel, etc.), i l'àrea activa d'aquesta regió de memòria.

Plane És una capa d'imatge, el contingut final que es mostrarà al *CRTC* és resultat de la composició de diferents *planes*. N'hi ha de tres tipus diferents:

- *Primary*: Típicament utilitzat per la imatge de fons, obligatori i un per *CRTC*.
- *Cursor*: Utilitzat per mostrar el cursor del ratolí per exemple, opcional i un per *CRTC*.
- *Overlay*: Utilitzats per beneficiar-se de la composició per *hardware*. Pot mostrar finestres amb contingut dinàmic (com per exemple un vídeo). És opcional i cada *CRTC* en pot tenir zero o més.

CRTC : Tot i que el nom prové de controlador del tub de raigs catòdics, és el component encarregat de realitzar l'*scanout*, és a dir, generar el contingut composat cap a una o més pantalles. També és a on es configura la resolució, i els temps de sincronització de la pantalla.

Encoder : És l'encarregat de convertir un *frame* (sortida del *CRTC*) al format i codificació que accepta el *connector*, com per exemple *TMDS*[28] per *HDMI*.

Connector : Representa un connector físic (*HDMI*, *DP*, *VGA*, etc.) cap a un monitor o pantalla (sigui integrada o externa), per tant és a on es genera el senyal elèctric.

Així doncs, mitjançant l'API de DRM/KMS podem consultar les configuracions i capacitats dels *CRTCs*, *encoders* i *connectors*, establir modes (resolució) i *framebuffers*, obtenir el nombre de *planes* disponibles, quins formats de píxel suporten i a quin *CRTC* es poden connectar, i obtenir propietats d'aquests objectes, en definitiva.

9.1.2 Deficiències de l'API antiga no atòmica

Tot i les capacitats de l'API '*legacy*' de DRM/KMS, aquesta té certes deficiències que fan que estigui lluny de ser tan potent com promet; per exemple: els *mode sets* es fan molt tard, i per tant, si fallen, no hi ha forma de tirar enrere, l'API no garanteix que la configuració serà aplicada en la seva totalitat en el pròxim *frame*, el *page-flipping* dels *framebuffers* i l'establiment d'aquests en els *hardware planes* no està sincronitzat amb el *VBLANK* (en algunes implementacions és fins i tot bloquejant), fet que inutilitza completament les capacitats d'aquests, i un llarg etcètera.

9.1.3 *Atomic mode-setting*

Per tal de solucionar les deficiències mencionades anteriorment, i afegir noves capacitats, des de fa uns anys els desenvolupadors de DRM intenten que els *drivers* evolucionin cap a una interfície nova, anomenada *atomic mode-setting* i sota l'eslògan: 'cada *frame* és perfecte'. La gran potència d'aquesta nova filosofia resideix en els següents factors:

- Permet validar una configuració abans d'aplicar-la:
  - El *hardware* no es toca fins que el *driver* garanteix que la configuració és valida i es pot aplicar
  - No hi ha necessitat de tirar enrere (*rollback*)
- Permet actualitzacions atòmiques a una configuració:

[28] `https://en.wikipedia.org/wiki/Transition-minimized_differential_signaling`

  - Es poden actualitzar diversos *planes* a la vegada
  - Això garanteix '*frames* perfectes'
- Permet la simplificació i unificació dels *drivers*:
  - El codi 'llegat' es pot eliminar
  - Gran part del codi s'unifica en 'funcions d'ajuda' (conegudes com a *helpers*)

9.2 Mesa

La versió suportada de cada API gràfica que Mesa implementa depèn de cada *driver*, ja que cada un té la seva implementació, i per tant el seu propi estat. Per aquest fet, Mesa també proporciona *wrappers* o abstraccions als diferents *ioctl()s* específics de cada dispositiu. Aquests *wrappers* tenen el format d'una llibreria anomenada *libdrm_specific*, on *specific* és el nom del *driver* del dispositiu gràfic, a més a més d'una llibreria genèrica que agrupa totes les funcionalitats bàsiques anomenada *libdrm*.

9.2.1 Tipus de *buffers* gràfics

El subsistema DRM en si ja exposa d'una interfície, anomenada *Dumb-Buffers*[29], que ens permet crear *buffers* que podran ser utilitzats pels components d'aquest subsistema, i en particular, realitzar-ne l'*scanout* (ja sigui en el *primary* o en la resta de *planes*). La característica principal d'aquest tipus de *buffers* gràfics és que permeten mapejar-los a l'espai de direccions de la CPU mitjançant la crida al sistema `mmap`, i posteriorment escriure-hi. Tot i això, és possible que l'accés per GPU a aquests *buffers* no sigui possible, així que habitualment s'utilitzen per realitzar tasques senzilles on es renderitzi mitjançant la CPU i no s'hi hagi d'involucrar renderitzat per GPU.

Per reservar *dumb-buffers* s'ha de realitzar un `ioctl` amb el paràmetre `DRM_IOCTL_MODE_CREATE_DUMB` sobre el descriptor de fitxer de DRM obert, i després per mapejar-lo a l'espai d'adreces primer es crida un `ioctl` amb el paràmetre `DRM_IOCTL_MODE_MAP_DUMB` i posteriorment l'`mmap`.

[29] `http://manpages.ubuntu.com/manpages/xenial/man7/drm-memory.7.html#Dumb-Buffers`

El format de píxel utilitzat pels *dumb-buffers* segueix la definició de formats de píxel i modificadors utilitzada dins DRM que es regeix per un identificador del tipus *FOURCC*[30], definit a `include/drm/drm_fourcc.h`[31] en el codi de Linux.

Si el que volem és reservar *buffers* que puguin ser utilitzats per renderitzar, Mesa també proporciona una API anomenada GBM (*Generic Buffer Management*) en format de llibreria: *libgbm*. GBM també realitza la tasca d'abstracció, ja que cada un dels *drivers* de DRM té els seus *ioctl()s* propis per reservar *buffers* accelerats, i per tant internament utilitza els diversos *libdrm_specific* per facilitar aquesta tasca.

El *FB id* que s'ha de passar en les crides de *libdrm* que el requiereixin s'ha d'obtenir a partir dels *buffers* natius (*dumb buffers* o *buffers* de GBM) i utilitzant les crides `drmModeAddFB` i derivades de la mateixa *libdrm*.

9.2.2 Ús bàsic de *libdrm*

La majoria de funcions de *libdrm* tenen com a primer paràmetre un *file descriptor* (*fd*) que ha de ser obert mitjançant la crida al sistema `open` a partir del node de la targeta gràfica, usualment situat a `/dev/dri/card0`.

Aquesta llibreria és *event-driven*, i per tant tots els esdeveniments de *libdrm* els rebrem llegint a través del descriptor de fitxer obert. Habitualment per esperar a aquests esdeveniments s'utilitza la crida al sistema `poll`, passant-hi el nostre *fd* obert de DRM i dient que bloquegi fins que hi hagi esdeveniments de lectura. Quan `poll` desperti, voldrà dir que hi ha *events* que s'han de tractar, i per tant s'ha de cridar a `drmHandleEvent`. Aquesta funció té un *struct* com a paràmetre, que conté els punters a funcions dels diferents esdeveniments.

Utilitzant la funció `drmGetCap` podem obtenir informació diversa, com la mida del *cursor plane* que suporta el *hardware*, si el *driver* suporta *dumb-buffers*, etc. La funció `drmModeGetResources` és clau, ja que retorna una estructura amb una llista de tots els *CRTCs*, *connectors* i *encoders* disponibles. Per obtenir la informació específica dels *CRTCs*, *connectors* i *encoders* s'utilitzen les crides `drmModeGetCrtc`, `drmModeGetConnector` i `drmModeGetEncoder` respectivament. Paral·lelament també disposem de `drmModeGetPlane` per tal d'obtenir informació dels diversos *planes*, com per exemple l'identificador o el seu tipus.

[30] `https://www.fourcc.org/`

[31] `http://elixir.free-electrons.com/linux/latest/source/include/drm/drm_fourcc.h`

9.2.3 Funcions no atòmiques de *libdrm*

Per una banda, `drmModeGetCrtc` retorna l'estat actual del *pipeline* de DRM/KMS, i per altra banda, podem aplicar un estat propi amb `drmModeSetCrtc`. Aquesta última crida té com a paràmetres els següents elements principals: el *fd*, el *CRTC id*, el *FB id* que volem posar al *primary plane*, el *connector id*, i el mode (la mida) que volem utilitzar.

Els *page-flips* es sol·liciten amb la funció `drmModePageFlip`. Els paràmetres més importants són els següents: el *fd*, el *CRTC id*, per facilitar la gestió de diversos *CRTCs*, el *FB id* que es vol posar al *primary plane*, i uns *flags* per indicar quins esdeveniments es volen obtenir.

Per situar i moure el *cursor plane* s'utilitzen les crides `drmModeSetCursor` i `drmModeMoveCursor` respectivament, i per modificar l'estat dels *overlay planes* es fa servir `drmModeSetPlane`, que té els següents paràmetres: el *fd*, el *plane id*, l'*id* del *CRTC* a on volem posar-lo, el *FB id* que volem que mostri l'*overlay*, diversos *flags*, i la posició i mides d'origen i destí d'aquest. A la figura 3 hi podem observar com actuen els paràmetres de posició i mida (`crtc_x`, `crtc_y`, `crtc_w`, `crtc_h`, `src_x`, `src_y`, `src_w` i `src_h`).

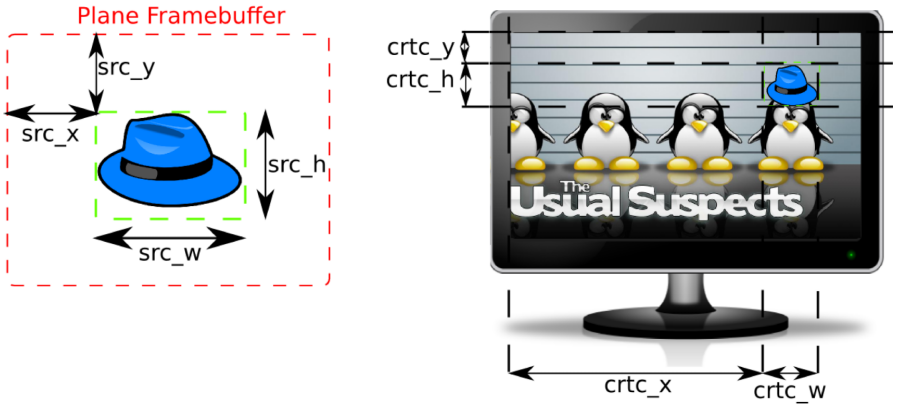


Figura 3: Paràmetres de posicionament d'un plane en un CRTC.

Font: ***https://events.linuxfoundation.org/sites/events/files/slides/brezillon-drm-kms.pdf***

9.2.4 Funcions atòmiques de *libdrm*

Un dels problemes de `drmModeSetPlane` és que aquesta funció està subespecificada, és a dir, en funció de la implementació i de si el *hardware* permet o no realitzar modificacions en la configuració dels *overlays* mentre hi ha un *scanout* en progrés, aquesta funció bloquejarà o no fins al proper *VBLANK* abans de poder canviar la mencionada configuració. Com a conseqüència d'aquest comportament no definit, no és bona idea utilitzar `drmModeSetPlane` en codi que té com a intenció ser multiplataforma i compatible amb el *hardware* de diferents venedors, amb el consegüentment desaprofitament dels recursos disponibles.

La solució a aquest problema és utilitzar l'API atòmica de *libdrm*: canviar la configuració dels objectes DRM (*CRTCs*, *connectors*, *encoders*, *planes*, etc.) es realitza mitjançant el que s'anomena *object properties*. La idea principal és que cada objecte té una llista de propietats, identificades amb un *string* mitjançant el qual es pot obtenir el *property id* que s'ha d'utilitzar per modificar el valor d'aquella propietat. Donat l'identificador d'un objecte i el seu tipus, la funció `drmModeObjectGetProperties` retorna un *array* amb tots els identificadors de les propietats de l'objecte, i per obtenir el nom de la propietat, el seu valor actual i més informació addicional de cada un d'aquests objectes es crida `drmModeGetProperty`. Cada *driver* de DRM ha d'implementar unes propietats mínimes genèriques per cada tipus objecte[32], i si el *hardware* té funcionalitats addicionals, aquestes es poden exposar com a propietats addicionals de l'objecte, i per tant no cal afegir noves funcions a l'API de DRM.

Les propietats dels objectes són d'un tipus concret, i per tant el significat del seu valor canviarà en funció d'aquest:

- *Range* i *signed range*: Els valors establerts estan dins un màxim i un mínim.
- *Enum*: El valor va des de 0 fins al nombre d'enumeracions definides menys 1.
- *Bitmask*: El valor és un nombre de 64 bits, on cada bit té un propòsit diferent.
- *Object*: S'utilitzen per vincular altres objectes de DRM.
- *Blob*: Guarden l'identificador a un *blob*, és a dir, dades binàries disposades com un *array*.

[32] `https://01.org/linuxgraphics/gfx-docs/drm/drm-kms-properties.html`

Modificar els valors de les propietats requereix el que s'anomena un *atomic commit*: els canvis es van acumulant localment (sense que el *kernel* hi intervingui) en un *struct* de mida variable, i finalment es crida a `drmModeAtomicCommit` perquè DRM ho apliqui d'una sola tacada. Aquesta estructura de dades sobre la qual es van acumulant els canvis és del tipus `drmModeAtomicReq *`, s'obté cridant a `drmModeAtomicAlloc`, i és doncs, el primer paràmetre de la funció `drmModeAtomicAddProperty`, utilitzada per modificar el valor d'una propietat. A la figura 4 hi podem observar aquest procés.

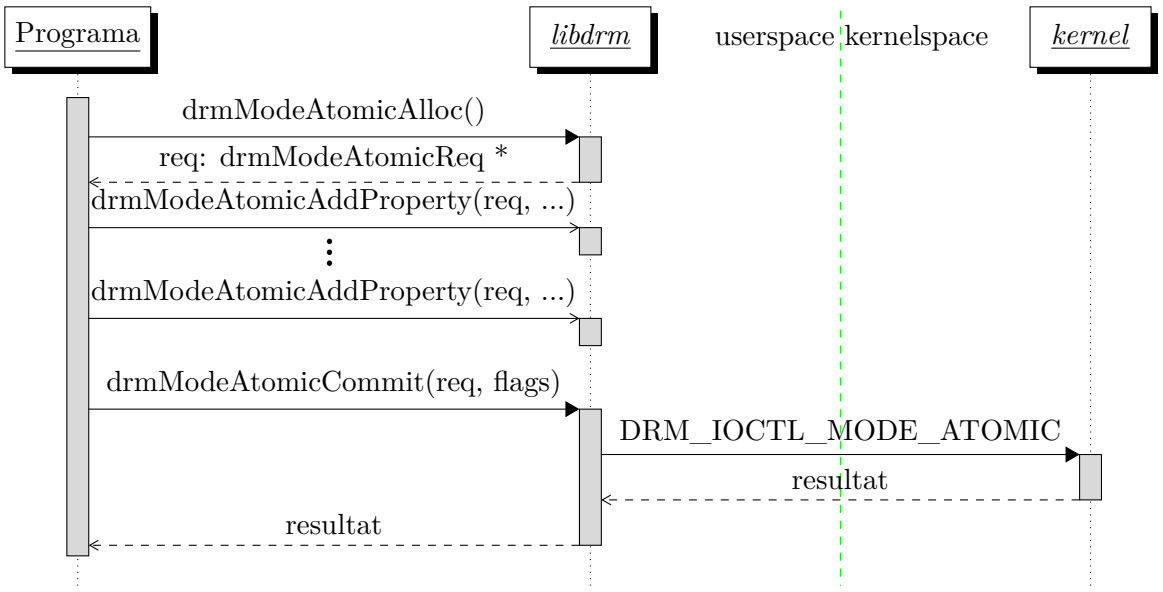


Figura 4: Diagrama de seqüència d'un atomic commit.

La funció `drmModeAtomicCommit` té diversos *flags* importants, que es poden ajuntar amb una *or* binària:

- `DRM_MODE_ATOMIC_ALLOW_MODESET`: indica que es faci un *mode-set*, és a dir, canviar la configuració o mida del *CRTC*, si hi ha canvis en la configuració d'aquest. Aquest *flag* és important, ja que realitzar un *mode-set* pot ser extremadament costós depenent del *hardware*, i per tant només s'hauria de realitzar quan és estrictament necessari (per exemple al canviar la resolució del monitor).
- `DRM_MODE_ATOMIC_TEST_ONLY`: indica que la configuració no s'apliqui i només retorni si el *hardware* seria capaç d'aplicar-la o no, és a dir, si la configuració donada és bona o no. Pot ser útil per anar construint l'estat de mica en mica anant provant si el que estem configurant és valid.

- `DRM_MODE_ATOMIC_NONBLOCK`: fa que la funció retorni instantàniament sense bloquejar fins al proper *VBLANK*.
- `DRM_MODE_PAGE_FLIP_EVENT`: indica que es vol realitzar un *page-flip*.

9.3 X server

El servidor X està dissenyat de forma que és *network transparent*, això vol dir que els clients i servidors no tenen per què estar funcionant en la mateixa màquina. Per a la comunicació utilitza *sockets* TCP/IP entre màquines remotes, i *Unix Domain Sockets* localment quan clients i servidor estan en la mateixa màquina.

Habitualment, els clients no implementen el protocol del servidor X directament, sinó que utilitzen llibreries externes. Tradicionalment s'ha utilitzat Xlib, tot i que ha aparegut una nova llibreria més moderna i lleugera anomenada XCB (*X11 C Bindings*).

Inicialment, X tenia accés exclusiu al *hardware* gràfic, per tant els clients enviaven comandes de renderitzat al servidor, i era aquest qui renderitzava el *framebuffer* (*indirect rendering*). A la que va començar a aparèixer el renderitzat 3D, i en particular l'accelerat per una GPU, aquest sistema d'enviar les comandes es va tornar ineficient i es va passar cap a un model nou anomenat *direct rendering* en el qual els clients tenien accés directe sobre el *hardware* gràfic (en contra del clàssic *indirect rendering*).

Per poder utilitzar *direct rendering* d'OpenGL és requisit que els clients del X server creïn contexts d'OpenGL a les seves finestres, i per aquest motiu es va crear l'extensió GLX[33] (*OpenGL Extension to the X Window System*). Però amb l'arribada d'OpenGL ES i EGL (la nova API genèrica independent per crear contexts accelerats), es va deixar d'utilitzar GLX i es va passar a utilitzar EGL. D'aquesta forma, la manera moderna de crear contexts d'OpenGL i OpenGL ES en X és mitjançant l'extensió `EXT_platform_x11`[34] d'EGL.

La implementació més popular de X és X.Org (abans coneguda com a XFree86).

9.3.1 X.Org

Aquesta implementació està dividida en dues parts:

[33] `https://en.wikipedia.org/wiki/GLX`

[34] `https://www.khronos.org/registry/EGL/extensions/EXT/EGL_EXT_platform_x11.txt`

- *Device Independent X* (DIX): és la part genèrica, la que interacciona amb els clients i du a terme el *software rendering* (renderització per CPU en comptes de per GPU). També realitza el *loop* en els *sockets* pel lliurament d'esdeveniments.
- *Device Dependent X* (DDX): conté els *drivers* a nivell d'usuari dels dispositius d'entrada/sortida, i en particular dels gràfics. Els *drivers* DDX s'anomenen *xf86-video-foo*, on *foo* és el nom del *hardware* o del seu fabricant. Amb la recent aparició del KMS, està sent possible escriure un *driver* genèric anomenat *xf86-video-modesetting*.

9.3.2 *Direct Rendering Infrastructure* (DRI)

La infraestructura composta pel conjunt del servidor X i les seves llibreries per clients, Mesa, i el subsistema DRM del *kernel* s'anomena *Direct Rendering Infrastructure* o DRI, tal i com es mostra a la figura 5.

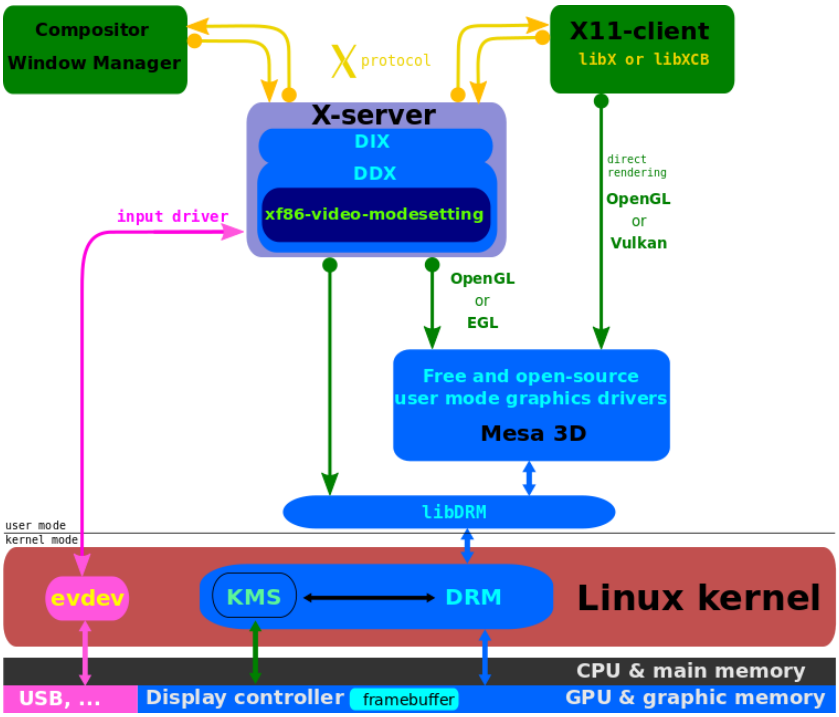


Figura 5: Infraestructura DRI amb el servidor X.

Font: Pròpia, basada en `https://en.wikipedia.org/wiki/File:The_Linux_Graphics_Stack_and_glamor.svg`

9.3.3 Procés de composició

Cap implementació dels *Window Managers*/compositors del servidor X utilitza l'API atòmica de DRM, i per tant aquests estan limitats a utilitzar només el *primary* i com a molt el *cursor plane*. Els clients renderitzen cap al seu propi *framebuffer*, que és enviat cap al compositor del servidor X, i aquest s'encarrega de compositar els diversos clients per generar el *framebuffer* definitiu que es mostrarà mitjançant el *primary plane*, tot realitzant un *page-flip*. A la figura 6 es mostra de forma esquemàtica el procés de composició mencionat.

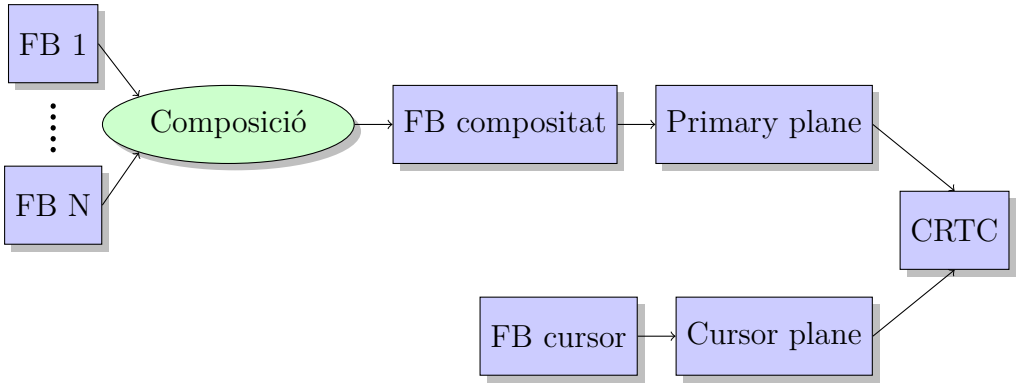


Figura 6: Procés de composició del servidor X.

9.4 Wayland

Sota *Wayland*, igual que en X, els clients i el servidor es comuniquen a través d'un *Unix socket*. El servidor/compositor utilitza la llibreria *libwayland-server*, i els clients utilitzen *libwayland-client* per comunicar-se amb el compositor (anàleg a Xlib i XCB).

A través d'aquest *socket*, les dades s'envien en un format concret descrit per *Wayland*[35]. Aquestes dades són els missatges especificats en els protocols de *Wayland*, descrits mitjançant el format XML[36], als quals s'aplica una eina anomenada *wayland-scanner*, que genera les capçaleres del codi per al client i servidor.

[35] `https://wayland.freedesktop.org/docs/html/ch04.html#sect-Protocol-Wire-Format`

[36] eXtensible Markup Language, permet crear i definir la gramàtica de llenguatges de marques.

El model d'operació està basat en interfícies (*interfaces*), especificades en els protocols, que s'utilitzen per interactuar amb el servidor. Cada interfície proporciona *requests* (missatge del client al servidor), *events* (missatge del servidor al client) i codis d'error. Quan un client s'inicia, el servidor li envia una llista de totes les interfícies disponibles (*global objects*) a través de la interfície `wl_registry`, aleshores aquest pot triar d'utilitzar aquesta interfície cridant a la funció `bind` o no.

El protocol base, descrit en el fitxer `wayland.xml`[37], conté l'especificació de totes les interfícies bàsiques que hauria d'implementar qualsevol compositor de *Wayland*.

A més a més, el projecte *Wayland* també consta d'un repositori anomenat *wayland-protocols*, que conté XMLs de nous protocols addicionals que s'estan creant en conjunt entre diferents entorns d'escriptori populars de Linux (*GNOME*, *KDE*, etc.) per tal que siguin compatibles entre ells, o bé protocols avantguardistes en fase de proves que encara no són estables.

A la figura 7 es mostra una comparació, en forma de diagrama de blocs entre X (esquerra) i *Wayland* (dreta). S'hi pot observar com el nombre de components necessaris a *Wayland* és notablement inferior als necessaris per al servidor X, i en particular com el servidor i el compositor/WM són una mateixa entitat a *Wayland*.

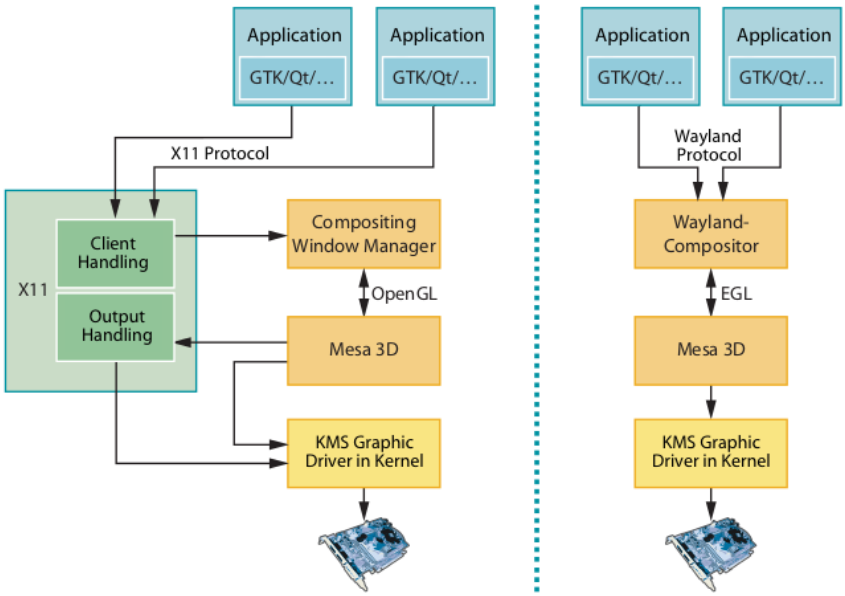


Figura 7: Comparació simplificada entre X11 i Wayland.

Font: http://www.h-online.com/news/item/Wayland-s-1-0-milestone-fixes-graphics-protocol-1734638.html

[37] `https://cgit.freedesktop.org/wayland/wayland/tree/protocol/wayland.xml`

9.4.1 Tipus de dades bàsiques

El *file descriptor* del *socket* que connecta el client i servidor està encapsulat dins el tipus natiu `wl_display`. A partir d'un `wl_display` es poden crear finestres gràfiques, definides pel tipus `wl_surface`. Els continguts d'aquesta superfície són proporcionats pel tipus genèric `wl_buffer`. La implementació més simple d'aquest `wl_buffer` és la que utilitza memòria compartida, obtinguda a partir d'un `wl_shm_pool`, alhora creat pel gestor de *memory pools* `wl_shm`.

Per vincular un `wl_buffer` a un `wl_surface` determinat s'utilitza el *request* `attach` de la interfície `wl_surface`. Aquesta vinculació no és immediata sinó que l'estat de les `wl_surface`s és *double-buffered*, és a dir, es van guardant els canvis en un estat temporal que no s'aplica fins que no s'enviï el *request* `commit`, d'aquesta forma es gaudeix d'actualitzacions atòmiques de l'estat. L'*event* `release` de la interfície `wl_buffer` indica quan el compositor ha deixat d'utilitzar un *buffer*, que per tant està lliure i es pot tornar a utilitzar.

La interfície `wl_shell` és proporcionada pels compositors que implementen interfícies gràfiques d'usuari a l'estil escriptori, tot i que ha quedat obsoleta, ja que les seves capacitats no són suficients per implementar el que es considera un entorn d'escriptori modern.

9.4.2 Infraestructura gràfica

Tot i que pot semblar que s'hagi hagut de llençar tota la infraestructura creada al voltant d'X, i s'hagi hagut de tornar a escriure de nou, el cert és que no s'ha hagut de fer res d'això, i es pot aprofitar quasi tota la infraestructura només realitzant petits canvis. Un d'aquests canvis ha sigut la creació de dues noves extensions d'EGL: `EGL_EXT_platform_wayland`[38], necessària perquè els clients puguin crear contexts gràfics accelerats a *Wayland*, i `EGL_WL_bind_wayland_display`[39] perquè el servidor pugui exposar suport d'aquests contexts gràfics cap als clients.

[38] `https://www.khronos.org/registry/EGL/extensions/EXT/EGL_EXT_platform_wayland.txt`

[39] `https://cgit.freedesktop.org/mesa/mesa/tree/docs/specs/WL_bind_wayland_display.spec`

La implementació d'aquestes extensions depèn completament de la implementació d'EGL. En el cas de Mesa, quan el servidor crida a la funció `eglBindWaylandDisplayWL` de l'extensió `EGL_WL_bind_wayland_display`, aquest exposa una nova *interface* anomenada `wl_drm`[40] cap als clients. Paral·lelament, els clients poden obtenir un objecte `EGLDisplay` cridant a la funció `eglGetPlatformDisplayEXT` de l'extensió `EGL_EXT_platform_wayland`, passant-hi un objecte del tipus natiu de *Wayland* `wl_display`. Posteriorment, per obtenir una superfície de renderitzat (`EGLSurface`) a partir de la finestra de *Wayland*, primer el client ha de crear una nova instància del tipus `wl_surface` (en particular del tipus `wl_egl_surface`, que és un *wrapper* a `wl_surface` incloent la mida de la superfície gràfica, ja que `wl_surface` no conté tal informació, necessària per EGL), i posteriorment ha d'utilitzar la funció `eglCreatePlatformWindowSurfaceEXT`, inclosa a la mateixa extensió.

Anàlogament, l'extensió `EGL_WL_bind_wayland_display` també afegeix un tipus nou a un paràmetre de la funció `eglCreateImageKHR`: `EGL_WAYLAND_BUFFER_WL`. Mitjançant aquest paràmetre el servidor pot crear un `EGLImage` a partir dels `wl_buffer` dels clients (sempre i quan s'hagin creat amb EGL), i d'aquesta forma utilitzar la imatge del client com a textura d'OpenGL.

A la figura 8 hi podem veure el *wrapper* `wl_egl_surface`, representat per la fletxa blava amb dues puntes que va des del *Wayland client* fins a la plataforma EGL *Wayland* de Mesa. Així mateix, també hi podem observar el protocol privat `wl_drm`, utilitzat per la implementació d'EGL de Mesa per crear superfícies gràfiques.

[40] `https://cgit.freedesktop.org/mesa/mesa/tree/src/egl/wayland/wayland-drm/wayland-drm.xml`

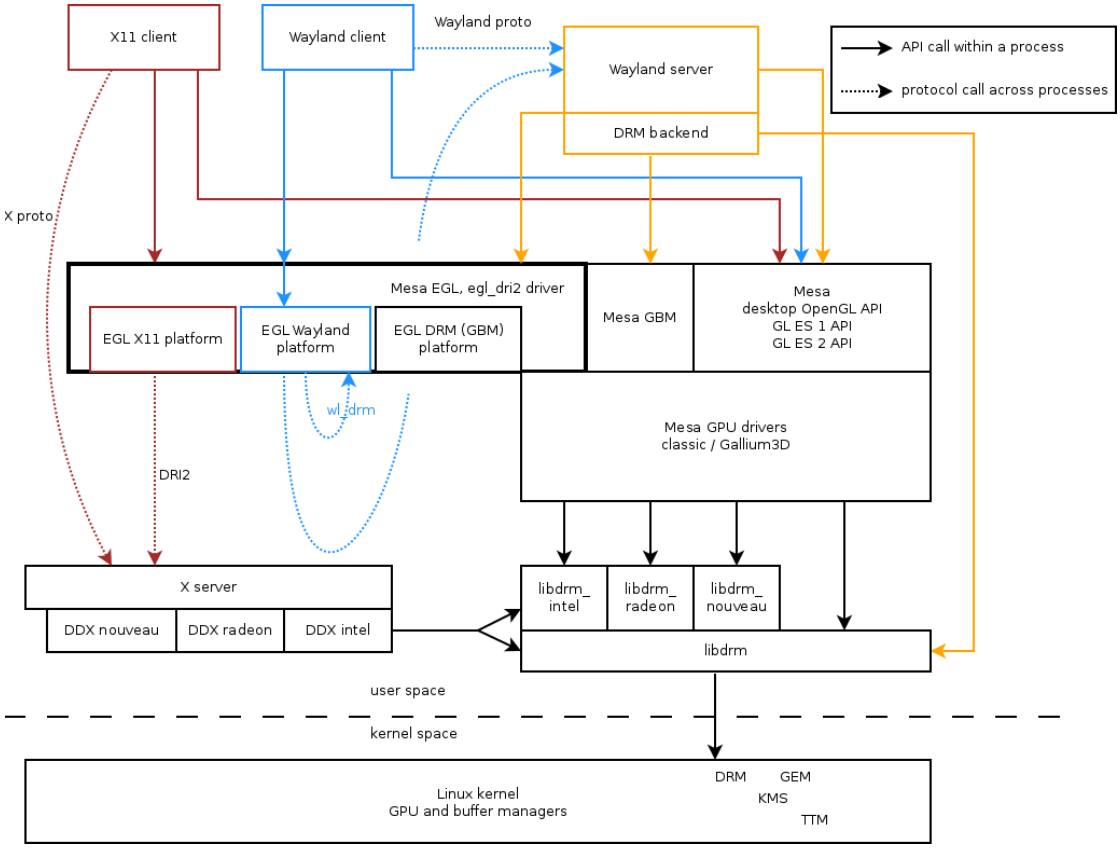


Figura 8: Arquitectura EGL de Mesa a Wayland.

Font: `http://ppaalanen.blogspot.com.es/2012/03/what-does-egl-do-in-wayland-stack.html`

9.5 Weston

A partir de la versió 1.12 de *Weston*[21], el codi d'aquest s'ha separat en dues parts: *libweston*, que conté tota la funcionalitat interna del compositor, i el propi *Weston*, que ara es pot considerar com a un compositor referència de *libweston*, d'aquesta forma ara és molt més fàcil implementar un compositor de *Wayland* mitjançant *libweston*.

Per tal de llegir esdeveniments d'entrada, com per exemple el teclat, ratolí i pantalles tàctils, *Weston* utilitza la interfície *evdev* de Linux. Inicialment *Weston* utilitzava directament aquesta interfície, tot i que es va veure que el codi podria ser útil per a altres compositors (i fins i tot per a X.org), i es va extreure tot aquest codi en una llibreria anomenada *libinput*[41].

El disseny de *Weston* és completament modular, amb la finalitat de poder estendre considerablement les seves capacitats sense haver de tocar el codi central. D'aquests mòduls n'hi ha de tres tipus, i es representen mitjançant el format de les llibreries dinàmiques `.so` de Linux, que es poden carregar mitjançant les funcions `dlsym`[42] del carregador d'enllaç dinàmic: els *backends*, i els *shells* i de forma més genèrica, els *plugins*. A més a més el codi que s'encarrega de realitzar el renderitzat també està modularitzat en el que s'anomenen els *renderers*.

A la figura 9 es mostra com estan interconnectats els conceptes introduïts prèviament, en particular els *backends*, els *shells* i els *renderers*.

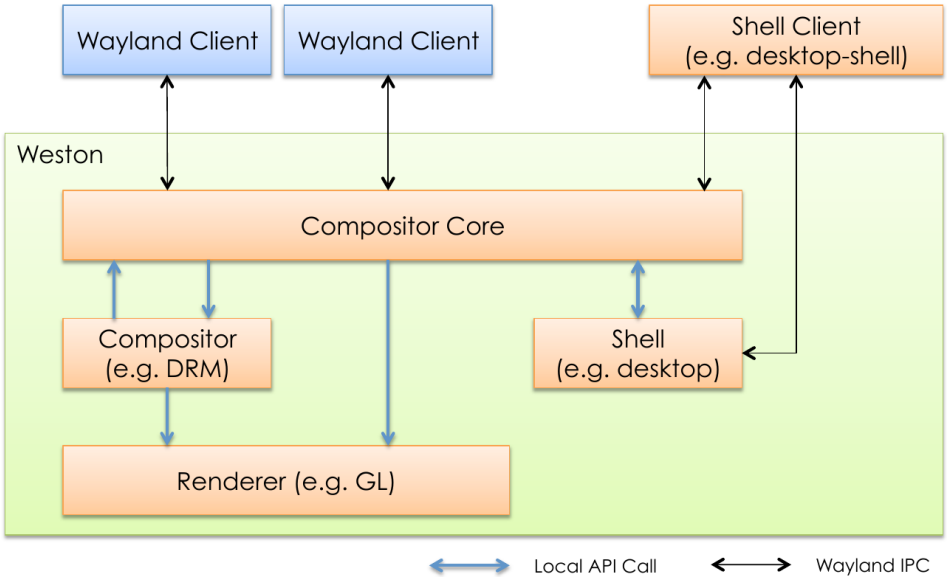


Figura 9: Arquitectura de Weston.

Font: *http://events.linuxfoundation.org/sites/events/files/slides/als2015_wayland_weston_v2.pdf*

[41] `https://www.freedesktop.org/wiki/Software/libinput/`
[42] `http://man7.org/linux/man-pages/man3/dlsym.3.html`

9.5.1 *Backends*

Els *backends* implementen el codi que fa que *Weston* sigui visible gràficament, i aquest en proporciona de tres diferents:

- `drm-backend.so`: Actua sobre DRM i KMS de Linux. Suporta múltiples monitors. És el *backend* principal pensat per a usuaris finals.
- `wayland-backend.so`: Executa *Weston* com a una finestra d'un altre compositor *Wayland*, com per exemple una altra instància de *Weston*.
- `x11-backend.so`: Executa *Weston* com a una finestra del servidor X. Pot ser útil en l'àmbit de desenvolupament de *Weston*.

9.5.2 *Shells*

Els *shells* proporcionen usabilitat pel que fa a l'entorn gràfic (en cert aspecte el que en X seria el *Window Manager*). *Weston* n'inclou tres:

- *Desktop shell*: Similar al que seria un entorn d'X modern, concentrant-se en les interfícies típiques de teclat i ratolí i de gestió de finestres. Aquest *shell* alhora està dividit en dues components que es comuniquen mitjançant un protocol *Wayland* privat: el *plugin* `desktop-shell.so` i el client *weston-desktop-shell* que proporciona un fons d'escriptori, un panell i bloqueig de pantalla
- *Fullscreen shell*: Destinat pels clients que necessiten ocupar tota la pantalla sencera, o fins i tot totes les pantalles.
- *IVI-shell*: *Shell* del tipus *In-vehicle infotainment shell* que exposa la API *GENIVI Layer Manager*[43] en l'àmbit de l'automoció.

9.5.3 *Plugins*

Els *plugins* són mòduls externs al codi principal de *Weston*, i es poden utilitzar per afegir noves funcionalitats.

[43] `https://at.projects.genivi.org/wiki/display/WIE/Wayland+IVI+Extension+Home`

9.5.4 *Renderers*

Un altre component important són els *renderers*, que s'encarreguen de dibuixar l'entorn gràfic del mateix *Weston*, és a dir, les finestres dels clients, el fons d'escriptori, la barra d'eines, etc. D'aquests n'hi ha dos disponibles, i es pot triar quin utilitzar des de la línia d'ordres:

- *gl-renderer*: Utilitza acceleració *hardware* gràfica mitjançant OpenGL, i per tant és el *renderer* preferit que es tria per defecte.
- *pixman-renderer*: Utilitza la llibreria de manipulació de píxels i composició d'imatges a baix nivell i per *software pixman*[44]. També s'utilitzarà com a *fallback* en el cas que el *gl-renderer* no hagi pogut crear un context OpenGL.

Cal destacar que el context OpenGL del *gl-renderer* es crea mitjançant EGL, i en el cas d'utilitzar-se el `drm-backend.so`, la superfície gràfica sobre la qual es renderitzarà serà creada mitjançant la llibreria *libgbm* de Mesa i després utilitzant l'extensió `EGL_MESA_platform_gbm`[45] per tal de crear l'`EGLDisplay` corresponent.

9.5.5 Procés de composició

Abans d'utilitzar l'*atomic mode-setting*, *Weston* realitzava la composició i estava limitat al *primary* i *cursor plane* de la mateixa manera que el servidor X ho està (figura 6). Ara però, emprant l'*atomic mode-setting* (objectiu del projecte) permet utilitzar els *overlay planes*, tal com es mostra a la figura 10, sense pèrdua de rendiment.

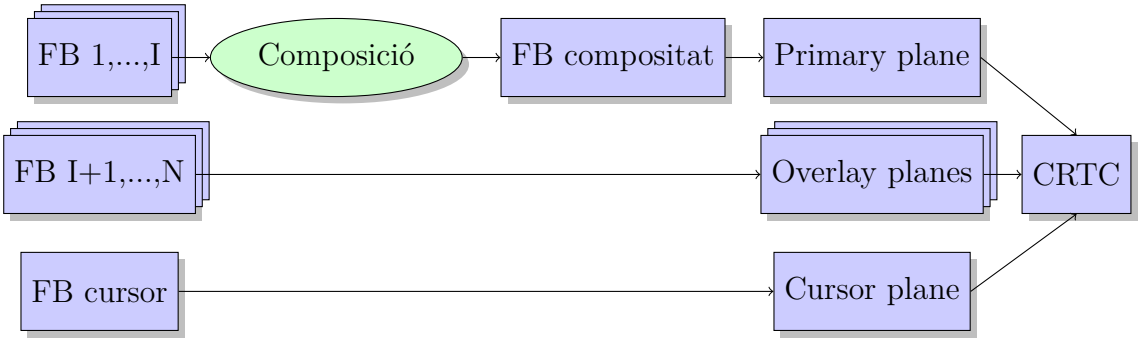


Figura 10: Procés de composició de Weston utilitzant l'API atòmica.

[44] `http://www.pixman.org/`

[45] `https://www.khronos.org/registry/EGL/extensions/MESA/EGL_MESA_platform_gbm.txt`

9.5.6 Estructures de dades

En el codi de *Weston*, cada instància del compositor es representa amb l'estructura `weston_compositor`, que es podria considerar la de més amunt de tot de la jerarquia, i per tant a dins hi trobem les estructures `weston_renderer` i `weston_backend`. Aquestes dues estructures simplement contenen punters a funcions (*vtable* o *dispatch table*) que han d'implementar el *renderer* i *backend* en qüestió, a fi d'obtenir un disseny modular.

A més a més hi trobem que gran part dels objectes principals de *Wayland* estan encapsulats dins d'estructures de dades per tal de facilitar-ne la gestió. Totes les imatges o *buffers* que es mostren es representen amb l'estructura de dades `weston_view`. Cada `weston_view` equival a una instància visible d'una `weston_surface` (que encapsula un `wl_surface`), és a dir, dóna unes coordenades de posició, a quin *output* s'ha de mostrar, entre altres coses, a les superfícies; a la vegada, cada `weston_surface` conté les dades necessàries per poder utilitzar un `weston_buffer` (encapsula un `wl_buffer`).

Un dels atributs principals dels `weston_surface` és el que s'anomena *damage*. Aquest és una regió rectangular o conjunt d'aquestes que engloba tots els canvis visibles que s'han fet sobre el `weston_buffer` respecte a l'última vegada que s'ha utilitzat, i d'aquesta forma es pot optimitzar el renderitzat evitant tornar a pintar regions que no han canviat.

Els `weston_output` representen un monitor, sigui real o virtual (quan s'utilitza un *backend* que no sigui el `drm-backend.so`), i per tant contenen la seva informació: mida, taxa de refresc, posició d'aquest *output* en coordenades globals, i tot de variables d'estat per tal de gestionar-lo.

Ja que *Weston* utilitza la llibreria *pixman* en el *pixman-renderer*, i aquesta a més a més de dibuixar també exporta una sèrie de tipus de dades i funcions per la manipulació de regions i conjunts de regions rectangulars (tipus `pixman_region32_t`), les estructures de dades definides per *Weston* que necessiten manipular regions rectangulars utilitzen les de l'API de *pixman*.

10 Especificació

El subsistema DRM de Linux s'ha anat adaptant i actualitzant al llarg del temps, en funció de les noves millores del *hardware* que ha anat sortint al mercat. Però tot i això, la tecnologia dels controladors de vídeo i GPUs ha evolucionat més de pressa que no pas el suport *software* d'aquestes novetats, cosa que ha comportat que tot i que ja fa alguns anys que el mateix *hardware* disposa de circuits per rellevar tasques que normalment faria la CPU, aquests, com per exemple els *hardware planes*, no són utilitzats per la majoria d'entorns d'escriptori d'X o fins i tot pel mateix *Weston*.

La voluntat dels desenvolupadors d'utilitzar *atomic mode-setting* a Weston ja fa alguns anys que persisteix, però aquest nou mode d'utilitzar el subsistema gràfic requereix les últimes versions del *kernel* Linux i funcionalitats i canvis que encarà s'estan discutint, i que per tant avui en dia en la majoria de *drivers* de GPUs i controladores de vídeo no estan implementades. Per exemple, està previst que el *driver* de les GPUs integrades dels processadors de l'empresa Intel (anomenat i915) activi l'*atomic mode-setting* per defecte a la versió 4.12 de Linux[46], que es llençarà oficialment a principis del juliol del 2017. El desenvolupador Daniel Stone de l'empresa Collabora[47] és un dels contribuïdors principals a la part dels gràfics de Weston i en l'*stack* gràfic de Linux en general, i en el moment de començar l'activitat d'optimització de *Weston*, ja anava per l'onzena versió d'una branca a *git* que introduïa suport primitiu per *atomic mode-setting*[48]. Per tant, tots els canvis i millores es realitzaran sobre aquesta branca, ja que és la que conté els últims canvis en relació a l'API atòmica de DRM.

Durant la part de familiarització del codi de *Weston* i proves amb aquest, vaig trobar diversos *bugs* de menor importància i un que realment era seriós, així que vaig decidir que solucionar en especial aquest *bug* seria part de la implementació del projecte.

A més a més de solucionar aquest *bug* important, vaig acordar amb els desenvolupadors de *Weston* realitzar dues millores importants: fer que s'utilitzi el *pixel format modifier blob* dels *hardware planes*, i afegir suport bàsic de sincronització explícita entre productor i consumidor de *buffers* gràfics (*explicit fencing*).

[46] `https://www.phoronix.com/scan.php?page=news_item&px=Intel-Atomic-Default-Change`

[47] `https://www.collabora.com/`

[48] `https://git.collabora.com/cgit/user/daniels/weston.git/log/?h=wip/2017-04/atomic-v11-WIP`

Així doncs, les millores i optimitzacions realitzades a *Weston* al llarg del projecten tenen com a objectiu modernitzar l'ús que aquest en fa sobre DRM, en particular relacionades amb la utilització de l'*atomic mode-setting*, amb la finalitat d'aprofitar millor els recursos disponibles del sistema, amb una conseqüent millora de l'experiència de l'usuari i estalvi d'energia.

11 Disseny

En els següents apartats s'explica com es realitzaran les optimitzacions de *Weston* i *atomictest*, nom donat al programa de prova creat per familiaritzar-me amb els conceptes de DRM i *libdrm*.

11.1 Programa de prova *atomictest*

En primer lloc es realitzarà la implementació mitjançant l'API antiga (en altres paraules, no atòmica) de *libdrm*, i posteriorment es canviaran les crides a les funcions de l'API no atòmica per les noves crides atòmiques. Això hauria de comportar que no hi hagi cap mena de defecte visual ni pèrdua de rendiment quan s'utilitzin els *overlays*.

La idea principal és obrir un context DRM de la targeta gràfica, buscar el camí *CRTC* $\rightarrow$ *Encoder* $\rightarrow$ *Connector* actual (existent si ja hi ha algun altre context DRM actiu, com per exemple el servidor X o un compositor *Wayland*) o un que sigui compatible, i utilitzar el *primary plane* per pintar un fons de pantalla, el *cursor plane* per pintar una imatge que es pugui moure amb el ratolí, tot utilitzant *libinput* per llegir els esdeveniments d'aquest, i utilitzar tots els *overlay planes* que siguin compatibles amb el *CRTC* triat.

Utilitzar l'API no atòmica inicialment tindrà els següents inconvenients principals, que seran resolts en passar-ho a atòmic:

- Actualitzacions de l'estat dels *planes* no atòmica, fet que pot comportar defectes visuals o desfasament entre l'entrada dels dispositius d'interacció, el ratolí, i la retroalimentació gràfica que aquests haurien de comportar.
- La funció utilitzada per establir la posició i mida dels *overlay planes* no està completament especificada, i per tant cada implementació pot resultar en comportaments diferents: en algunes cridar la funció és bloquejant fins al proper *VBLANK* cosa que comporta una reducció dels *FPS* (fotogrames per segon) resultants molt important a mesura que s'utilitzen *overlay planes*, i en altres el canvi es realitza instantàniament, cosa que de nou comporta defectes visuals (*tearing*).

11.2 Millores i optimitzacions de *Weston*

Com ja s'ha explicat anteriorment, s'han realitzat dues millores al codi font de *Weston*, que s'expliquen seguidament.

11.2.1 Utilització del *pixel format modifiers blob*

Les dades guardades en un *buffer* gràfic estan representades mitjançant un format de píxel. El format més habitual acostuma a ser una de les diferents variants de RGB (*red-green-blue*), com per exemple el RGBA8888, on cada píxel es codifica mitjançant quatre components de 8 bits (1 byte) cada una, tres de les quals indiquen el color, i la quarta la transparència (*alpha).* En cas de dispositius amb escassa memòria o baix amplada de banda d'accés a memòria, s'utilitzen formats en els quals els píxels ocupen menys espai, RGB565 n'és un clar exemple.

Això porta a un problema, i és que en dispositius moderns on els diversos components del *pipeline* gràfic de l'ordinador els han fet diversos fabricants, és possible que el productor de *buffers* gràfics sigui capaç de produir-ne en uns formats de píxel concrets, i que el consumidor només sigui capaç de llegir-ne (per produir el senyal elèctric cap al monitor, per exemple) uns altres.

A més a més, i sobretot impulsat per l'auge dels dispositius mòbils i sistemes encastats, s'han buscat noves maneres d'organitzar els píxels per tal de millorar la taxa d'encerts a les *caches* de les GPUs. Usualment, quan es té una imatge 2D (representat com un *array* unidimensional en memòria) de $width \times height$ píxels, l'índex de l'*array* per accedir a l'element 2D (x, y) es calcula com: $x+y \times height$. Si estem recorrent la imatge en una direcció (d'esquerra a dreta), estarem aprofitant la localitat espacial que les *caches* tant saben aprofitar. En canvi serà molt ineficient si accedim a un píxel d'una fila diferent, és a dir, accedint verticalment (patró utilitzat en el filtre bilineal). Per alleujar aquesta ineficiència, les dades gràfiques es poden guardar en formats que no siguin lineals en memòria, com per exemple en el que s'anomena *swizzling* i *tiling.*

El *tiling* consisteix en dividir la imatge en blocs més petits (anomenats *tiles*). Cada tile té la mida que fa que hi càpiga en una pàgina (habitualment 4KiB) o línia de *cache*, i per tant redueix la pèrdua de rendiment que es podria ocasionar a l'accedir a pàgines de memòria o línies de *cache* diferents quan s'accedeixen a píxels d'un mateix *tile.* A la figura 11 hi podem observar com estan organitzats els *tiles* en els dos formats de *tiling* més habituals en arquitectures d'Intel.

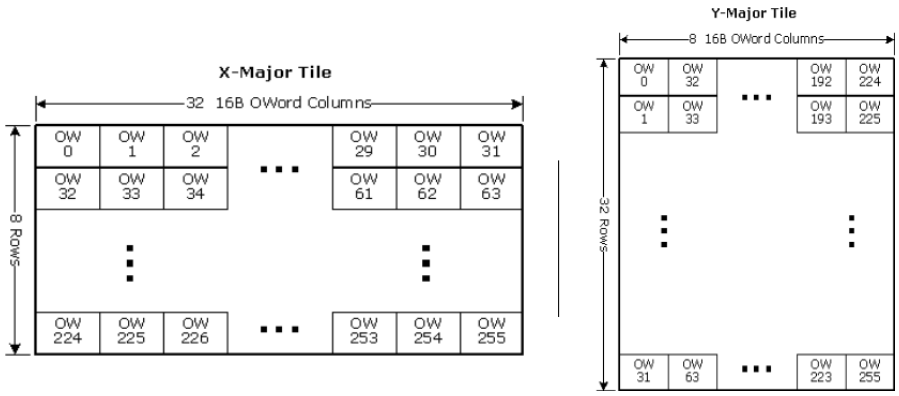


Figura 11: Formats de tiling d'Intel.

Font: ***https://01.org/sites/default/files/documentation/intel_os_gfx_prm_vol5_-_memory_views_1.pdf***

Per altra banda, el *swizzling* consisteix en intercalar els bits de la coordenada x amb els de la coordenada y de tal forma que un nombre serà proper a un altre si aquests dos representen coordenades (x, y) properes. El patró més utilitzat en *swizzling* és el que segueix el format de la corba *Z-order*, o també anomenada *Morton order*[49].

En el subsistema DRM al *swizzling* i *tiling* se'ls anomena *modifiers*, ja que en cert sentit, estan modificant el format amb el qual es representen els píxels en memòria.

Així doncs, els *hardware planes* es consideren consumidors o lectors de *buffers*, i per tant s'ha de tenir en compte el fet que el *hardware* que hi ha al darrere d'aquests *planes* només sigui capaç de manipular uns formats de píxel concrets, i cada format d'aquests pot suportar cap, o sigui representació lineal, o diversos *format modifiers*.

Per tal d'exposar els formats i els seus modificadors suportats per cada *hardware plane*, un enginyer d'Intel va proposar afegir un nou `ioctl()` al subsistema DRM[50], tot i que al final no es va acabar acceptant i es va preferir que aquesta informació s'exposés com a una propietat del tipus *blob*, i que per tant ja es pot obtenir mitjançant un `ioctl()` ja existent sense haver d'afegir-ne cap de nou.

[49] `https://fgiesen.wordpress.com/2011/01/17/texture-tiling-and-swizzling/`
[50] `https://patchwork.kernel.org/patch/9482395/`

Després de discutir-ho amb els desenvolupadors de DRM, vaig proposar el format de l'*struct* d'aquest blob i hi va haver conveni, fins al punt que aquest enginyer va enviar nous *patches*[51] als desenvolupadors de DRM utilitzant aquest nou format. És essencial disposar de quins formats de píxel i modificadors suporta cada *hardware plane*, ja que quan s'hi vol assignar un *buffer*, el format d'aquest ha de ser compatible amb els suportats pel *plane*.

La millora consisteix doncs en dos punts:

1. Obtenir i guardar els formats de píxel i cada un dels seus modificadors suportats per cada *hardware plane*.
2. Estendre la comprovació de la compatibilitat entre el *buffer* i el *hardware plane* mitjançant els modificadors de píxel suportats per cada un.

11.2.2 Utilització d'*explicit fencing*

Actualment, el *kernel* de Linux utilitzat per l'ús d'escriptori habitual i en concret el seu subsistema DRM realitzen el que s'anomena sincronització implícita/*implicing fencing*.

Això significa que el *kernel* realitza un seguiment implícit, sense involucrar-hi el *userspace* i sense que aquest se n'adoni, de tots els *buffers* gràfics reservats (*allocated*) i les dependències que hi pugui haver entre aquests. Internament cada *buffer* té assignat doncs, una *fence* o barrera que s'utilitzarà per gestionar els accessos de lectura i escriptura i evitar condicions de carrera.

Un exemple molt clar és la tasca que realitza el compositor quan ha d'utilitzar el *renderer* per compositar una sèrie de clients accelerats per *hardware*: cada client utilitza una API gràfica accelerada per dibuixar el contingut cap a un *buffer* propi, després quan el *buffer* s'envia cap al compositor, al seu torn aquest utilitzarà també una API gràfica per utilitzar els diversos *buffer*s dels clients com si fossin textures, i per tant generar la imatge final que es mostrarà per pantalla. Aquesta dependència de lectura-escriptura, on es pot dir que clients escriuen i el compositor llegeix, requereix certa sincronització per evitar que el lector llegeixi dades que encara s'estan escrivint, i per aquest motiu, quan el *kernel* veu que un *job*/*batch buffer* que ha d'enviar a la GPU depèn d'un altre encara en procés, insereix una barrera.

[51] `https://lists.freedesktop.org/archives/dri-devel/2017-May/140500.html`

L'*implicit fencing* comporta dos problemes molt importants que fan que s'estigui treballant per evolucionar la infraestructura de DRM cap al *explicit fencing*. Un d'aquests problemes és la incapacitat de l'*implicit fencing* de gestionar les dependències entre el productor i el consumidor d'un *buffer* quan aquests dos són dispositius DRM diferents, cas molt habitual avui en dia degut a l'auge dels sistemes amb múltiples GPUs (també coneguts com a *PRIME* en el context de DRM), fet que comporta condicions de carrera en accedir a dades invàlides. L'altre problema és que si un client triga massa a renderitzar, per exemple perquè el seu *workload* és molt gran, quan el compositor intenti utilitzar el seu *buffer*, tot ell es bloquejarà esperant que el client acabi, cosa que pot arribar a resultar molt molesta pels usuaris.

Així doncs, la sincronització explícita funciona de la següent manera: quan un programa renderitza mitjançant una API gràfica, s'envia un *batch buffer* cap al *kernel* perquè passi a executar-lo la GPU, un cop s'ha fet *submit* d'aquest *batch buffer*, immediatament el *kernel* retorna un *file descriptor* que representa la barrera de finalització de la tasca associada, i per tant aquesta barrera es pot enviar de nou a DRM perquè s'esperi que s'activi la barrera abans de realitzar l'operació corresponent (simulant el comportament d'*implicit fencing*) o consultar l'estat de la barrera, i en cas que encara no s'hagi activat i que per tant el *job* que el client ha enviat a la GPU encara no s'ha completat, utilitzar el *buffer* previ disponible que enviat pel client, tot evitant que els bloqueigs generats pel client repercuteixin al compositor.

A més a més de permetre sincronitzar dos dispositius DRM diferents i d'evitar que els clients puguin bloquejar el sistema completament, la sincronització explícita gaudeix de molts altres avantatges que fan que sigui imprescindible si es vol modernitzar i optimitzar l'ús de tota la pila gràfica del sistema operatiu. Alguns dels avantatges més importants es mostren a continuació:

- Menys variació entre el comportament dels *drivers* de diferents dispositius: com que la sincronització és explícita, dóna menys lloc que uns controladors insereixin bloquejos en llocs on altres controladors no n'insereixen i viceversa.
- Permet processament paral·lel de *suballocations*: És habitual reservar un *buffer* molt gran i realitzar particions dins d'ell per realitzar tasques diferents paral·leles. Amb l'*explicit fencing* cada *sub-buffer* pot obtenir una *fence* pròpia i per tant és possible realitzar-hi treball paral·lel sense bloquejar les altres tasques.
- Millora la traçabilitat i la 'debugabilitat': en tot moment podem consultar l'estat d'una *fence* i per tant podem saber en quin estat es troben els *buffers* gràfics.

El sistema operatiu per telèfons mòbils Android des de fa temps requereix que la infraestructura gràfica suporti *explícit fencing*[52], i per aquest motiu es van afegir extensions i funcions a EGL per poder convertir una *fence* d'EGL al tipus de *fence* natiu d'Android encapsulat dins un descriptor de fitxer, i a l'inrevés. Molt recentment s'està intentant dotar la versió genèrica de Linux de suport de l'*explicit fencing*[53], reaprofitant moltes de les idees i conceptes apresos en Android, i fins i tot les extensions d'EGL d'Android necessàries per a la sincronització explícita s'han implementat a Mesa.

En conjunt, la millora que necessita *Weston* per utilitzar l'*explicit fencing* satisfactòriament consta del següent:

1. Afegir un protocol/interfície nova a *Wayland* que permeti enviar *fences* en forma de descriptors de fitxer des dels clients fins al compositor.
2. Utilitzar les *fences* dels clients per tal de:
    - Modificar *Weston* perquè esperi que les *fences* enviades pels clients s'hagin activat abans d'utilitzar els *buffers* enviats per aquests, amb la finalitat d'evitar completament els possibles bloqueigs causats per clients lents.
    - Col·locar les *fences* enviades pels clients als punts de sincronització de les *fences*, que són la propietat `IN_PLANE_FD` dels *hardware planes* i la crida `eglWaitSyncKHR` d'EGL, per tal de suportar sistemes on el productor i el consumidor d'un *buffer* són dispositius diferents.

Tot i que l'ideal seria poder implementar tots els punts descrits prèviament, l'abast de la implementació d'aquesta millora no incorpora el punt d'utilitzar les *fences* enviades pels clients als punts de sincronització, ja que per poder comprovar que funciona, és necessari desactivar l'*implicit fencing*, i per tant s'ha de modificar la implementació d'EGL de Mesa (afegint una extensió per exemple). També necessitaria un sistema amb múltiples dispositius DRM, del qual no disposo.

[52] `https://source.android.com/devices/graphics/#synchronization_framework`
[53] `https://lwn.net/Articles/685049/`

D'aquesta manera, la millora que es realitzarà sobre *Weston* consisteix en el següent: en primer lloc s'afegirà un protocol/interfície nova a *Wayland* que permeti enviar *fences* en forma de descriptors de fitxer des dels clients fins al compositor; en segon lloc, es copiarà el programa de prova de *Weston* `weston-simple-egl` en un de nou fent que utilitzi les extensions d'EGL d'Android disponibles a Mesa per tal d'obtenir una *fence* nativa i que utilitzi el nou protocol per enviar-la cap al compositor; i en tercer lloc, es modificarà *Weston* perquè esperi que les *fences* enviades pels clients s'hagin activat abans d'utilitzar els *buffers* enviats per aquests, amb la finalitat d'evitar completament els possibles bloqueigs causats per clients lents.

12 Implementació

En aquesta secció es mostra el detall tècnic de com funciona el programa de prova *atomictest*, de com s'han dut a terme les dues optimitzacions realitzades, i de com s'ha solventat l'error que hi havia en un dels algorismes de *Weston*.

12.1 Programa de prova *atomictest*

La implementació d'aquest programa consta de dues parts: primer utilitzant la API antiga, i posteriorment utilitzant la API atòmica.

12.1.1 Codi inicial no atòmic

En primer lloc hem d'obrir el node de la targeta gràfica (`/dev/dri/card0`). Seguidament obtenim les capacitats i informació de tots els recursos disponibles utilitzant `drmGetCap` i `drmModeGetResources`. D'aquesta forma podem iterar sobre cada un dels *connectors*, mirant que estigui connectat, i per cada un d'aquests mirem si té algun *encoder* connectat. Si ja en té un, i aquest està alhora connectat a un *CRTC*, ja hem trobat el camí *CRTC* $\rightarrow$ *Encoder* $\rightarrow$ *Connector* que utilitzarem, en cas contrari, iterem sobre tots els *encoders* compatibles amb el nostre *connector*, i realitzem un altre bucle niuat que iteri sobre tots els *CRTCs* fins que en trobem un que sigui compatible amb l'*encoder*. En tots dos casos ens guardem la mida del monitor, trobada en l'*struct* d'informació del *connector*. Necessitem les funcions `drmModeGetCrtc`, `drmModeGetConnector` i `drmModeGetEncoder` per aquesta tasca.

En aquest punt ja tenim els identificadors del *CRTC*, el *connector* i l'*encoder* que utilitzarem, i per tant és un bon moment per obtenir tota la llista de *planes* disponibles en el dispositiu. Utilitzem `drmModeGetPlane` per obtenir la informació de cadascú d'aquests, però abans de guardar-la, s'ha de comprovar que sigui compatible amb el nostre *CRTC*. El següent pas que fem és reservar memòria per als diferents components gràfics mitjançant *dumb-buffers*: en reservem un per al cursor, dos per el *primary plane* (*double buffering*) i un per cada *overlay plane*.

Abans de realitzar el *mode-setting*, és a dir, aplicar la nostra configuració, és bona idea guardar-nos la configuració actual per tal de poder-la restaurar quan el programa acabi. Utilitzem `drmModeGetCrtc` per guardar-nos l'estat actual, i procedim a aplicar la nostra configuració amb la crida `drmModeSetCrtc`.

L'esdeveniment de DRM que ens interessa és el de notificació de la finalització del *page-flip*, ja que quan aquest passi podem dibuixar el següent *frame* i posar un nou *page-flip* a la cua. Per realitzar l'espera passiva bloquejant utilitzem `poll`, i posteriorment, quan aquesta desperti, li passarem la direcció del nostre *handler* a l'atribut `page_flip_handler` de l'*struct* a `drmHandleEvent`.

Abans d'entrar en el *loop* principal, que es dedica a fer la crida a `poll` mencionada anteriorment de forma constant, hem de posar a la cua un *page-flip* inicial utilitzant la funció `drmModePageFlip`.

Si tot ha anat bé, ja tenim la configuració aplicada, el primer *page-flip* encuat i estem esperant que la notificació d'aquest arribi. Un cop hagi arribat s'executarà el nostre *handler* i podrem procedir a re-dibuixar els *dumb-buffers*, resituar el *cursor plane* (crides `drmModeSetCursor` i `drmModeMoveCursor`), resituar els diversos *overlay planes* (`drmModeSetPlane`) i de nou tornar a demanar un *page-flip*.

12.1.2 Utilització de l'API atòmica

Per passar el codi a les funcions atòmiques, l'estructura del codi no ha de canviar gaire, i moltes de les funcions que s'utilitzaven abans se segueixen utilitzant. El canvi més destacat està en la forma d'establir l'estat dels *CRTCs*, els *connectors* i els *planes* i la forma de demanar un *page-flip*, que ara s'utilitzen propietats dels objectes que es van acumulant i s'apliquen al realitzar el *atomic commit*.

D'aquesta manera, l'anterior crida a `drmModeSetCrtc` la podem canviar per un *commit* atòmic amb els *flags* `DRM_MODE_ATOMIC_ALLOW_MODESET`. Aquest *commit* només es realitza un cop a l'inici del programa per aplicar el *pipeline CRTC* $\rightarrow$ *Encoder* $\rightarrow$ *Connector*. En comptes d'utilitzar `drmModeSetPlane` per configurar els *overlays* i `drmModeMoveCursor` per moure el *cursor plane*, ho acumulem amb `drmModeAtomicAddProperty` i canviem la crida a `drmModePageFlip` per una a `drmModeAtomicCommit`, tot passant-hi els nostres canvis acumulats i `DRM_MODE_ATOMIC_NONBLOCK | DRM_MODE_PAGE_FLIP_EVENT` com a *flags*.

12.2 Utilització del *pixel format modifiers blob*

L'`ioctl()` proposat inicialment i la seva contrapart a *libdrm* s'anomenaven `DRM_IOCTL_MODE_GETPLANE2` i `drmModeGetPlane2` respectivament. La idea era que `DRM_IOCTL_MODE_GETPLANE2` retornés una estructura de dades amb la mateixa informació que la que retorna `DRM_IOCTL_MODE_GETPLANE` però afegint-hi més atributs al final: els modificadors del format de píxel suportats per un *plane* donat.

Finalment, però, el que està prosperant és afegir una propietat del tipus *blob* anomenada `IN_FORMATS` als objectes *plane*, que conté la llista de formats de píxel i modificadors suportats. El format d'aquest *blob*, que podem observar a la figura 12, està fortament basat en el que s'havia proposat inicialment amb `DRM_IOCTL_MODE_GETPLANE2`.

Una millora important que s'ha fet és afegir suport d'un modificador a un nombre il·limitat de formats, ja que anteriorment s'utilitzava una màscara de 64 bits, on el bit i-èsim indica si el format i-èsim de la llista de formats suportats és compatible amb el modificador, i per tant com a màxim 64 formats podien suportar un modificador donat. Aquesta millora s'ha realitzat afegint una finestra lliscant a la màscara de bits (atribut `offset` de `drm_format_modifier`), d'aquesta manera per poder exposar suport a més de 64 formats de píxel és tan simple com afegir elements `drm_format_modifier` duplicats a la llista de modificadors tot canviant el desplaçament de la finestra i la màscara de bits en qüestió.

L'estructura del *blob* exposat a l'espai d'usuari (`drm_format_modifier_blob`) s'ha versionat tant amb un identificador de versió, com amb un atribut `flags`, no utilitzat ara per ara. Per tal de suportar un nombre variable il·limitat de formats i de modificadors, els atributs `formats_offset` i `modifiers_offset` contenen a quina distància des de l'inici de l'*struct* es troben els *arrays* corresponents, i tal com el seu nom indica, `count_formats` i `count_modifiers` indiquen quants elements n'hi ha de cada.

```
struct drm_format_modifier {
       uint64_t formats;
       uint32_t offset;
       uint32_t pad;
       uint64_t modifier;
} __attribute__ ((__packed__));

struct drm_format_modifier_blob {
        uint32_t version;
        uint32_t flags;
        uint32_t count_formats;
        uint32_t formats_offset;
        uint32_t count_modifiers;
        uint32_t modifiers_offset;
} __attribute__ ((__packed__));
```

Figura 12: Estructura d'un modificador de format de píxel i del blob.

A *Weston*, els *planes* s'encapsulen en una estructura de dades anomenada `drm_plane`. Quan el `drm-backend.so` s'inicialitza, es crea una nova estructura d'aquestes per cada *plane* existent en el *hardware*, i posteriorment s'afegeix a una llista que el compositor té. En el moment de crear l'estructura, es demana a DRM una llista de totes les propietats i informació que cada *plane* determinat té, i es guarden aquestes dades per tal de tenir-les accessibles fàcilment sense haver d'efectuar una crida a *libdrm* cada vegada que es vulguin consultar. Així doncs, el primer canvi que s'ha de realitzar és demanar-li a DRM que retorni el *blob id* de la propietat identificada amb l'*string* `IN_FORMATS`.

Un cop s'ha obtingut l'identificador del *blob*, es pot cridar a la funció `drmModeGetPropertyBlob` per tal d'obtenir les seves dades. Com que els *blobs* són dades binàries crues, s'ha de fer *casting* explícitament a un punter a `drm_format_modifier_blob` per poder utilitzar-lo com a tal. La informació dels formats de píxel suportats i dels modificadors de cada un també es repliquen a la mateixa estructura `drm_plane` per tenir-hi un accés fàcil, tal com es pot observar a la figura 13. Replicar aquestes dades no és tan senzill com realitzar un simple `memcpy` ja que les estructures són completament diferents. En primer lloc, es recorren tots els formats reportats pel *blob*, en segon lloc, en un bucle interior es recorren tots els modificadors reportats i finalment es mira si el modificador actual és compatible amb el format actual; si ho és, s'afegeix el modificador a la llista dels suportats pel format de forma dinàmica utilitzant `realloc`, en cas contrari es passa al següent modificador.

```
struct drm_plane {
        /* ... */
        struct {
                uint32_t format;
                uint32_t count_modifiers;
                uint64_t *modifiers;
        } formats[];
};
```

Figura 13: Canvis a l'estructura `drm_plane` al afegir modificadors de format de píxel.

Cal destacar que en cas que el *blob* `IN_FORMATS` dels *planes* no existeixi (si s'utilitza una versió de Linux massa antiga, per exemple), la informació dels modificadors suportats de l'estructura de `drm_plane` de *Weston* s'inicialitza només amb els formats de píxel reportats a l'estructura `drmModePlane *` obtinguda al cridar a `drmModeGetPlane`, i per tant representa que els formats no suporten cap modificador, cosa que pot comportar que si un client de *Wayland* dibuixa en un format que utilitza modificadors, aquest no es podrà posar directament en un *hardware plane*, sinó que s'hauria de renderitzar manualment cap al *primary plane* utilitzant el *renderer*.

Ara que ja tenim la llista de formats i modificadors suportats per cada *plane*, hem d'actualitzar la funció que comprova si el format d'una `weston_view` és compatible amb un *overlay plane* donat, i que s'utilitza en l'algorisme que assigna les diferents `weston_view`s als *plane*s. Efectuar aquesta comparació és relativament senzill, ja que només s'ha de realitzar dos bucles niuats que iterin sobre els formats i els modificadors del *plane* i que comprovin si algun d'aquests coincideix amb el format del *framebuffer* de la `weston_view`, tal com podem veure en l'algorisme de la figura 14.

Require: *plane*, el *plane*
Require: fb, el *fb*
  function ÉS FB COMPATIBLE AMB PLANE(*plane*, fb)
    foreach $f \in plane.formats$ **do**
      if $f = fb.format$ **then**
        foreach $m \in f.modifiers$ **do**
          if $m = fb.modifier$ **then**
            return true
          end if
        end for
      end if
    end for
    return false
  end function

Figura 14: Algorisme que retorna si el format d'un framebuffer és compatible amb un plane.

12.3 Utilització d'*explicit fencing*

El primer pas per afegir suport d'*explicit fencing* és disposar d'un protocol *Wayland* que permeti enviar *fences* en format de descriptor de fitxers des dels clients cap al compositor.

La capa d'abstracció que permet executar aplicacions d'Android sobre el sistema operatiu basat en Linux de Google Chrome OS, anomenada *ARC++*, utilitza *Wayland*[54] per dur a terme la comunicació gràfica i de dispositius d'entrada i la gestió de finestres entre la infraestructura d'Android i el mateix Chrome OS. A més a més d'utilitzar algun dels protocols definits en el repositori oficial *wayland-protocols*, *ARC++* també n'ha hagut de crear de nous per poder obtenir noves funcionalitats, sent una d'aquestes l'*explicit fencing*[55]. Així doncs, en comptes de crear el meu propi protocol, he decidit utilitzar aquest, i així m'asseguro que funciona correctament, ja que ha estat altament provat pels mateixos enginyers de Google sense que hagin sorgit problemes.

Aquest protocol és bastant simple i el funcionament per enviar una *fence* cap al compositor és molt senzill: el compositor exposa una nova interfície, `zcr_linux_explicit_synchronization_v1` que els clients poden enllaçar; aquesta interfície disposa d'un *request* anomenat `get_synchronization` que a partir d'una `wl_surface` crea una nova instància d'una altra interfície: `zcr_synchronization_v1`, que alhora disposa d'un mètode anomenat `set_acquire_fence` que té com a paràmetres el descriptor de fitxer de la *fence* que es vol enviar cap al compositor.

Per tal d'exposar aquest nou protocol cap als clients, en primer lloc s'ha d'afegir el *XML* que descriu el protocol a la llista de fitxers que *wayland-scanner* ha d'executar, per tal de generar les capçaleres d'aquest. Seguidament s'ha hagut d'exposar la interfície global `zcr_linux_explicit_synchronization_v1` cap els clients mitjançant la funció `wl_global_create` i passant-hi com a paràmetre les estructures de dades generades pel *wayland-scanner*, i un *struct* de punters a funcions que implementin els *requests* del protocol.

Com que *Weston* encapsula les `wl_surface`s en el tipus propi `weston_surface`, i les instàncies de la interfície `zcr_synchronization_v1` també es creen a partir d'una `wl_surface`, he afegit un nou atribut anomenat `explicit_sync_resource` del tipus `wl_resource` a la estructura `weston_surface` que guarda aquestes instàncies. Així doncs, quan un client crida `get_synchroniztion` per tal d'obtenir una nova instància del tipus `zcr_synchronization_v1`, el primer que faig és mirar si ja hi ha un `explicit_sync_resource` existent, en cas de ser-hi, s'ha de retornar l'error `synchronization_exists` tal i com s'especifica en el protocol, i si no, creo un nou `wl_resource` del tipus `zcr_synchronization_v1` que a la vegada el guardo al `weston_surface` corresponent i el retorno cap al client.

[54] `https://lwn.net/Articles/701964/`
[55] `https://codereview.chromium.org/2397843002/`

En aquest punt, els clients ja poden utilitzar les instàncies de la interfície `zcr_synchronization_v1` per utilitzar el *request* `set_acquire_fence`, que té com a paràmetre el descriptor de fitxer de la *fence* que es vol enviar, tenint en compte que tal com la majoria de les interfícies que utilitzen un `wl_surface` com a base, aquests *requests* modifiquen un estat que és *double-buffered* i que s'aplica de forma atòmica en el proper `commit`.

Per poder comprovar que l'*explicit fencing* funciona correctament, tant l'enviament de la *fence* com la posterior utilització d'aquesta, he creat un nou client de prova anomenat `simple-egl-explicit-synchronization.c`, que està basat en el client `simple-egl.c`, ja proporcionat per *Weston*. El funcionament de l'original `simple-egl.c` és molt simple: és connecta al servidor de *Wayland*, crea una finestra gràfica (`wl_surface`) i inicialitza un context d'OpenGL tot utilitzant EGL i utilitzant les extensions de *Wayland* corresponents, per finalment acabar dibuixant un triangle que va girant. El que volem ara és poder obtenir una *fence* cada vegada que es dibuixi el triangle, i enviar-la al compositor perquè actuï en conseqüència.

Obtenir una primitiva de sincronització en EGL es fa mitjançant la funció `eglCreateSyncKHR` de l'extensió `EGL_KHR_fence_sync`[56]. Aquesta funció, que retorna un objecte del tipus `EGLSyncKHR`, consta de tres paràmetres: un `EGLDisplay dpy` que representa el context EGL, un `EGLenum type` que indica el tipus de primitiva de sincronització, i un *array* `const EGLint *attrib_list` que permet passar-hi una llista d'atributs.

Amb la incorporació de l'extensió `EGL_ANDROID_native_fence_sync` per part de Google, els paràmetres que pot rebre `eglCreateSyncKHR` s'han estès de la següent manera: s'ha creat un tipus de primitiva de sincronització nativa nova (`EGL_SYNC_NATIVE_FENCE_ANDROID`) i s'ha afegit un nou possible atribut (`EGL_SYNC_NATIVE_FENCE_FD_ANDROID`), que permet passar-hi o bé un descriptor de fitxer vàlid per tal de crear un `EGLSyncKHR` a partir d'aquest descriptor, o passar-hi `EGL_NO_NATIVE_FENCE_FD_ANDROID` amb l'efecte d'obtenir un `EGLSyncKHR` vinculat a la llista de comandes que s'enviarà cap a la targeta gràfica i que s'activarà quan aquesta hagi acabat de renderitzar.

[56] `https://www.khronos.org/registry/EGL/extensions/KHR/EGL_KHR_fence_sync.txt`

En el nostre cas, ens interessa obtenir la primitiva de sincronització `EGLSyncKHR` vinculada a la finalització de la feina per part de la GPU. Per poder obtenir el descriptor de fitxer natiu a partir d'aquesta primitiva d'EGL hem d'utilitzar la funció `eglDupNativeFenceFDANDROID`, també afegida en l'extensió `EGL_ANDROID_native_fence_sync`. Aquesta funció, que retorna el descriptor de fitxer natiu, simplement rep com a paràmetres l'`EGLDisplay` del nostre context EGL, i la primitiva `EGLSyncKHR` a partir de la qual volem obtenir el descriptor de fitxer. Abans de poder obtenir el descriptor de fitxer, hem de cridar a la funció d'OpenGL `glFlush` per forçar que la primitiva de sincronització s'hagi creat. Un cop obtingut el descriptor de fitxer, podem destruir l'`EGLSyncKHR` corresponent mitjançant la funció `eglDestroySyncKHR` inclosa en l'extensió `EGL_KHR_fence_sync`, ja que hem duplicat la referència.

Finalment el client ja pot enviar utilitzant el *request* `set_acquire_fence` de la interfície `zcr_synchronization_v1` el descriptor de fitxer vinculat a la *fence* que indica la finalització de la feina per part de la GPU.

A la figura 15 és mostra en forma de codi C els passos que ha de seguir un client per poder obtenir i enviar la primitiva de sincronització utilitzant el protocol de sincronització explícita mencionat anteriorment.

```
/* Inicialització */
struct wl_surface *surface = ...;
EGLDisplay dpy = ...;

struct zcr_linux_explicit_synchronization_v1 *explicit_sync =
        wl_registry_bind(registry, name,
                         &zcr_linux_explicit_synchronization_v1_interface, 1);

struct zcr_synchronization_v1 *synchronization =
        zcr_linux_explicit_synchronization_v1_get_synchronization(
                explicit_sync, surface);

/* Dibuixat */
glDrawArrays(...);

EGLint attrib_list[] = {
        EGL_SYNC_NATIVE_FENCE_FD_ANDROID, EGL_NO_NATIVE_FENCE_FD_ANDROID,
        EGL_NONE,
};
EGLSyncKHR *egl_sync = eglCreateSyncKHR(dpy, EGL_SYNC_NATIVE_FENCE_ANDROID,
                                        attrib_list);

glFlush();

int fence_fd = eglDupNativeFenceFDANDROID(dpy, egl_sync);

eglDestroySyncKHR(dpy, egl_sync);

zcr_synchronization_v1_set_acquire_fence(synchronization, fence_fd);

close(fence_fd);

eglSwapBuffers(dpy, ...); /* Realitza un wl_surface.commit */
```

Figura 15: Codi d'exemple d'utilització de les extensions d'Android i del protocol de sincronització explícita.

Pel que fa al compositor, *Weston* utilitza una estructura anomenada `weston_surface_state` on va guardant tots els canvis que realitzen les peticions dels clients, i per tant cada `weston_surface` té un atribut d'aquest tipus amb el nom de `pending`. Com que ara els clients ens poden enviar una *fence*, he afegit un descriptor de fitxer (`acquire_fence`) a l'estructura `weston_surface_state` per poder guardar-la.

Originalment, quan un client realitzava un `commit`, es cridava a la funció `weston_surface_commit` de *Weston*, i a la vegada, aquesta cridava a `weston_surface_commit_state` passant-hi el `pending` de la `weston_surface` en qüestió com a paràmetre per tal de realitzar l'actualització atòmica de l'estat (copiar els atributs de `pending` a la seva `weston_surface`). Però ara no volem que l'estat s'apliqui immediatament, sinó que en el cas que s'hagi enviat una *fence*, volem esperar que la *fence* s'activi, ja que aquest fet indica que el client ha acabat de renderitzar i per tant podem mostrar el seu nou *buffer* sense que el compositor es bloquegi.

S'ha de tenir en compte que els `commit`s de les superfícies actuen com una *mailbox*, és a dir, només s'utilitzarà l'últim `commit` realitzat, i els anteriors es descarten en el cas que el client n'hagi fet més d'un i el compositor no hagi tingut temps de poder mostrar-los tots. Abans d'introduir el codi per l'*explicit fencing*, *Weston* ja duia a terme aquest comportament gràcies al fet que la funció `weston_surface_commit_state` es crida cada cop que un client envia un `commit`, i aquesta reemplaça l'estat de la `weston_surface` indiscriminadament.

Ara però, hem de considerar que és possible que quan el compositor rebi un `commit` dels clients, l'estat de la superfície en qüestió contingui el descriptor d'una *fence*, enviada mitjançant el nou protocol, i per tant ja no podem aplicar l'estat pendent indiscriminadament, sinó que hem de diferir l'aplicació de l'estat fins a l'activació de la *fence*. Per poder posposar l'aplicació de l'estat el que he fet ha sigut afegir un altre atribut del tipus `weston_surface_commit_state` a l'estructura `weston_surface`, que he anomenat `fence_pending`, i el mecanisme que utilitzo és molt simple: quan rebo un `commit`, si no hi ha cap *fence* vàlida a l'estat pendent actual (`pending`), aplico el `commit` instantàniament (tal com es feia abans), i en cas que n'hi hagi una de vàlida, en comptes d'aplicar l'estat pendent a la `weston_surface`, el que faig és copiar tota l'estructura `pending` cap a la nova estructura `fence_pending` i indicar que vull esperar que la *fence* s'activi. En ambdós casos, si ja hi havia una espera a una *fence* anterior pendent, la cancel·lo i tanco el seu descriptor de fitxer (comportament *mailbox*).

Aprofitant que el bucle principal de *Weston* està basat en el *dispatching* d'esdeveniments de descriptors de fitxer, una bona forma d'esperar que una *fence* s'activi és afegir el seu descriptor de fitxer a aquest *event loop* utilitzant la crida `wl_event_loop_add_fd` (que retorna un `wl_event_source`), ja que d'aquesta manera l'espera és passiva en comptes d'activa i per tant s'estalvia CPU. A aquesta crida hi passo la funció que vull que s'executi, i que s'executi quan es rebin esdeveniments de lectura. Quan la funció es cridi, com a conseqüència de l'arribada d'esdeveniments de lectura, utilitzo l'`ioctl SYNC_IOC_FILE_INFO` per obtenir informació sobre la *fence* i saber si s'ha activat. Quan s'activi, indicant que el client ha acabat de renderitzar el *buffer*, agafo l'estat pendent `fence_pending` i el copio a la seva `weston_surface` en qüestió, a més a més de tancar el descriptor de fitxer de la *fence* i de cancel·lar i eliminar la seva espera al *event loop* atès que ja no seran necessaris.

Durant la realització d'aquesta millora he estat discutint amb els desenvolupadors dels controladors gràfics d'Intel, Mesa i de *Wayland* per tenir un *feedback* sobre si la manera que estava implementat l'*explicit fencing* estava sent correcta o no. Gràcies a això, un d'aquests desenvolupadors va adonar-se que hi havia un *bug* en l'activació de la *fence* en el codi de Mesa, i per tant va enviar un *patch* per tal de solucionar-ho[57].

A la figura 16 s'hi mostra com *Weston* actuava anteriorment en rebre un `commit` d'un client. Per contra a la figura 17 s'hi mostren els canvis més rellevants mencionats, tant a les estructures de dades com al mateix codi de *Weston*, que he hagut de realitzar per afegir *explicit fencing*.

```
static void
weston_surface_commit(struct weston_surface *surface)
{
        /* Aplica directament el commit. */
        weston_surface_commit_apply(surface, &surface->pending);
}
```

Figura 16: Codi original de Weston abans d'afegir sincronització explícita.

[57] `https://lists.freedesktop.org/archives/mesa-dev/2017-June/158832.html`

```c
struct weston_surface {
        /* ... */
        struct weston_surface_state pending;
        /* S'ha afegit: */
        struct weston_surface_state fence_pending;
        struct wl_resource *explicit_sync_resource;
        struct wl_event_source *fence_source;
};
```

```c
struct weston_surface_state {
        /* ... */
        /* S'ha afegit: */
        int acquire_fence;
};
```

```c
static int
on_fence_readable(int fd, uint32_t mask, void *data)
{
        int ret;
        struct sync_file_info info;
        struct weston_surface *surface = data;

        memset(&info, 0, sizeof(info));
        ret = ioctl(fd, SYNC_IOC_FILE_INFO, &info);
        if (ret < 0 || info.status == 1) { /* Si la fence s'ha activat... */
                /* Cancel·lem i eliminem l'espera actual. */
                wl_event_source_remove(surface->fence_source);
                surface->fence_source = NULL;
                close(fd);

                /* Apliquem el commit. */
                weston_surface_commit_apply(surface, &surface->fence_pending);
                weston_surface_state_init(&surface->fence_pending);
        }

        return 1;
}

static void
weston_surface_commit(struct weston_surface *surface)
{
        /* Si ja hi ha una espera en procés, la cancel·lo (mailbox). */
        if (surface->fence_source) {
                close(surface->fence_pending.acquire_fence);
                wl_event_source_remove(surface->fence_source);
                surface->fence_source = NULL;
        }

        if (surface->pending.acquire_fence == -1) { /* El client no ha enviat cap fence. */
                weston_surface_commit_apply(surface, &surface->pending);
        } else {                                    /* El client ha enviat una fence. */
                struct weston_compositor *compositor = surface->compositor;
                struct wl_event_loop *loop = wl_display_get_event_loop(compositor->wl_display);

                surface->fence_pending = surface->pending;

                surface->fence_source = wl_event_loop_add_fd(loop,
                                                             surface->pending.acquire_fence,
                                                             WL_EVENT_READABLE,
                                                             on_fence_readable, surface);

                weston_surface_state_init(&surface->pending);
        }
}
```

Figura 17: Canvis més destacats del codi de Weston per afegir sincronització explícita.

12.4 Solució d'un *bug* en l'algorisme d'assignació dels *hardware planes*

Una altra de les millores realitzades és, com ja s'ha mencionat anteriorment, la solució d'un *bug* que hi havia en l'algorisme que s'encarrega d'assignar les `weston_view`s (finestres dels clients i elements gràfics del compositor) als *hardware planes* en el `drm-backend.so`, en concret a la funció `drm_output_propose_state`.

Aquest algorisme funciona mitjançant *clipping* (retalls), unions i interseccions de regions rectangulars, ja que la geometria de cada `weston_view` es representa mitjançant una estructura de dades d'aquest tipus (`pixman_region32_t`).

El repintat dels diferents *outputs* es realitza quan s'invoca a la funció `output_repaint_timer_handler`, que es crida automàticament quan salta un temporitzador. El temporitzador es torna a reprogramar al final del mateix *handler*, d'aquesta manera es realitzen els repintats periòdicament. Com que cada *output* (habitualment una pantalla) té el seu cicle de refresc propi, i n'hi poden haver múltiples, el temporitzador es programa en un temps calculat (`msec_to_next`) utilitzant el temps actual (`now`) i la següent fórmula, on `next_repaint` retorna el pròxim múltiple del període de refresc de l'*output*:

$$msec\_to\_next = \min_{\forall o \in output\_list} (next\_repaint(o) - now)$$

Quan el temporitzador es dispara, el primer que fa el *handler* és avisar al *backend* que s'està a punt de començar un nou cicle de repintat (`repaint_begin`). Seguidament, per cada *output* de la llista, es comprova si necessita repintar-se, si aquest és el cas, es mira si l'*output* disposa d'una funció d'assignació de `weston_view`s a *hardware planes*, i si la té, es crida a aquesta funció, en cas contrari s'agafen totes les `weston_view`s del compositor i es mouen a la llista `primary_plane` per renderitzar-los de forma manual utilitzant el *renderer* posteriorment. Després s'invoca a la funció de `repaint` de l'*output*, que utilitzarà el *renderer* en ús per efectuar el pintat dels elements del `primary_plane`, i finalment, quan s'han repintat tots els *outputs* es torna a avisar al *backend* que el cicle de repintat s'ha acabat (`repaint_flush`). A la figura 18 es mostra en forma de pseudocodi aquest algorisme.

```
Require: compositor, el weston_compositor
  function OUTPUT_REPAINT_TIMER_HANDLER(compositor)
    compositor.backend.repaint_begin()
    foreach output ∈ compositor.output_list do
      if ¬output.needs_repaint then
        continue
      end if
      if ∃output.assign_planes then
        output.assign_planes()
      else
        foreach view ∈ compositor.view_list do
          move_view_to_plane(view, compositor.primary_plane)
        end for
      end if
      output.repaint()
    end for
    compositor.backend.repaint_flush()
  end function
```

Figura 18: Algorisme de repintat de Weston.

L'únic *backend* que assigna una funció `assign_planes` als *outputs* és `drm-backend.so` (funció `drm_assign_planes`), ja que aquest és l'únic que utilitza *libdrm*, i per tant que pot utilitzar els *hardware planes*.

L'objectiu de la funció `drm_assign_planes` és assignar el nombre màxim de `weston_views` a *hardware planes* possible, i en conseqüència, la funció delega aquesta responsabilitat cridant a `drm_output_propose_state` dues vegades amb diferents *flags*: primer li passa `DRM_OUTPUT_PROPOSE_STATE_PLANES_ONLY` per indicar que s'intentin assignar totes les `weston_views` a *hardware planes*, i que si alguna no s'ha pogut assignar, que retorni fals, i després, en el cas que la primera crida hagi fallat, es passa el *flag* `DRM_OUTPUT_PROPOSE_STATE_MIXED` indicant que les `weston_views` que no es puguin assignar a *hardware planes* es moguin a la llista `primary_plane`, per efectuar el posterior renderitzat.

Així doncs, que la funció `drm_output_propose_state` operi correctament és vital per tal de treure el màxim profit als *hardware planes*, ja que és la funció encarregada de l'assignació de `weston_views` a aquests, i és precisament en aquesta funció on hi havia el *bug*.

La funció `drm_output_propose_state` comença inicialitzant dues regions, `renderer_region` i `occluded_region`, del tipus `pixman_region32_t`, que s'utilitzaran per optimitzar l'assignació de les `weston_view`s i evitar processar les que estiguin totalment ocultes tapades per d'altres. Seguidament s'itera per tota la llista de `weston_view`s que té el compositor (`view_list`). Una característica molt important de la `view_list` és que està ordenada començant per les `weston_view`s que estan més amunt (que no estan tapades per altres `weston_view`s), fins a arribar a la que hi ha al darrere de tot (habitualment la imatge del fons d'escriptori).

Un cop dins el bucle, primer es mira si la `weston_view` actual s'ha de mostrar en el *output* sobre el qual s'estan assignant les `weston_view`, i en cas negatiu es procedeix a la següent iteració. Si a més a més de mostrar-se al *output* actual, també s'ha de mostrar en d'altres, es força el renderitzat (en comptes d'intentar posar-la a un *hardware plane*), ja que això simplifica molt codi i es necessitaria una infraestructura encara no disponible (*explicit fencing*).

En aquest punt és quan s'utilitza la regió `occluded_region` per intentar descartar les `weston_view`s totalment ocultes. El que es fa és realitzar una resta de regions (`pixman_region32_subtract`) entre la geometria de la `weston_view` i la `occluded_region`, i si el resultat d'aquesta resta dóna una regió buida, significaria que podem passar a la iteració següent, tot i que precisament en aquest punt és on hi ha el *bug*.

El problema està quan una `weston_view` té un tros de la seva geometria a fora del *output*, per exemple quan s'ha mogut amb el cursor, i s'intenta posar una altra `weston_view` en pantalla completa, que per tant la `occluded_region` passa a ser igual a la regió del *output*. En aquest cas, la resta entre la geometria de la `weston_view` i la `occluded_region` donaria una regió no buida quan ho hauria de ser, fet que que desencadena que s'assignin punters quan no s'han d'assignar i acaba amb un *segmentation fault*. La solució que he realitzat per solucionar el *bug* és molt senzilla: un cop feta la resta entre la `weston_view` i la `occluded_region`, faig una intersecció de regions (`pixman_region32_intersect`) sobre el resultat de la resta i la geometria d'*output*, d'aquesta manera es podria dir que retallo tot el que queda a fora d'aquest i que per tant no s'ha de tenir en compte.

Posteriorment l'algorisme comprova si s'ha de forçar el renderitzat de la `weston_view` o no. Per dur a terme aquesta tasca, s'utilitza la regió `renderer_region`. El que fa és la intersecció entre la geometria de la `weston_view` i la `renderer_region`, i si el resultat dóna una regió que no és buida, significa que hi ha superposició i que per tant la `weston_view` s'ha de renderitzar i no es pot posar en un *hardware plane*, ja que representa que està parcialment tapada.

Ara que ja sabem si la `weston_view` és candidata a ser posada en un *hardware plane* o s'ha de renderitzar manualment, actualitzem `occluded_region` només en aquest últim cas, aplicant-hi una unió de regions (`pixman_region32_union`) entre ella mateixa i la regió de la `weston_view`.

Per últim, s'ha d'intentar posar la `weston_view` un *hardware plane* només si s'ha decidit que no s'ha de forçar el seu renderitzat. Primer es prova si pot anar al *cursor plane* (`drm_output_prepare_cursor_view`), en cas negatiu, es prova amb el *scanout/primary plane* (`drm_output_prepare_scanout_view`), i si tampoc va, finalment es prova amb els *overlay planes* (`drm_output_prepare_overlay_view`). Si no s'aconsegueix assignar la `weston_view` a cap *hardware plane*, s'actualitza `renderer_region` realitzant una unió de regions entre ella mateixa i la geometria de la `weston_view`, i posteriorment, la funció que ens ha cridat (`drm_assign_planes`) es donarà compte i posarà la `weston_view` a la llista `primary_plane` per forçar-ne el renderitzat.

Tota aquesta lògica explicada prèviament es mostra en forma l'algorisme a la figura 19. La fletxa indica en quin punt s'ha realitzat la modificació per tal de solucionar el *bug* que hi havia.

Require: *compositor*, el *weston_compositor*
Require: *output*, el *weston_output* al qual estem assignant els *planes*
Require: *state*, el *drm_output_state* actual
  $renderer\_region = \varnothing$
  $occluded\_region = \varnothing$
  foreach $view \in compositor.view\_list$ **do**
    $ps = NULL$
    $force\_renderer = false$
    $occluded = false$
    if $\neg(view.output\_mask \,\&\, (1 << output.id))$ **then**
      continue
    end if
    if $view.output\_mask \neq (1 << output.id)$ **then**
      $force\_renderer = true$
    end if
    if $(view.geometry \setminus occluded\_region) = \varnothing$ **then**
      continue
    end if
    if $(view.geometry \cap renderer\_region) \neq \varnothing$ **then**
      $force\_renderer = true$
    end if
    if $view.is\_opaque$ **then**
      $occluded\_region = occluded\_region \cup view.geometry$
    end if
    if $\neg force\_renderer$ **then**
      $ps = drm\_output\_prepare\_cursor\_view(state, view)$
    end if
    if $\neg force\_renderer \wedge ps = NULL$ **then**
      $ps = drm\_output\_prepare\_scanout\_view(state, view)$
    end if
    if $\neg force\_renderer \wedge ps = NULL$ **then**
      $ps = drm\_output\_prepare\_overlay\_view(state, view)$
    end if
    if $ps \neq NULL$ **then**
      continue
    end if
    $renderer\_region = renderer\_region \cup view.geometry$
  end for

$((view.geometry \setminus occluded\_region) \cap output.geometry) = \varnothing$

Figura 19: Algorisme d'assignació de `weston_views` *a hardware planes.*

13 Resultats

En aquest apartat es mostren quins resultats s'han obtingut a causa d'haver dut a terme el projecte. En primer lloc es mostraran les conseqüències d'utilitzar o no *atomic mode-setting* mitjançant el programa de prova realitzat *atomictest*, i en segon lloc es mostrarà com ha influït a *Weston* i al seu rendiment les diverses millores que s'han realitzat al llarg del projecte.

13.1 Programa de prova *atomictest*

La primera versió del programa, que consistia a utilitzar les crides no atòmiques, pateix d'una degradació dels FPS resultants molt important, ja que tal com s'ha mencionat anteriorment, la funció utilitzada per situar els *overlay planes* (`drmModeSetPlane`) és en la majoria d'implementacions bloquejant final proper *VBLANK*.

De manera teòrica, considerant el bloqueig que causa `drmModeSetPlane`, podem deduir que l'expressió que ens retorna la reducció en el nombre d'FPS resultants en funció del nombre d'*overlays* utilitzats, i per tant, crides a la funció, i, és la següent: $1 - (1/(i+1))$.

Per altra banda, si utilitzem l'API atòmica (i per tant les propietats dels objectes DRM), tal com fa la segona versió d'*atomictest*, no es produiria tal bloqueig. D'aquesta manera, si suposem que aquests *VBLANKs* succeeixen 60 vegades per segon (60 FPS), si comparem el *frame-rate* que obtindríem en situar N *overlay planes* quan utilitzem `drmModeSetPlane` i quan utilitzem l'API atòmica, el que obtindríem teòricament és el mostrat a la figura 20.

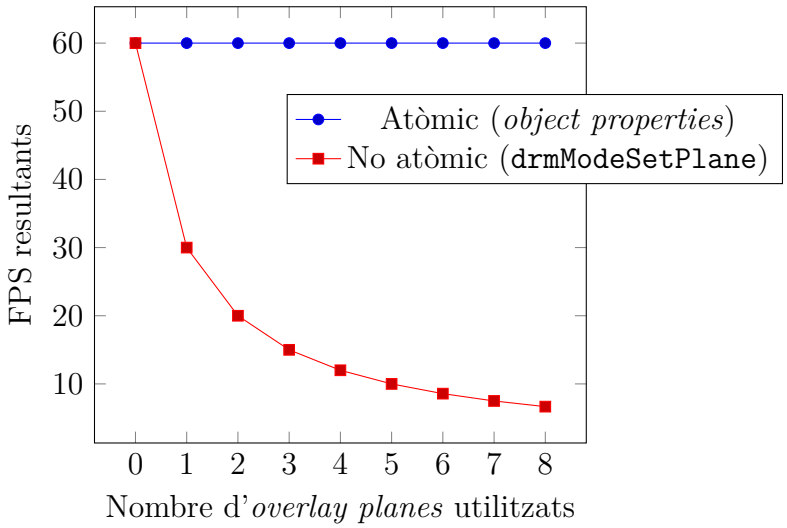


Figura 20: FPS resultants teòrics en funció dels overlays i l'atomic mode-setting.

Amb la finalitat d'obtenir resultats empírics, he realitzat una lleugera modificació a *atomictest* que permet passar-hi com a paràmetre el nombre d'*overlays* que es volen utilitzar, i ara quan s'acaba l'execució es retorna el temps emprat, el nombre *frames* mostrats i els FPS en mitjana.

Per poder comparar el comportament teòric amb l'empíric, he executat *atomictest* tantes vegades com nombre *overlays* disposen els dispositius, que són 8 en el cas de la *Raspberry Pi 3*, i 1 en el cas de l'ordinador portàtil, durant 20 segons, i utilitzant 60FPS com a *frame-rate* base, tot obtenint els FPS resultants. Tal com podem observar a la figura 21, els resultats teòrics i els empírics coincideixen d'una forma molt clara.

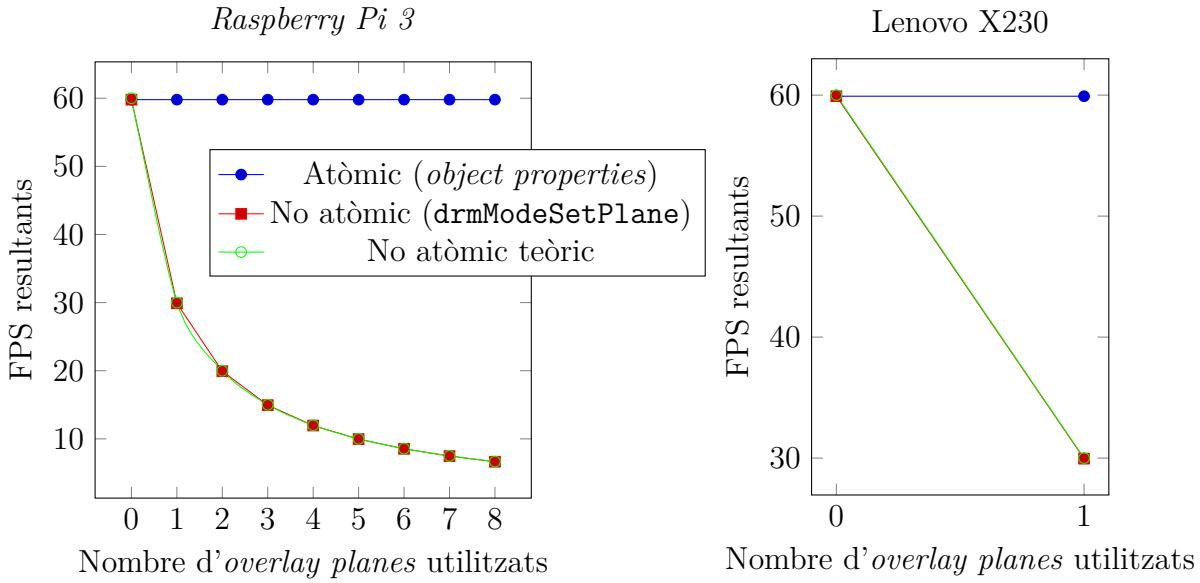


Figura 21: FPS resultants en funció del nombre d'overlays i utilitzar o no atomic mode-setting.

13.2 Utilització del *pixel format modifiers blob*

La branca de *Weston* utilitzada com a base per totes les modificacions del projecte, la del desenvolupador Daniel Stone, ja introduïa un ús inicial de les funcions atòmiques, afegides en les versions més recents del *kernel* de Linux i *libdrm*. Aquestes noves funcions han permès entre altres coses, poder utilitzar els *hardware planes* dels que disposa el nostre sistema, tot i que amb matisos, ja que abans no hi havia forma d'obtenir els modificadors de píxel suportats per cada *plane*, cosa que potencialment reduïa el possible ús d'aquests. Així doncs, es pot considerar que aquesta millora, la d'introduir suport a *Weston* del *pixel format modifiers blob*, augmenta la possibilitat de poder posar un client/*framebuffer* arbitrari en un *hardware plane*.

Per poder provar si la modificació funciona correctament o no, s'hauria de forçar la utilització de modificadors de format de píxel per part dels clients, cosa que en funció de la implementació d'EGL, ja fa per defecte. Si es vol tenir control total sobre tant el format de píxel com el modificador d'aquest, s'haurien de reservar els *framebuffers* utilitzant la llibreria *libgbm* de Mesa, i l'ús d'aquesta no ha estat en cap moment dins la planificació del projecte.

Tot i que no puc comprovar de forma directa el resultat d'aquesta modificació pel motiu mencionat, sí que puc comprovar d'una forma més genèrica quin efecte tenen els *hardware planes* en l'àmbit d'estalviar al compositor haver de renderitzar el client.

Per dur a terme aquestes proves de rendiment, he realitzat una modificació a les eines d'Intel `intel-gpu-tools`[58]. Tot i que inicialment aquestes eines sí que només eren un conjunt de proves i eines pel controlador de DRM d'Intel, progressivament s'hi han anat sumant altres venedors, fins a convertir-se les eines de prova de DRM per excel·lència.

L'eina `intel_perf_counters` (inclosa dins de `intel-gpu-tools`) llegeix els comptadors *hardware* dels que les GPU d'Intel disposen. Un d'aquests comptadors, anomenat *3D/GPGPU Render Target Writes*, retorna el nombre de píxels total als quals la GPU hi ha escrit, així que aquest és el que he utilitzat per comprovar l'estalvi de renderització per part del compositor. La modificació que he realitzat sobre aquesta eina m'ha facilitat l'obtenció dels resultats que he necessitat, tot i que malauradament, la prova només s'ha pogut realitzar sobre l'ordinador portàtil Lenovo X230, ja que la *Raspberry Pi 3* no disposa dels comptadors *hardware* necessaris.

[58] `https://cgit.freedesktop.org/xorg/app/intel-gpu-tools/`

La prova de rendiment ha consistit en el següent: he obtingut el nombre de píxels totals que la GPU ha escrit mitjançant l'eina `intel_perf_counters` i executant el client *simple-egl* de *Weston* diverses vegades durant 20 segons cada una, primer amb diferents resolucions i passant-hi un paràmetre per forçar l'opacitat de la finestra, i després amb les mateixes resolucions que abans però deixant que la finestra tingui transparència, és a dir, no sigui opaca. L'objectiu de fer que la finestra sigui opaca és permetre a *Weston* que posi aquesta en un *hardware plane* i així doncs, eviti haver-la de renderitzar.

Si consideréssim que la GPU només ha de realitzar la tasca del client *simple-egl* (el que anomeno resultat òptim) amb una certa amplada i alçada, el nombre de píxels que aquesta hauria d'escriure durant un període de temps determinat i a uns certs FPS són els següents: $amplada \cdot alçada \cdot frames/s \cdot temps$, però hem de considerar que la interfície gràfica base de *Weston* també disposa d'altres clients, com per exemple el panell que mostra l'hora, i que de tant en tant aquests poden enviar `commits` cap al compositor. Així doncs, a la taula 10 es mostra el nombre d'escriptures de la GPU en realitzar la prova mencionada.

Resolució / #píxels	**No Opac**	**Opac**	**Resultat òptim**
352x240 / 84480	311,954,801	109,242,032	101,376,000
480x360 / 172800	637,605,940	225,361,960	207,360,000
858x480 / 411840	1,519,423,748	532,388,518	494,208,000
1280x720 / 921600	3,403,479,559	1,192,419,019	1,105,920,000

Taula 10: Número de Render Target Writes durant 20 segons a 60FPS.

Com podem observar, el resultat òptim i el de la finestra opaca són molt similars, cosa que significa que realment estem descarregant la feina de la tasca que hauria de realitzar el compositor en renderitzar el client cap a les capacitats que té el nostre *hardware* mitjançant els *hardware planes*, i per tant estalviant *bandwidth* d'accessos a memòria. Si en canvi mirem els resultats amb la finestra no opaca, veiem que quasi es tripliquen les escriptures que la GPU realitza, i tot i que en un principi això pot semblar estrany, ja que un esperaria una duplicació, aquest fet té una explicació molt senzilla: *Weston* comença renderitzant les `weston_views` que hi ha més a sota, per tant en primer lloc pintarà els píxels del fons d'escriptori que hi ha just a sota de la finestra, i seguidament és quan hi pintarà la finestra. Si *Weston* no ho fes d'aquesta manera, les finestres amb transparència no es visualitzarien correctament. El fet de no arribar a triplicar exactament el nombre de píxels que la GPU escriu és molt possiblement degut a la *cache* que aquesta disposa.

De forma gràfica, aquests mateixos resultats es mostren a la figura 22.

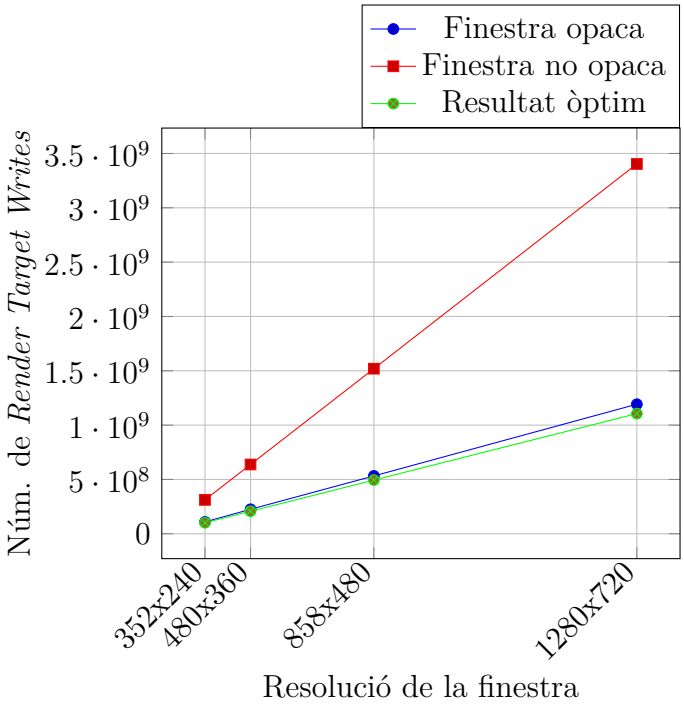


Figura 22: Número de Render Target Writes en funció de la opacitat de la finestra, durant 20 segons a 60FPS.

13.3 Utilització d'*explicit fencing*

Malauradament, la majoria de GPUs integrades o en sistemes encastats (com per exemple la *Raspberry Pi 3*) que trobem al mercat, no suporten preempció de tasques (*batches*), a més a més que el *driver* DRM d'aquests dispositius encua les tasques de manera FIFO (*first in, first out*), i per tant la tasca del compositor no es realitzarà fins que totes les anteriors, és a dir, les dels clients, hagin acabat.

En conseqüència, aquest fet suposa que encara que el compositor s'esperi mitjançant una *fence* que el nou *buffer* que ha enviat el client hagi acabat de renderitzar-se abans d'utilitzar-lo, el compositor continuarà patint un bloqueig quan aquest intenti renderitzar. Quan s'utilitza una GPU d'aquestes característiques, l'únic moment en el qual es pot evitar aquest bloqueig és quan el compositor no ha de dur a terme cap mena de renderitzat, ja que per exemple tots els clients es poden posar en *hardware planes*, o que cap client ha realitzat un *commit* d'algun dels seus *buffers* respecte a l'última vegada que s'ha renderitzat, i per tant no cal realitzar cap mena de renderitzat que actualitzi el *primary plane*.

Per altra banda, les targetes més modernes dels dos venedors de GPUs discretes (no integrades en el mateix xip de la CPU), com són NVIDIA i AMD, sí que suporten la preempció de tasques, tot i que no disposo de cap dispositiu amb aquest *hardware*, i a més a més m'hauria de posar en contacte amb els desenvolupadors d'aquestes companyies per informar-me l'estat dels seus *drivers* de Linux. El que és cert és que AMD està intentant dotar els seus controladors de Linux de suport per poder planificar tasques d'alta prioritat en les seves targetes gràfiques[59].

Així doncs, no he pogut realitzar una prova real que demostri que els meus canvis que afegeixen *explicit fencing* funcionen correctament, tot i que he fet *debugging* mitjançant *logging* (escrivint a un fitxer) i sembla que actuen correctament, de manera que és molt probable que en una GPU que suporti preempció de tasques i que els seus controladors de Linux tinguin bon suport, és molt possible que funcionin bé.

13.4 Solució d'un *bug* en l'algorisme d'assignació dels *hardware planes*

El resultat obtingut gràcies a la solució del *bug* que hi havia en una part tan vital de *Weston* com és l'algorisme que assigna les `weston_views` en els *hardware planes* no comporta cap mena de millora de rendiment en si, sinó que evita que *Weston* acabi inesperadament (*crash*) en un cas d'ús tan habitual com posar un client a pantalla completa quan n'hi ha un altre que s'ha mogut parcialment a fora de la pantalla.

Així doncs comprovar el resultat és molt senzill: només cal obrir *Weston*, moure un client parcialment a fora de la pantalla, i posar a pantalla completa un altre client. En el cas que funcioni correctament significa que el *bug* s'ha resolt de forma satisfactòria.

[59] `https://lists.freedesktop.org/archives/amd-gfx/2017-February/005842.html`

14 Revisió del projecte

En aquest apartat es revisaran tots aquells aspectes planejats originalment a l'inici del projecte i que potencialment poden haver sofert desviacions. Aquests aspectes consten de l'estudi econòmic plantejat, canvis en el temps dedicat a les tasques que comporten la realització del projecte, és a dir, canvis en la planificació, i possibles canvis en la metodologia.

14.1 Planificació temporal

Inicialment es va plantejar dividir la realització del projecte en tres grans activitats principals: la configuració de l'entorn de treball, l'aprenentatge de les eines necessàries, i realitzar els canvis al codi de *Weston* per tal d'implementar les millores i optimitzacions. A la taula 11 hi podem observar la desviació (en hores) entre el temps originalment planificat i el finalment dedicat a les diverses tasques. En els següents apartats s'explicarà el progrés realitzat en cada una d'elles i les possibles desviacions que hi hagi hagut en comparació de la planificació inicial.

Tasca	**Temps planificat**	**Temps dedicat**
Gestió del projecte	75	75
Configuració de l'ordinador	18	10
Instal·lació *cross-compiler*	12	8
Compilació del *kernel* Linux	12	8
Configuració de la *Raspberry Pi 3*	13	9
Aprenentatge *autotools*	10	6
Aprenentatge *libdrm*	36	60
Familiarització codi de *Weston*	22	12
Buscar pla per optimitzar *Weston*	22	12
Realitzar l'optimització de *Weston*	140	180
Proves de rendiment	55	35
Memòria final	35	35
Total	**450**	**450**

Taula 11: Comparació entre el temps planificat i el dedicat a les tasques del projecte.

14.1.1 Configuració de l'entorn

Pel que fa a les tasques de la configuració de l'entorn, es pot dir que es va estimar massa temps a aquesta activitat amb un marge d'unes 20 hores de més (va durar 35 hores en comptes de les 55 estimades). Considero que aquest fet és principalment a causa de la meva experiència prèvia al projecte a la instal·lació i configuració de sistemes Linux, i per tant en cap moment vaig entrebancar-me.

14.1.2 Aprenentatge de les eines

L'aprenentatge de les eines, també ha constat en escriure un programa de prova molt senzill utilitzant *autotools* (sistema de compilació que utilitza Wayland i Weston), i un altre de més complex que utilitza *libdrm* (llibreria que fa servir Weston per comunicar-se amb el subsistema DRM de Linux).

En primer terme, el programa utilitzant *autotools* ha sigut una tasca fàcil gràcies a l'abundant quantitat d'informació, tutorials, i presentacions que hi ha *online* sobre aquest sistema de compilació.

En segon terme, el programa de prova de *libdrm* ha sigut una tasca més complicada de l'esperat, ja que a diferència d'*autotools*, no hi ha una documentació explícita de l'API de la llibreria, i encara menys informació en format tutorial; per tant vaig haver d'aprendre com funcionava mirant i entenent el codi d'altres projectes[60][61] que utilitzen *libdrm*. No només el fet de disposar d'aquesta escassa informació va complicar la tasca, sinó que a fi d'entendre com funciona la nova API atòmica de DRM, vaig escriure el programa de prova dues vegades: primer utilitzant l'API feta servir fins ara (és a dir, no atòmica), i després portant el codi a la nova API atòmica. Si recursos de l'API antiga ja n'hi havia pocs, de l'API nova encara n'hi ha menys, cosa que em va comportar alguna vegada a fer preguntes al xat IRC oficial on hi ha els desenvolupadors de DRM.

Tots aquests inconvenients que no tenia presents en l'estimació del cost de les tasques inicialment, van fer que a l'hora de la veritat la realització d'aquesta segona activitat trigués unes 20 hores de més, és a dir, vaig acabar invertint les hores estalviades a la primera activitat (14.1.1) en aquesta.

[60] `https://github.com/dvdhrm/docs/tree/master/drm-howto`
[61] `https://cgit.freedesktop.org/mesa/kmscube/`

14.1.3 Optimització de *Weston*

D'aquesta última activitat, que ha consistit a realitzar l'optimització de *Weston* i realitzar les proves de rendiment pertinents, en podria dir que dins el que cap, vaig planificar prou bé el temps que necessitaria, tot i que hi he acabat invertint més temps de l'originalment planificat. Aquest fet ha comportat que no he pogut dedicar tot el temps desitjat per dur a terme les proves de rendiment, cosa que fa que no hagi disposat de tantes mostres com podria haver volgut per poder treure millors resultats de la realització d'aquest projecte.

14.2 Metodologia

El mètode plantejat inicialment que consistia en el següent s'ha seguit de forma eficaç: el desenvolupament es realitza des d'un ordinador host i una Raspberry Pi 3. A l'ordinador portàtil és a on es du a terme la modificació del codi i les execucions principals de Weston, tot i que de tant en tant també s'ha utilitzat la Raspberry Pi 3 per comprovar que els canvis realitzats funcionen correctament als dos dispositius.

Pel que fa a la metodologia de treball, tal com s'havia plantejat, he seguit en certs aspectes el desenvolupament *agile*: els requisits i solucions evolucionen amb el temps segons la necessitat del projecte. Això es deu al fet que Weston és un *software* real que està en constant desenvolupament, i fins que no vaig arribar a la tasca de familiaritzar-me amb el codi de Weston, no podia especificar precisament quines millores realitzaria. També s'ha de tenir en compte que m'he hagut d'adaptar a la filosofia de desenvolupament de Weston: parlar amb els desenvolupadors al xat IRC, enviar els canvis a aquests, consultar la *mailing list*, etc.

14.3 Costos i estudi econòmic

A causa dels canvis en el temps dedicat a les tasques del projecte respecte al temps planificat inicialment el cost total del projecte s'ha vist lleugerament modificat en comparació el pressupost total, tal com es pot veure a la figura 12.

	Amortització (€/h)	Durada (h)	Cost (€)
Gestió del projecte		**75**	**1.425**
ThinkPad X230	0.014		1.05
Google Nexus 4	0.005		0.375
Configuració de l'ordinador		**10**	**0.14**
ThinkPad X230	0.014		0.14
Instal·lació *cross-compiler*		**8**	**0.112**
ThinkPad X230	0.014		0.112
Compilació del *kernel* Linux		**8**	**0.184**
ThinkPad X230	0.014		0.112
Raspberry Pi 3 Model B	0.002		0.016
TV LED LG M2631D	0.007		0.056
Configuració de la *Raspberry Pi 3*		**9**	**0.081**
Raspberry Pi 3 Model B	0.002		0.018
TV LED LG M2631D	0.007		0.063
Aprenentatge autotools		**6**	**0.084**
ThinkPad X230	0.014		0.084
Aprenentatge libdrm		**60**	**1.38**
ThinkPad X230	0.014		0.84
Raspberry Pi 3 Model B	0.002		0.12
TV LED LG M2631D	0.007		0.42
Familiarització codi de Weston		**12**	**0.168**
ThinkPad X230	0.014		0.168
Buscar pla per optimitzar Weston		**12**	**0.168**
ThinkPad X230	0.014		0.168
Realitzar l'optimització de Weston		**180**	**4.14**
ThinkPad X230	0.014		2.52
Raspberry Pi 3 Model B	0.002		0.36
TV LED LG M2631D	0.007		1.26
Proves de rendiment		**35**	**0.805**
ThinkPad X230	0.014		0.49
Raspberry Pi 3 Model B	0.002		0.07
TV LED LG M2631D	0.007		0.245
Memòria final		**35**	**0.49**
ThinkPad X230	0.014		0.49
Costos indirectes			**704**
Recursos humans			**9000**
Total sense IVA			**9,713.18**
Total amb IVA	21% sobre 9,713.18		**11752.95**

Taula 12: Cost total del projecte.

14.4 Lleis i regulacions

Aquest projecte en cap moment ha obtingut o emmagatzemat dades o fitxers de dades amb caràcter personal, i per tant queda exclòs de la llei orgànica 15/1999 de protecció de dades de caràcter personal (LOPD)[62].

Pel que fa a les llicències *software*, el projecte *Wayland*, és a dir, és *Wayland*, *Weston* i *Wayland protocols*, utilitzaven inicialment la llicència *Historical Permission Notice and Disclaimer (HPND)*[63] tot i que posteriorment es va canviar a la *MIT License*[64] ja que és la que utilitza la implementació X.org del servidor X i es volia que *Wayland* gaudís de les mateixes condicions que X.org.

[62] `http://www.agpd.es/portalwebAGPD/canaldocumentacion/informes_juridicos/reglamento_lopd/index-ides-idphp.php`

[63] `https://opensource.org/licenses/HPND`

[64] `https://opensource.org/licenses/MIT`

15 Conclusions

En l'àmbit tècnic, puc considerar que els objectius marcats a l'inici del projecte s'han assolit satisfactòriament. En primer lloc, el programa de prova *atomictest* demostra empíricament que la nova API atòmica de DRM, i en particular l'*atomic mode-setting* i totes les seves millores relacionades, realment disposa d'avantatges molt importants respecte de les versions antigues d'aquest subsistema, utilitzades fins ara. En segon lloc, les dues millores realitzades a *Weston* ajuden a fer que aquest sigui un compositor més eficient, que aprofiti els recursos del sistema sobre el qual s'executa, i que per tant estalviï energia. I en tercer lloc, la solució del *bug* que hi havia en un dels algorismes principals de *Weston* que acabava desembocant en un *crash* permet evitar la possible frustració dels usuaris que es trobin en aquest problema.

A escala personal, aquest projecte m'ha permès aprendre nous conceptes i mètodes que estic molt segur que em seran útils en el meu futur professional. Per una banda he après com configurar i compilar projectes *open-source* tan rellevants com són el *kernel* de Linux, Mesa i *Wayland* entre altres. Per altra banda m'he comunicat amb els desenvolupadors de *Wayland* i *Weston* utilitzant tant una *mailing list* com el xat *IRC*, eines molt usades en projectes d'aquest caire. I per últim, el projecte m'ha introduït a eines de producció de documents tècnics i professionals, com ara *LaTeX*[65], a més a més d'haver perfeccionat el meu domini del sistema de control de versions *git*.

[65] `https://www.latex-project.org/`

16 Treball futur

Al llarg de tot aquest document s'ha fet èmfasi diverses vegades en el fet que el subsistema DRM de Linux està en procés d'evolució i maduració cap a una infraestructura gràfica moderna. El recent afegit *atomic mode-setting* al mateix *kernel* ha suposat la primera petjada en aquesta direcció de modernització del subsistema gràfic, el qual en comparació amb altres sistemes operatius, *Linux* sempre s'havia quedat enrere.

Així doncs, en aquest apartat es plantejaran diferents millores i idees que tot just s'estan començant a introduir a DRM, partint de la base de l'*atomic mode-setting*, i que *Weston* i la resta de programes gràfics i compositors en general se'n podran aprofitar altament un cop aquestes finalment aterrin.

16.1 Extensió de l'*explicit fencing*

Tot i que aquest projecte ja ha introduït un suport preliminar del l'*explicit fencing* a *Weston*, encara hi ha camí per avançar abans que aquests canvis puguin arribar al codi oficial.

D'entrada, el protocol *Wayland* utilitzat per enviar la *fence* és extret de Chromium OS, i per tant l'ideal seria afegir-lo al repositori *wayland-protocols* de forma que altres projectes també el puguin utilitzar de forma fàcil.

A més a més, els canvis realitzats en aquest projecte permeten evitar utilitzar *framebuffers* que els clients encara estiguin renderitzant, si el *hardware* ho suporta, però no s'ha implementat la utilització del descriptor de fitxer de la *fence* en els punts de sincronització on aquest es pot inserir per tal de millorar-ne la *debuggabilitat*. Un d'aquests punts de sincronització és la propietat `IN_PLANE_FD` dels *hardware planes*, i l'altre la crida `eglWaitSyncKHR` d'EGL per quan s'utilitza el *renderer*.

Per últim, disposar d'un perfecte ús de l'*explicit fencing* és essencial per poder suportar de forma correcta sistemes amb múltiples GPUs, en especial sistemes on els *buffers* utilitzats o mostrats per una GPU són produïts per una d'altre. Aquest cas d'ús és molt habitual en sistemes moderns i actualment s'està proposant[66] com enfocar-lo de forma òptima.

[66] `https://www.x.org/wiki/Events/XDC2016/Program/xdc-2016-prime-sync.pdf`

16.2 Utilitzar els *writeback connectors*

Un dels problemes que comporta realitzar captures de pantalla o gravar-ne el contingut quan s'utilitzen *hardware planes* és que aquests s'han de desactivar mentre es duguin a terme aquestes accions, ja que es necessita disposar del *framebuffer* complet, produït pel *renderer* per tal de guardar-lo a disc.

Per resoldre aquest problema, algunes de les últimes GPUs que trobem en el mercat disposen en el seu *hardware* del que s'anomenen *writeback engines*. Aquests *writeback engines* el que permeten és reescriure a memòria, en un *framebuffer*, el resultat de la composició *hardware* que generen els CRTCs, paralel·lament a l'enviament d'aquesta composició cap a l'*encoder* i posteriorment cap al *connector* per acabar a la pantalla.

Pel que fa al suport d'aquests sobre Linux, ja hi han *patches* preparats pel subsistema DRM[67] que n'afegeixen suport de forma que els *writeback engines* s'exposen a l'API DRM com un tipus de *connector* nou, anomenat *writeback connector*.

En conclusió, seria molt interessant dotar *Weston* de suport dels *writeback connectors*, si el sistema en disposa, i quan la implementació d'aquests hagi arribat oficialment a Linux, per tal de poder evitar desactivar els *hardware planes*, i per tant totes les millores que aquests suposen, quan es realitzen accions tan comunes com gravar o realitzar captures de pantalla.

16.3 Nou *allocator* que reemplaça GBM

Linux i Unix en general estan plens de la falta d'una forma estandarditzada per reservar memòria gràfica i qualsevol altre tipus de memòria de dispositius accelerats en l'espai d'usuari des de fa força temps.

Les interfícies de DRM de Linux només exposen mètodes de reservar memòria que són específics per cada venedor i dispositiu. Tot i que la infraestructura *DMA-BUF*[68] permet compartir aquest *buffers* gràfics entre diversos controladors de DRM, no hi ha una forma estàndard de reservar aquests *buffers* que garanteixi que els paràmetres (formats de píxel i modificadors) siguin els òptims o fins i tot compatibles.

[67] `https://lwn.net/Articles/703202/`
[68] `https://01.org/linuxgraphics/gfx-docs/drm/driver-api/dma-buf.html`

La llibreria GBM de Mesa i Gralloc[69] en Android s'han convertit en l'estàndard per defecte per reservar *buffers* gràfics, però no són suficientment expressives com per poder reservar memòria amb els paràmetres ideals per tal de treure el màxim rendiment dels components que utilitzaran aquests *buffers*.

Així doncs, la idea és crear una nova API, ja en procés[70], que permeti la reserva de *buffers* gràfics que puguin ser compartits entre diversos dispositius, com per exemple que els *frames* gravats per una càmera puguin ser utilitzats en els *hardware planes* de la GPU de forma directa, i que aquesta API sigui suficientment expressiva que permeti que els paràmetres d'aquests *buffers* siguin els òptims per part de tots els dispositius implicats.

Finalment s'hauria de modificar *Weston* per tal que utilitzi aquesta nova API en comptes de GBM.

16.4 High Dynamic Range (HDR) i *framebuffers* de punt flotant

En els darrers anys hi ha hagut un auge en augmentar tant les resolucions de les pantalles com en incrementar el rang de colors que aquestes poden mostrar, el que es coneix com a *High Dynamic Range* o HDR[71].

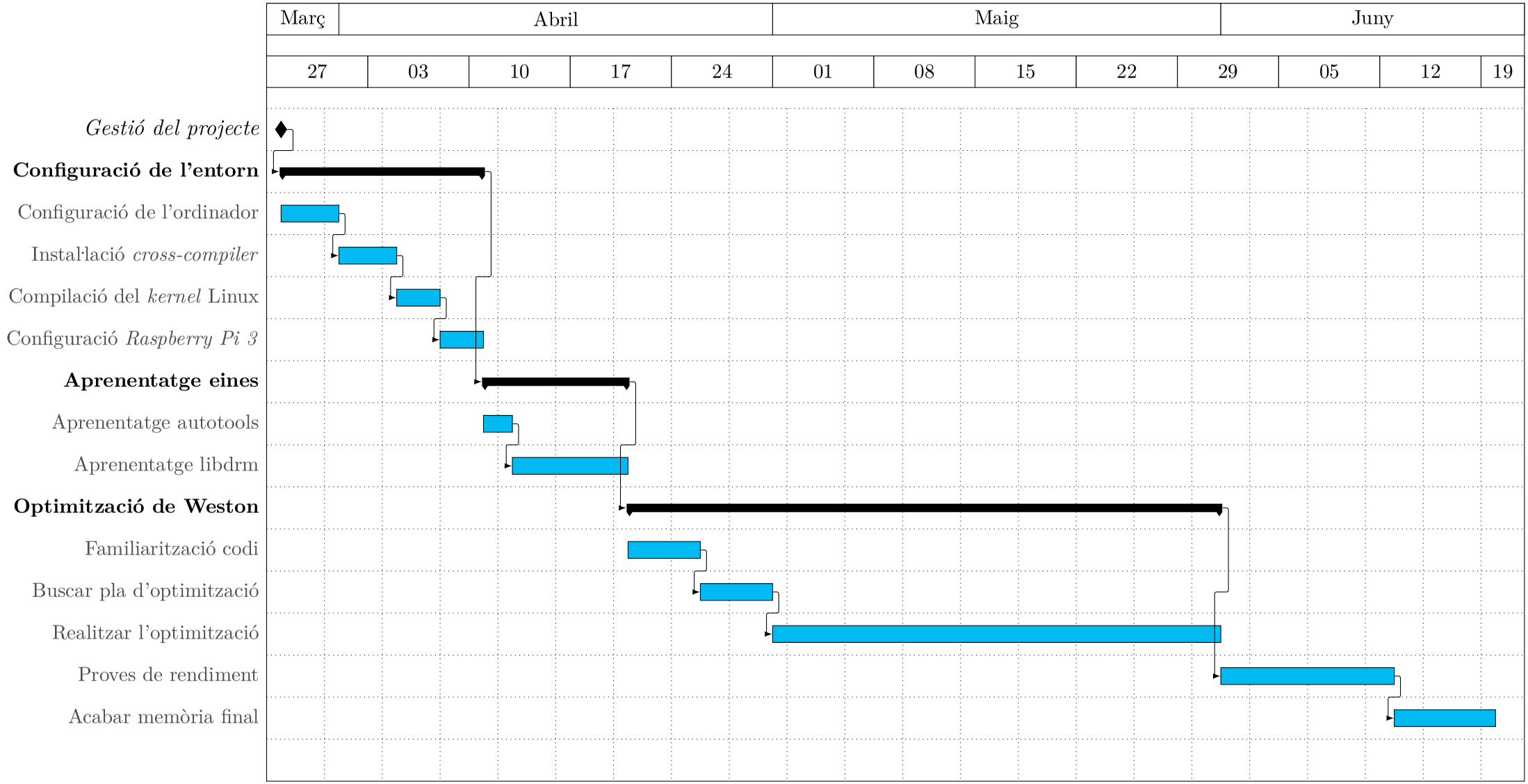
Per poder disposar d'aquest rang incrementat de colors s'acostuma a utilitzar *framebuffers* que no es representen els valors dels píxels en nombres enters, sinó que s'utilitzen números de punt flotant (normalment de mitja precisió, coneguts com a *FP16*).

La infraestructura de Linux s'ha quedat endarrerida en aquest aspecte[72], i en primer lloc DRM hauria d'obtenir si les pantalles o monitors suporten HDR mitjançant la informació disponible en l'EDID d'aquestes. En segon lloc, l'aplicació que realitza renderitzat HDR necessita d'una forma de proporcionar les metadades específiques del format HDR utilitzat cap al monitor.

A més a més, els compositors, i en particular *Weston* han de ser capaços de gestionar *framebuffers* FP16, i sobretot de gestionar de forma eficaç quan un client HDR ha de ser renderitzat cap al *primary plane*, ja que si el *framebuffer* d'aquest *primary plane* no és també FP16, es perdrà precisió.

[69] `https://source.android.com/devices/graphics/#gralloc`
[70] `https://github.com/cubanismo/allocator`
[71] `https://en.wikipedia.org/wiki/High_dynamic_range`
[72] `https://lwn.net/Articles/702563/`

A Diagrama de Gantt



B Instruccions de compilació

L'*atomic mode-setting* requereix les versions més noves del *kernel* de Linux. En cas de disposar una CPU d'Intel (controlador i915), l'*atomic mode-setting* s'activa per defecte a partir de la versió 4.12 de Linux, i en cas d'utilitzar una versió més antiga es pot activar passant el paràmetre `i915.nuclear_pageflip=1` quan s'inicia el sistema.

Si es disposa d'una versió inferior a la 4.7 aproximadament, és molt recomanable compilar l'última versió de Linux, tot seguint aquestes[73] instruccions si s'utilitza Ubuntu o derivats, o aquestes[74] en el cas d'Arch Linux.

La millora que introdueix suport dels modificadors de format de píxel requereix d'uns *patches* que encara no s'han aplicat al *kernel* oficial. El repositori *git* on els podem trobar és aquest[75].

El suport de les extensions d'Android relacionades amb l'*explicit fencing* es va acabar d'afegir en aquest[76] *commit*, així que també és molt recomanable compilar manualment l'última versió de Mesa.

Tant per la compilació de Mesa (i *libdrm*) com de *Wayland*, *Weston*, i les seves dependències es poden seguir aquestes[77] instruccions, tot i que amb el matís que la versió de *Weston* sobre la qual he realitzat el projecte ja no utilitza les eines de compilació *autotools* sinó que utilitza *Meson*[78] i *Ninja*[79]. Així doncs per poder compilar la versió pròpia de *Weston* cal anar a la carpeta on aquest es troba (carpeta `programes/weston`) i executar les següents comandes:

```
$ mkdir build
$ cd build
$ meson
$ ninja install
```

[73] `https://wiki.ubuntu.com/KernelTeam/GitKernelBuild`
[74] `https://wiki.archlinux.org/index.php/Kernels/Arch_Build_System`
[75] `https://cgit.freedesktop.org/~bwidawsk/drm-intel/log/?h=blobifier-v5`
[76] `https://cgit.freedesktop.org/mesa/mesa/commit/?id=6403e3765114b4f540ed1f5a714fbf1ff4d1bedc`
[77] `https://wayland.freedesktop.org/building.html`
[78] `http://mesonbuild.com/`
[79] `https://ninja-build.org/`

El programa de prova *atomictest* té dues versions: la no atòmica la trobem dins la carpeta `programes/atomictest_no_atomic`, i l'atòmica, que la podem localitzar a `programes/atomictest`. Per poder compilar aquest programa es necessita *autotools* i algunes de les dependències mencionades a les instruccions de compilació de *Weston*, en particular, la llibreria *libinput*. Un cop dins la carpeta d'*atomictest* corresponent, els passos a seguir per compilar-lo són els següents:

```
$ autoreconf -i
$ ./configure
$ make
```

I per executar-lo (des d'una consola virtual[80]):

```
$ ./src/atomictest
```

Cal mencionar que si executem *atomictest* amb permisos de superusuari (*sudo*), el programa gaudirà de permisos suficients per poder obrir i utilitzar els dispositius d'entrada (ratolí i teclat) que *libinput* necessita.

A la carpeta `benchmarks` a més a més de trobar-hi els resultats de les proves de rendiment (`results_atomictest.txt` i `results_render_target_writes.txt`) també hi trobem les modificacions en format *diff* que s'han hagut de fer sobre les eines `intel-gpu-tools` (`intel-gpu-tools.diff`) i sobre el programa de *Weston* `simple-egl` (`simple-egl.diff`), a més a més de l'*script Python*[81] `render_target_writes.py` que realitza les execucions amb diferents resolucions de `simple-egl`. Per tal d'aplicar aquests *diffs* es pot utilitzar l'eina *git apply*[82].

[80] `https://en.wikipedia.org/wiki/Virtual_console`
[81] `https://www.python.org/`
[82] `https://git-scm.com/docs/git-apply`

Referències

[1] Georgi Dalakov. *The NLS system of Douglas Engelbart*. http://history-computer.com/Internet/Birth/EngelbartNLS.html. [Últim accés: febrer de 2017].

[2] Frank McCown. *History of the Graphical User Interface (GUI)*. https://www.harding.edu/fmccown/gui/history-gui.pptx. Harding University. [Últim accés: febrer de 2017].

[3] Pàgina web oficial. *Past Research in the DSG*. http://gregorio.stanford.edu/History.html. [Últim accés: febrer de 2017].

[4] William I. Nowicki. *Partitioning of Function in a Distributed Graphics System*. http://bitsavers.org/pdf/stanford/v-system/nowicki_CSL-85-282_VGTS.pdf, 1985. [Últim accés: febrer de 2017].

[5] Karin Cheung i Carol Chow i Mike Li i Jesse Koontz i Ben Self. *Project Athena*. http://web.mit.edu/6.933/www/final.pdf, 1999. [Últim accés: febrer de 2017].

[6] Robert W. Scheifler. *Debut of X*. http://www.talisman.org/x-debut.shtml, 1984. [Últim accés: febrer de 2017].

[7] The Open Group. *The Single UNIX Specification*. http://www.unix.org/what_is_unix/single_unix_specification.html. [Últim accés: febrer de 2017].

[8] freedesktop.org. *DRI Wiki*. https://dri.freedesktop.org/wiki/, 2011. [Últim accés: febrer de 2017].

[9] freedesktop.org. Direct Rendering Manager (DRM). https://dri.freedesktop.org/wiki/DRM/, 2014. [Últim accés: febrer de 2017].

[10] Desenvolupadors del kernel de Linux. *Linux GPU Driver Developer's Guide, Introduction*. https://www.kernel.org/doc/html/latest/gpu/introduction.html, 2016. [Últim accés: febrer de 2017].

[11] Desenvolupadors de X.Org. *X.Org*. https://www.x.org/wiki, 2010. [Últim accés: febrer de 2017].

[12] Daniel Vetter. *Atomic mode setting design overview, part 1*. https://lwn.net/Articles/653071/, 2015. [Últim accés: febrer de 2017].

[13] Kristian Høgsberg. *Premature publicity is better than no publicity.* http://hoegsberg.blogspot.com.es/2008/11/premature-publicity-is-better-than-no.html, 2008. [Últim accés: febrer de 2017].

[14] Autors de la Raspberry Pi. *Raspberry Pi.* https://www.raspberrypi.org/. [Últim accés: febrer de 2017].

[15] Gustavo Padovan. *Collabora Contributions to Linux Kernel 4.10.* http://padovan.org/blog/2017/02/collabora-contributions-to-linux-kernel-4-10/, 2017. [Últim accés: febrer de 2017].

[16] Desenvolupadors d'Android. *Implementing the Hardware Composer HAL.* http://source.android.com/devices/graphics/implement-hwc.html, 2017. [Últim accés: febrer de 2017].

[17] Brian Starkey. *Introduce writeback connectors.* https://lwn.net/Articles/704647/, 2016. [Últim accés: febrer de 2017].

[18] Chris Michael. *Ecore_Drm2: How to Use Atomic Modesetting.* https://blogs.s-osg.org/ecore_drm2-use-atomic-modesetting/, 2016. [Últim accés: febrer de 2017].

[19] Desenvolupadors de sshfs. *sshfs: A network filesystem client to connect to SSH servers.* https://github.com/libfuse/sshfs. [Últim accés: febrer de 2017].

[20] Diari ABC. *Barcelona, capital «startup» española.* http://www.abc.es/espana/catalunya/economia/abci-barcelona-capital-startup-espanola-201701111753_noticia.html. [Últim accés: març de 2017].

[21] Bryce Harrington i desenvolupadors de Weston. *[ANNOUNCE] weston 1.12.0.* https://lists.freedesktop.org/archives/wayland-devel/2016-September/031123.html, 2016. [Últim accés: febrer de 2017].