



# Universidad Austral de Chile

Facultad de Ciencias de la Ingeniería  
Escuela de Ingeniería Civil en Informática

## LOUVRE: LIBRERÍA C++ PARA CREAR COMPOSITORES WAYLAND

Proyecto para optar al título de  
**Ingeniero Civil en Informática**

PROFESOR PATROCINANTE:

CRISTÓBAL ALEJANDRO NAVARRO GUERRERO  
INGENIERO CIVIL EN INFORMÁTICA  
DOCTOR EN CIENCIAS DE LA COMPUTACIÓN  
POST-DOCTORADO ANID EN GPU COMPUTING

PROFESORA INFORMANTE:

ELIANA SCHEIHING GARCÍA  
INGENIERA CIVIL MATEMÁTICA  
D.E.A EN MATHEMATIQUE  
DOCTEUR EN STATISTIQUE

PROFESORA INFORMANTE:

VALERIA HENRÍQUEZ  
INGENIERA CIVIL EN INFORMÁTICA  
DOCTORA EN SOFTWARE, SISTEMAS Y COMPUTACIÓN  
MAGISTER EN DIRECCIÓN DE MARKETING DIRECTO Y DIGITAL

**Eduardo Alfredo Hopperdietzel Ralph**

VALDIVIA – CHILE  
2022

## **DEDICATORIA**

Quiero agradecer a Juan Latorre Harcha, con quien compartimos hace algunos años la idea de crear un sistema operativo libre, gratuito e intuitivo, lo cual ha sido en gran parte motivación para llevar a cabo este trabajo, el cual estoy seguro que será una herramienta útil para cualquiera con el mismo anhelo.

También quiero agradecer a Diego Sandoval, con quien tuve la suerte de compartir este interés y tener conversaciones tanto inspiradoras como enriquecedoras durante todo el proceso. Le agradezco también por tomarse el tiempo al igual que a Felipe Quezada para probar la librería y proporcionar retroalimentación que permitió identificar varios aspectos a mejorar.

Finalmente, agradezco enormemente a mi familia por permitirme estudiar esta carrera, por dejarme ser y por haberme apoyado todos estos años.

## ÍNDICE

ÍNDICE DE TABLAS .....	IV
ÍNDICE DE FIGURAS .....	V
RESUMEN.....	VII
ABSTRACT .....	VIII
1. INTRODUCCIÓN .....	1
1.1 Wayland .....	1
1.1.1 Arquitectura y diferencias con Xorg .....	1
1.1.2 Complejidad de Wayland.....	3
1.2 Librerías alternativas .....	5
1.3 Impacto en Linux .....	5
1.4 Objetivo general y específicos .....	6
1.4.1 Objetivo general .....	6
1.4.2 Objetivos específicos .....	6
1.4.3 Resultados verificables.....	6
2. MARCO TEÓRICO .....	7
2.1 Patrones de diseño de APIs .....	7
2.1.1 Observer .....	7
2.1.2 Factory Methods .....	7
2.1.3 Pimpl Idiom.....	9
2.1.4 Singleton .....	9
2.2 Prácticas para el diseño de APIs .....	10
2.2.1 Prefijos .....	10
2.2.2 Templates .....	11
2.2.3 Operadores .....	11
2.2.4 Llave const .....	11
2.2.5 Argumentos por referencia o punteros .....	12
2.2.6 Argumentos por defecto.....	12
2.2.7 Bitfields y enums .....	12
2.3 Características de buenos diseños de APIs .....	13
2.4 Evaluación de usabilidad de APIs.....	14
3. ARQUITECTURA PROPUESTA.....	15
3.1 Componentes.....	15
3.1.1 Manejo de solicitudes de clientes.....	16
3.1.2 Protocolos.....	16
3.1.3 Backends gráficos .....	19
3.1.4 Backends de input .....	21
3.2 Fábrica de recursos.....	23
3.3 Inicialización del compositor .....	23

3.4 Dependencias e instalación .....	25
4. CLASES.....	27
4.1 Clase LComposer .....	27
4.2 Clase LOutput .....	27
4.2.1 Acceso a las salidas disponibles.....	30
4.2.2 Inicialización .....	30
4.2.3 Modos de una salida.....	30
4.2.4 Contextos gráficos.....	30
4.2.5 Layout .....	31
4.2.6 De-inicialización .....	31
4.3 Clase LOutputManager .....	31
4.4 Clase LOutputMode .....	32
4.5 Clase LPainter .....	32
4.6 Clase LOpenGL .....	33
4.7 Clase LTexture .....	33
4.7.1 Mecanismos de intercambio de buffers.....	34
4.8 Clase LSurface .....	35
4.8.1 Roles.....	36
4.8.2 Buffers.....	36
4.8.3 Mapeo.....	37
4.8.4 Daños .....	37
4.8.5 Callbacks .....	37
4.8.6 Orden.....	37
4.8.7 Posición.....	38
4.9 Clase LBaseSurfaceRole .....	38
4.10 Clase LCursorRole .....	39
4.11 Clase LDNDIconRole .....	39
4.12 Clase LSubsurfaceRole .....	40
4.12.1 Modos.....	40
4.13 Clase LToplevelRole.....	41
4.13.1 Geometría de ventana.....	41
4.14 Clase LPopupRole.....	43
4.15 Clase LPositioner .....	44
4.15.1 Recta de anclaje.....	45
4.15.2 Gravedad .....	45
4.15.3 Ajustes de restricción .....	46
4.16 Clase LSeat.....	46
4.17 Clase LPointer .....	47
4.18 Clase LKeyboard.....	48
4.19 Clase LCursor .....	48
4.19.1 Inicialización .....	49
4.19.2 Composición vía hardware.....	49
4.20 Clase LXCursor.....	49
4.21 Clase LClient.....	50
4.22 Clase LDataDevice.....	50

4.23 Clase LDataSource.....	51
4.24 Clase LDataOffer .....	51
4.25 Clase LDNDManager .....	52
4.26 Clase LLog .....	53
4.27 Clase LTime .....	53
4.28 Clase LPointTemplate .....	54
4.29 Clase LRectTemplate .....	55
4.30 Clase LRegion.....	56
<b>5. RENDIMIENTO .....</b>	<b>57</b>
<b>5.1 Benchmark .....</b>	<b>57</b>
5.1.1 Cálculo de consumo de CPU .....	58
5.1.2 Cálculo de consumo de GPU .....	58
5.1.3 Cálculo de cuadros por segundo .....	58
5.1.4 Ejecución.....	58
5.1.5 Hardware de prueba .....	59
<b>5.2 Resultados .....</b>	<b>59</b>
5.2.1 Cuadros por segundo.....	59
5.2.2 Consumo de CPU.....	62
5.2.3 Consumo de GPU.....	64
5.2.4 Conclusiones .....	65
<b>6. USABILIDAD .....</b>	<b>66</b>
<b>6.1 Resultados .....</b>	<b>66</b>
<b>7. CONCLUSIONES .....</b>	<b>68</b>
<b>7.1 Trabajos futuros .....</b>	<b>69</b>
<b>REFERENCIAS .....</b>	<b>70</b>
<b>ANEXOS .....</b>	<b>72</b>

## ÍNDICE DE TABLAS

<b>Tabla</b>	<b>Página</b>
Tabla 1: Consumo energético general de un sistema al utilizar Wayland y Xorg. ....	3
Tabla 2: Solicitudes del protocolo Wayland. ....	17
Tabla 3: Solicitudes del protocolo XDG Shell.....	18
Tabla 4: Solicitudes del protocolo XDG Decoration. ....	19
Tabla 5: Solicitudes del protocolo Pointer Gestures. ....	19
Tabla 6: Dependencias de Louvre.....	25
Tabla 7: Niveles de verbosidad de LLog. ....	53
Tabla 8: Características de las máquinas utilizadas en el benchmark.....	59

## ÍNDICE DE FIGURAS

Figura	Página
Figura 1: Arquitectura de Xorg y Wayland.	2
Figura 2: Ejemplo de constructor virtual.	8
Figura 3: Ejemplo del patrón Singleton.	10
Figura 4: Ejemplo de operadores.	11
Figura 5: Ejemplo de bitfields y enums.	13
Figura 6: Diagrama de componentes de Louvre.	15
Figura 7: Capas de gestión de solicitudes de clientes en Louvre.	16
Figura 8: Ejemplo de carga de backends.	20
Figura 9: Secuencia de creación, uso y destrucción de una superficie.	23
Figura 10: Secuencia de inicialización del compositor.	24
Figura 11: Diagrama de paquetes de Louvre.	27
Figura 12: Miembros de la clase LCompositor.	28
Figura 13: Layout de múltiples salidas.	29
Figura 14: Miembros de la clase LOutput.	29
Figura 15: Layout de múltiples salidas en un panel de preferencias.	31
Figura 16: Miembros de la clase LOutputManager.	32
Figura 17: Miembros de la clase LOutputMode.	32
Figura 18: Miembros de la clase LPainter.	33
Figura 19: Miembros de la clase LOpenGL.	33
Figura 20: Miembros de la clase LTexture.	34
Figura 21: Intercambio de buffers gráficos mediante memoria compartida.	34
Figura 22: Intercambio de buffers gráficos mediante DMA.	35
Figura 23: Miembros de la clase LSurface.	35
Figura 24: Clases que heredan a LBaseSurfaceRole.	36
Figura 25: Miembros de la clase LBaseSurfaceRole.	38
Figura 26: Miembros de la clase LCursorRole.	39
Figura 27: Miembros de la clase LDNDIconRole.	39
Figura 28: Miembros de la clase LSubsurfaceRole.	40
Figura 29: Ejemplo de superficie Toplevel.	41
Figura 30: Miembros de la clase LToplevelRole.	42
Figura 31: Geometría de ventana de una superficie Toplevel o Popup.	43
Figura 32: Ejemplo de superficies Popups anidadas.	43
Figura 33: Miembros de la clase LPopupRole.	44
Figura 34: Miembros de la clase LPositioner.	44
Figura 35: Posición del Popup según el punto de anclaje de la recta de anclaje.	45
Figura 36: Cambio de posición del Popup según su gravedad.	46
Figura 37: Miembros de la clase LSeat.	47
Figura 38: Miembros de la clase LPointer.	47
Figura 39: Miembros de la clase LKeyboard.	48

Figura 40: Miembros de la clase LCursor.....	48
Figura 41: Miembros de la clase LXCursor.....	49
Figura 42: Miembros de la clase LClient.....	50
Figura 43: Miembros de la clase LDataDevice.....	51
Figura 44: Miembros de la clase LDataSource.....	51
Figura 45: Miembros de la clase LDataOffer. ....	52
Figura 46: Miembros de la clase LDNDManager.....	52
Figura 47: Miembros de la clase LLog. ....	53
Figura 48: Miembros de la clase LTime. ....	53
Figura 49: Miembros de la clase LPointTemplate. ....	54
Figura 50: Miembros de la clase LRectTemplate. ....	55
Figura 51: Miembros de la clase LRegion. ....	56
Figura 52: Superficies del benchmark siendo renderizadas en Weston.....	57
Figura 53: Cuadros por segundo de Louvre y Weston en la máquina Dell. ....	59
Figura 54: Cuadros por segundo de Louvre y Weston en la máquina Apple. ....	60
Figura 55: Ilustración del efecto tearing. ....	60
Figura 56: Sincronización de hilo principal con un hilo de renderizado. ....	61
Figura 57: Consumo de CPU de Louvre y Weston en la máquina Dell. ....	62
Figura 58: Artefactos generados en Weston debido a falta de precisión. ....	63
Figura 59: Consumo de CPU de Louvre y Weston en la máquina Apple.....	64
Figura 60: Consumo de GPU de Louvre y Weston en la máquina Dell. ....	64
Figura 61: Consumo de GPU de Louvre y Weston en la máquina Apple. ....	65

## **RESUMEN**

El desarrollo de compositores Wayland es actualmente una tarea compleja que requiere una larga curva de aprendizaje. Aunque las librerías existentes simplifican algunas interfaces y facilitan el acceso a eventos de entrada, el control de monitores y funciones de renderizado, el desarrollador aún tiene que implementar la mayoría de las solicitudes de los protocolos de Wayland, lo que implica aprender las reglas de cada una y escribir cientos de líneas de código antes de poder validar el progreso.

Para abordar este problema, se propone una librería con una arquitectura basada en el patrón de diseño Factory y en el uso de métodos virtuales, con dos capas para procesar las solicitudes de clientes, donde en la primera se preprocesan, simplifican y se envían a la segunda mediante la invocación de métodos virtuales con una implementación predeterminada. Con esta estrategia, el desarrollador puede ver en pantalla un compositor funcional desde un inicio del desarrollo, y reimplementar los métodos virtuales de manera iterativa, sin necesidad de conocer todas las reglas de los protocolos con anticipación y con la posibilidad de validar su progreso frecuentemente. El uso de dos capas permite a la librería encargarse de manejar correctamente un gran numero de solicitudes presentes en los protocolos, disminuyendo así en un 79% la probabilidad de que el desarrollador cometiera errores que invaliden la compatibilidad del compositor con las aplicaciones en Linux.

La implementación de esta arquitectura y metodología busca disminuir la curva de aprendizaje para desarrollar compositores Wayland, simplificar el proceso y permitir a los desarrolladores centrarse en la lógica de sus compositores.

## ABSTRACT

Currently, the development of Wayland compositors is a complex task that requires a steep learning curve. While existing libraries simplify some interfaces and provide easier access to input events, displays control, and rendering utilities, developers still have to implement most of the Wayland protocol requests by themselves, which involves learning the rules for each and writing hundreds of lines of code before any progress can be validated.

To address this issue, a library with an architecture based on the Factory design pattern and the use of virtual methods is proposed, with two layers for handling client requests. The first layer preprocesses requests, and the second layer uses virtual methods with a default implementation. This approach allows developers to see a functional compositor from the early stage of development and iteratively reimplement virtual methods without needing to know all protocol rules in advance and with the ability to constantly validate progress. The use of two layers allows the library to correctly handle a large number of requests present in the protocols, reducing the likelihood of the developer committing errors that invalidate the compatibility of the compositor with Linux applications by 79%. The implementation of this architecture and methodology aims to decrease the learning curve for developing Wayland compositors, simplify the process, and allow developers to focus on the logic of their compositors.

# 1. INTRODUCCIÓN

## 1.1 Wayland

En 2008 surge Wayland, un nuevo protocolo de servidor gráfico utilizado principalmente en Linux y desarrollado para reemplazar a Xorg, el cual se compone tres componentes principales (Freedesktop Organization, 2012-a):

1. Un **lenguaje** basado en XML que permite describir interfaces de protocolos de comunicación entre aplicaciones y servidores gráficos y traducirlas a lenguaje C mediante su herramienta wayland-scanner.
2. Un **protocolo** descrito en su propio lenguaje con interfaces que brindan funcionalidades básicas como permitir a aplicaciones enviar sus buffers gráficos al servidor, o permitir a este último enviarles eventos de entrada, características de los monitores disponibles, entre otros tipos de mensajes.
3. Una **librería** escrita en C que facilita la implementación de las interfaces de los protocolos y ofrece un mecanismo de comunicación entre procesos mediante Unix Domain Sockets para que las aplicaciones y el servidor se puedan comunicar.

### 1.1.1 Arquitectura y diferencias con Xorg

La arquitectura de Wayland fue diseñada con el objetivo de simplificar el desarrollo, extensión y mantenimiento de aplicaciones y servidores gráficos, solucionando desde la base varias problemáticas presentes en la arquitectura de Xorg que no han podido ser resueltas de forma óptima en el tiempo, como la sincronización vertical, seguridad, extensibilidad y mantenibilidad (Høgsberg, Chapter 3. Wayland Architecture, 2012).

Para comprender en detalle la principal diferencia entre sus arquitecturas, es necesario primero tener claridad sobre que es un servidor gráfico y que es un compositor:

1. Un servidor gráfico, es un proceso que juega un rol intermediario entre las aplicaciones con interfaz gráfica (clientes) y el kernel. Por lo general es el único proceso con permisos para acceder al input del sistema y controlar monitores en una sesión, y es quién decide, bajo su propia política, a qué aplicación enviar eventos de entrada y cuales mostrar en pantalla (Wikipedia, 2022-a) esto con el fin de multiplexar los recursos del sistema.
2. Por otro lado, un compositor se encarga del renderizado y preprocesamiento de la salida gráfica, en conjunto con el servidor gráfico definen qué es lo que se muestra en pantalla y cómo se visualizan las aplicaciones. Para un compositor, las ventanas de las aplicaciones representan texturas que este puede manipular y mostrar en pantalla de múltiples formas.

Los compositores comúnmente utilizados renderizan en dos dimensiones y añaden decoraciones como efectos de sombreado, transparencia, difuminado, una barra de título, etc. Pero también existen algunos más sofisticados como los entornos en tres dimensiones, de realidad aumentada, virtual, etc (Wikipedia, 2022-b).

La principal diferencia entre la arquitectura de Xorg y Wayland es que en el primero, el servidor gráfico y el compositor son dos procesos distintos, mientras que en último ambos forman parte de un mismo proceso como se muestra en la Figura 1.

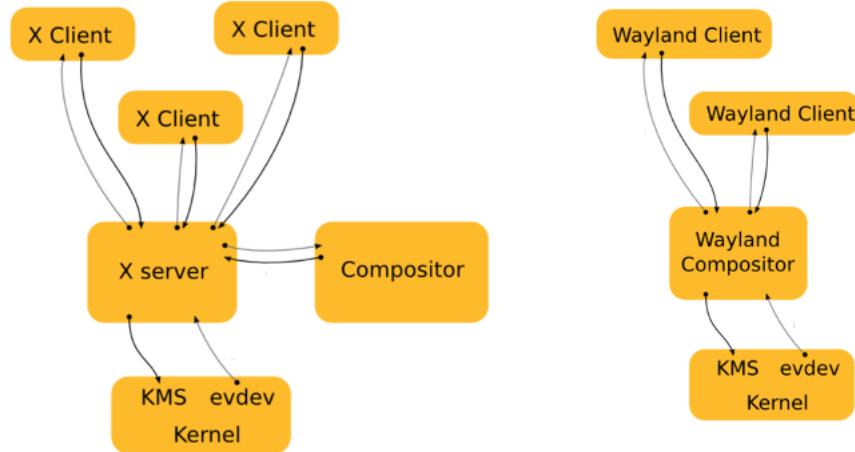


Figura 1: Arquitectura de Xorg y Wayland.<sup>1</sup>

Esto significa que en Xorg existe un mayor intercambio de información entre procesos, lo que aumenta la complejidad al momento de desarrollar un compositor y además añade un overhead innecesario que disminuye el rendimiento del sistema.

La razón de que el servidor gráfico y compositor estén separados en Xorg es debido a que su diseño inicial (1987) no contemplaba todos los requerimientos de entornos y aplicaciones gráficas modernas y por lo tanto no contaba con un compositor. Este último fue añadido posteriormente mediante extensiones a la librería original y de forma no óptima lo que dio origen a X-Composite (Packard & Johnson, 2007).

En la Tabla 1 se puede observar los resultados de un benchmark en el cual se midió el consumo energético general de un sistema utilizando los entornos de escritorio GNOME y KDE en sus versiones Wayland y Xorg, del cual se concluyó que en Wayland se consume en promedio 3 watts menos que en Xorg.

---

<sup>1</sup>Imagen disponible en <https://wayland.freedesktop.org/architecture.html>. Consultado el 16 de octubre de 2022.

Tabla 1: Consumo energético general de un sistema al utilizar Wayland y Xorg.<sup>2</sup>

Entorno de escritorio	Mínimo	Media	Máximo
GNOME Wayland	8.896 W	21.020 W	29.498 W
GNOME Xorg	4.814 W	24.412 W	29.981 W
KDE Wayland	6.610 W	21.150 W	29.416 W
KDE Xorg	7.608 W	23.795 W	29.816 W

Por estas razones Wayland es considerado el futuro en lo que a gráficos respecta en Linux y cada vez más entornos de escritorio como GNOME o KDE y varios toolkits gráficos como Qt y GTK han decidido incorporarlo. La prominencia de esta tecnología explica por qué en este trabajo se profundiza en dicha tecnología y no en Xorg.

### 1.1.2 Complejidad de Wayland

Como en Wayland el compositor y servidor gráfico conforman un mismo proceso, en adelante utilizará el término “compositor” para referirse a dicho proceso, o sea, al conjunto de ambos términos y se dejará a un lado su distinción.

También, se utilizará el término “solicitud” para referirse a un mensaje enviado desde una aplicación al compositor y el término “evento” para referirse a un mensaje enviado desde el compositor a una aplicación o desde el sistema al compositor.

Si bien la arquitectura de Wayland se enfoca en facilitar el desarrollo, extensión y mantenimiento de aplicaciones y compositores, sigue siendo una labor compleja desarrollar este último. La librería C de Wayland facilita la implementación de las interfaces de los protocolos, permitiendo al compositor escuchar solicitudes como también enviar eventos. Sin embargo, las reglas descritas en dichos protocolos sobre cómo debe actuar el compositor al recibir una solicitud o las aplicaciones al recibir eventos deben ser implementadas manualmente.

Cada protocolo contiene generalmente múltiples interfaces y en cada interfaz se definen y describen múltiples solicitudes y eventos. Por ejemplo, el protocolo de Wayland (a la fecha), cuenta con 22 interfaces, cada una destinada a una funcionalidad en específico y con una media de seis mensajes, lo que suma específicamente un total de 65 solicitudes y 58 eventos en todo el protocolo (Høgsberg, Wayland Protocol, 2022). Aún con esa cantidad de mensajes no alcanza a cubrir todas las funcionalidades típicamente requeridas por aplicaciones y entornos de escritorio modernos. Por esta razón, han surgido otros protocolos adicionales, comúnmente denominados “extensiones de Wayland” como XDG Shell, WP Presentation Time, WP Viewporter, y muchos otros que añaden funcionalidad extra al compositor, siendo algunos requeridos por gran número de aplicaciones.

---

<sup>2</sup> Datos extraídos desde <https://openbenchmarking.org/result/1805126-AR-KDEGNOMEW67>. Consultado el 16 de octubre de 2022.

Las interfaces de los protocolos también pueden evolucionar en el tiempo por lo que contienen un número entero positivo que incrementa con cada nueva versión (Høgsberg, Chapter 4. Wayland Protocol and Model of Operation, 2012). Cuando un cliente se conecta a un compositor, este le informa sobre todas las interfaces que posee implementadas y las versiones máximas que soporta, las aplicaciones luego utilizan versiones específicas de dichas interfaces cuyos números no pueden superar las versiones máximas soportadas por el compositor.

Por ejemplo la interfaz `wl_pointer` del protocolo de Wayland en su primera versión solo contaba con el método `send_axis()` para enviar eventos de scroll a los clientes, a partir de la quinta versión el compositor debe especificar primero el origen del evento con `send_axis_source()`, indicar su finalización con `send_axis_stop()`, definir el paso de la rueda de scroll con `send_axis_discrete()` y luego de cada conjunto de eventos de scroll enviar `send_frame()`. Actualmente en la octava versión, no se debe enviar el evento `send_axis_discrete()`, en cambio se debe utilizar `send_axis_value120()`.

Todo esto supone un problema al desarrollar un compositor Wayland, ya que se debe implementar todas las solicitudes y eventos de los protocolos correctamente en cada una de sus versiones, cuyas reglas en muchos casos son descritas de manera breve y ambigua, siendo necesario experimentar y realizar múltiples pruebas antes de lograr implementarlas correctamente.

Otra característica de Wayland es que no establece ningún mecanismo o solución para que el compositor pueda acceder a eventos de entrada del sistema o desplegar su contenido en pantalla (Høgsberg, Chapter 3. Wayland Architecture, 2012). Esto por un lado es positivo, ya que otorga flexibilidad para elegir la fuente generadora de eventos de entrada (backend de input) o el contexto sobre el cual se visualizará el compositor (backend gráfico). Por ejemplo, se podría utilizar una cámara que mediante el movimiento de las manos permita controlar un cursor en un sistema embebido o desplegar el compositor visualmente sobre una ventana en un servidor Xorg, sobre otro compositor Wayland anidado, directamente en pantalla utilizado las APIs de Linux, etc. Generalmente en entornos de escritorio Linux es preferible acceder a los eventos de entrada de los dispositivos que componen la máquina siendo utilizada como el touchpad, ratón, teclado, etc. Para esto se debe interactuar con algunas APIs de Linux como UDEV y EVDEV o utilizar alguna librería de mayor nivel como Libinput (The Libinput Team, 2019). Respecto a los gráficos, es preferible desplegar el compositor directamente en pantalla y renderizar de forma acelerada utilizando las GPUs presentes en el hardware. Para ello, es necesario interactuar con las APIs UDEV, GBM, KMS y DRM de Linux y utilizar librerías de renderizado como EGL y OpenGL (Linux Kernel Development Community, s.f-a).

Lamentablemente esto también supone un desafío para el desarrollador, ya que gran parte de las APIs en espacio usuario de Linux poseen escasa documentación, siendo necesario indagar exhaustivamente, buscar ejemplos de implementaciones similares, realizar reiteradas pruebas para implementarlas correctamente y finalmente definir una arquitectura que unifique todas las partes de forma eficiente.

## 1.2 Librerías alternativas

Debido a las problemáticas antes mencionadas, han surgido algunas librerías de mayor nivel como QtWayland y wl\_roots (las más robustas y ampliamente utilizadas actualmente) que están basadas en la librería de Wayland y ofrecen sus propios backends de entrada y gráficos, así como una arquitectura bien definida y varias funcionalidades que facilitan el uso de interfaces de protocolos. Sin embargo, utilizan el patrón *Observer*<sup>3</sup> y por lo tanto, no proporcionan una forma predeterminada de manejar las solicitudes de clientes descritas en las reglas de los protocolos. Esto significa que el desarrollador debe suscribirse manualmente a cada solicitud para poder manejarla, lo que requiere un conocimiento profundo y previo de sus reglas, y fuerza a escribir cientos de líneas de código antes de ver resultados en pantalla y validar que la implementación desarrollada funciona correctamente.

### QtWayland

QtWayland por un lado está escrita en C++, cuenta con un amplio set de clases para representar las interfaces de los protocolos de Wayland (The Qt Company, 2022-a), permitiendo también implementar interfaces adicionales mediante extensiones y utiliza el patrón de diseño *Observer* (mediante *signals* y *slots*) para poder suscribirse a las solicitudes realizadas por clientes (The Qt Company, 2022-b). También utiliza múltiples hilos de renderizado para mantener una alta tasa de cuadros por segundo al utilizar sincronización vertical (The Qt Company, 2022-c) y cuenta con buena documentación y varios ejemplos.

### wl\_roots

wl\_roots por otro lado, está escrita en C, también utiliza el patrón de diseño *Observer*, se caracteriza por su alta modularidad, utiliza un único hilo, es estricta respecto a la implementación de los protocolos, pero su API posee escasa documentación (wlroots, 2022).

## 1.3 Impacto en Linux

Pese a las ventajas que ofrece la arquitectura de Wayland (GamingOnLinux, 2023), no es de extrañar que se sigan utilizando principalmente entornos de escritorios basados en Xorg y que existan aún pocos compositores Wayland innovadores y completamente funcionales (Slant, 2022).

Un bajo nivel de innovación podría traer consecuencias negativas para la comunidad de usuarios de Linux, quienes probablemente se acostumbren a realizar tareas cotidianas de formas poco productivas, siendo esto además una barrera para nuevos entrantes con bajos

---

<sup>3</sup> El patrón Observer es un patrón de diseño de software que permite que objetos de una aplicación se suscriban a notificaciones de eventos o cambios de estado en otros objetos y se actualicen automáticamente.

conocimientos informáticos, quienes seguramente prefieran utilizar sistemas operativos privativos como Windows o macOS.

Finalmente, un bajo número de usuarios podría disminuir el interés de individuos para contribuir en Linux y el de empresas desarrolladoras de software para ofrecer versiones de sus productos para la plataforma

## **1.4 Objetivo general y específicos**

### **1.4.1 Objetivo general**

Mejorar la experiencia de usuario en Linux, promoviendo la innovación de entornos de escritorio mediante una librería escrita en C++ que facilite la creación de compositores Wayland.

### **1.4.2 Objetivos específicos**

1. Permitir crear compositores computacionalmente eficientes.
2. Reducir el tiempo y esfuerzo requerido para crear compositores compatibles con aplicaciones desarrolladas con toolkits gráficos que soportan Wayland en Linux.

### **1.4.3 Resultados verificables**

Para evaluar el cumplimiento del primer objetivo específico, se creará un compositor con Louvre que imite visualmente a Weston (el compositor de referencia desarrollado por los creadores de Wayland) y se realizará una prueba de benchmark que mida el uso de CPU en el espacio de usuario, el consumo de GPU en watts y el número de cuadros por segundo al que ambos pueden renderizar. La meta es lograr que el compositor desarrollado con Louvre tenga un rendimiento igual o superior que el de Weston en los tres indicadores.

Para evaluar el cumplimiento del segundo objetivo específico, se considera la probabilidad de que el desarrollador cometa errores al manejar solicitudes de los clientes presentes en las interfaces de los protocolos de Wayland, y requeridos por las aplicaciones en Linux. Esta probabilidad es un indicador de la cantidad de reglas de protocolos que el desarrollador debe conocer correctamente para crear un compositor funcional y se puede calcular contando el número de solicitudes que dependen de una correcta implementación, dividido entre el total de solicitudes. Se establece como meta reducir esta probabilidad en al menos un 50%.

Además, se solicitará a dos informáticos que desarrolleen su propio compositor con Louvre (de los cuales uno posee experiencia con otras librerías) para que posteriormente completen un formulario de evaluación de usabilidad de APIs C++.

## 2. MARCO TEÓRICO

Un aspecto fundamental a considerar para cumplir con los objetivos propuestos en este proyecto se relaciona con la usabilidad de la librería desarrollada. Por esta razón se describen a continuación patrones y prácticas relacionadas al diseño de APIs C++ y formas de medir su usabilidad.

### 2.1 Patrones de diseño de APIs

Reddy (2011), describe una serie de patrones de diseño aplicables a APIs C++, de los cuales varios han sido de utilidad para el desarrollo de este proyecto y se mencionan a continuación.

#### 2.1.1 Observer

Este patrón no es utilizado por Louvre, pero se incluye para resaltar la principal diferencia que posee respecto a las librerías alternativas mencionadas previamente.

El patrón *Observer* consiste en un objeto denominado “sujeto” que almacena una lista de objetos denominados “observadores” a los cuales notifica cuando sufre un cambio de estado emitiendo una señal que invoca generalmente uno de los métodos de los observadores.

La utilidad de este patrón yace en que **varios** objetos observadores pueden suscribirse **al mismo** objeto sujeto y ser notificados de forma simultánea cuando este emite una señal.

QtWayland y wl\_roots utilizan este patrón para permitir al desarrollador escuchar las señales generadas por las solicitudes de los clientes e implementar las reglas descritas en los protocolos. Esto implica que manualmente se deba suscribir a dichas señales y que por lo tanto no se pueda ofrecer una implementación por defecto de dichas reglas. Para que un compositor Wayland pueda funcionar es necesario que muchas de estas reglas principalmente las del protocolo de Wayland se encuentren implementadas, siendo por lo tanto la razón de por qué estas librerías fuerzan a conocer con anterioridad los protocolos y escribir cientos de líneas de código antes de poder ver resultados en pantalla.

En el Anexo A y Anexo B se incluye código de ejemplo que ilustra el uso de este patrón en QtWayland y wl\_roots respectivamente para ser notificado cuando un cliente crea un recurso xdg\_surface mediante la interfaz xdg\_shell.

#### 2.1.2 Factory Methods

Antes de continuar con la explicación de este patrón, se debe tener claridad sobre que es un método virtual. Un método virtual, es un método perteneciente a una clase cuya implementación puede ser sobrescrita al heredársela. Si en la clase padre el método se define, pero no se implementa, se denomina método puramente virtual y la clase padre pasa a ser abstracta, o sea, que no puede ser instanciada sin ser heredada y sin que se implemente dicho

método. Por el contrario, si en la clase padre el método se define e implementa, se denomina método virtual y la clase deja de ser abstracta, siendo posible sobrescribir la implementación del método al heredarla.

La ventaja de los métodos virtuales respecto a métodos convencionales, es que independiente de si una subclase es interpretada como su padre, al invocar uno de sus métodos virtuales siempre se ejecutará la implementación de la clase hija (si es que esta lo ha sobrescrito), mientras que con métodos convencionales se ejecutaría la implementación de la clase padre.

El patrón de diseño *Factory Methods*, también conocido como Constructor Virtual, permite crear instancias de objetos sin necesariamente definir su tipo específico, algo que no es posible actualmente con los constructores de clases en C++.

Siguiendo el ejemplo de la Figura 2, si se tiene la clase abstracta *Shape* y se quiere instanciar dinámicamente sus subclases *Circle*, *Square* y *Rectangle*, se debe crear otra clase que cumpla el rol de fábrica (*ShapeFactory*) que posea un constructor virtual, en este caso *getShape()*. La idea es que, dentro de dicho método, el usuario pueda elegir que subclase de *Shape* instanciar dinámicamente.

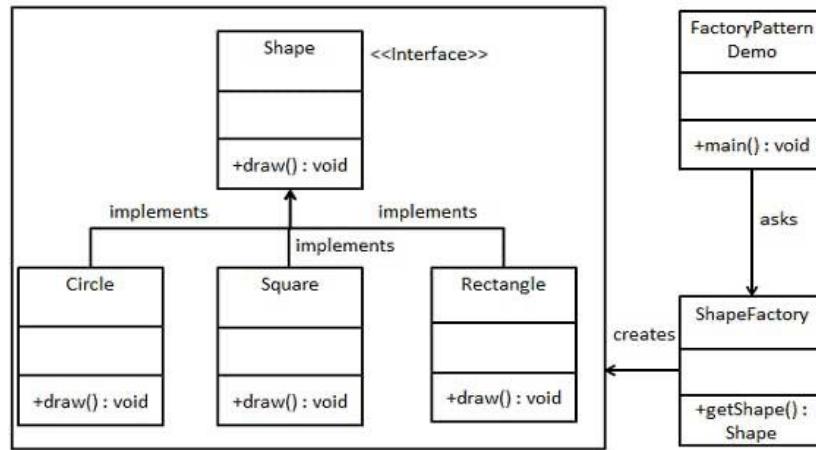


Figura 2: Ejemplo de constructor virtual.<sup>4</sup>

La librería desarrollada utiliza este patrón, posee una clase principal llamada LCompositor que cumple el rol de fábrica y contiene múltiples constructores y destructores virtuales para otras clases presentes en la librería, principalmente las que representan interfaces de protocolos creados por clientes Wayland, como las clases LSurface o LToplevelRole que representan las interfaces wl\_surface y xdg\_toplevel respectivamente.

Cada una de estas clases posee **métodos virtuales** dentro de los cuales se manejan algunas solicitudes de dichas interfaces. Estos métodos, son **invocados internamente** por la librería cuando recibe una solicitud de un cliente y no deben ser invocados por el desarrollador. Al ser métodos virtuales, Louvre ofrece una **forma predeterminada** para manejar cada

---

<sup>4</sup> Imagen disponible en <https://gist.github.com;brayancruces>. Consultado el 16 de octubre de 2022.

solicitud que cumple con las reglas de los protocolos y que el desarrollador puede sobrescribir si desea. Esto evita que deba aprender todas reglas de los protocolos de antemano e implementar todas las solicitudes manualmente antes de visualizar en pantalla un compositor funcional.

Para cambiar la forma predeterminada en que se maneja una solicitud, el desarrollador debe crear una subclase de la clase en la que se maneja la solicitud, sobrescribir el método virtual que representa la solicitud e instanciar y retornar la subclase dentro de su respectivo constructor virtual en LCompositor.

Por ejemplo, si el desarrollador quiere cambiar la forma por defecto en la que la librería maneja el cambio de dimensiones del buffer de una superficie, debe crear una subclase de LSurface (por ejemplo MySurface), sobrescribir su método virtual bufferSizeChanged() e instanciar MySurface en el constructor virtual createSurfaceRequest() de LCompositor.

El constructor virtual createSurfaceRequest() es invocado por la librería cuando un cliente crea una superficie, y retorna una nueva instancia de LSurface por defecto, lo que significa que el compositor invoca el método bufferSizeChanged() de LSurface cuando las dimensiones del buffer de la superficie cambian. Al retornar en cambio una nueva instancia de MySurface, el compositor invoca el método bufferSizeChanged() de MySurface (definido por el desarrollador).

### **2.1.3 Pimpl Idiom**

La idea de este patrón de diseño es mejorar la modularidad de la API. Reddy lo categoriza como un patrón de diseño, pero indica que en la práctica es estrictamente una técnica. Consiste en almacenar todos los miembros privados de una clase en un struct (u otra clase), cuya definición se oculta de la cabecera, brindando exclusivamente un puntero a este como un miembro privado. De esta forma toda implementación interna (privada) de la API queda oculta al usuario, lo cual es beneficioso, ya que cualquier cambio futuro en la implementación interna de la librería no forzará al usuario a actualizar sus cabeceras y por lo tanto el código que haya implementado.

La mayoría de las clases en Louvre siguen esta práctica. Por ejemplo, la clase LSurface posee todos sus miembros públicos en la cabecera LSurface.h y los privados en LSurfacePrivate.h. Para acceder a la API privada de LSurface, se define un atributo público imp() que retorna un puntero opaco a su instancia de LSurfacePrivate. Este atributo es utilizado internamente por la librería y no es requerido por los usuarios. Si un usuario desea acceder a la API privada de LSurface debe incluir la cabecera LSurfacePrivate.h, en cuyo caso corre un mayor riesgo de que su compositor no sea compatible con nuevas versiones de la librería.

### **2.1.4 Singleton**

El objetivo de este patrón es permitir crear clases que no puedan ser instanciadas múltiples veces, por ejemplo, una clase “Time” con métodos que retorne la hora actual, no necesita ser instanciada múltiples veces, una única instancia puede ser utilizada globalmente para

tener acceso a la hora actual. Siguiendo el ejemplo de la Figura 3, para evitar que la clase sea instanciada múltiples veces, se debe definir un constructor privado, crear una instancia estática de sí misma como miembro privado y un método público que retorne una referencia de esa instancia. Y para evitar que el usuario haga una copia de la instancia, se debe sobrescribir su constructor de copia.

```
class Time
{
public:
    static Time *getTime()
    {
        // Crea la instancia única si no existe
        if(!m_time)
            m_time = new Time();

        return m_time;
    }

    // Eliminamos constructor de copia
    Time(const Time&) = delete;

private:
    // Constructor privado
    Time(){};

    // Instancia única privada
    static Time *m_time = nullptr;
}
```

Figura 3: Ejemplo del patrón Singleton.

## 2.2 Prácticas para el diseño de APIs

Reddy (2011), plantea también una serie de buenas prácticas al momento de diseñar una API que previenen ciertas problemáticas y mejoran su usabilidad y rendimiento, varias de las cuales fueron aplicadas durante el desarrollo de Louvre y se describen a continuación.

### 2.2.1 Prefijos

Para evitar conflictos con clases o métodos de otras librerías, Reddy recomienda utilizar un prefijo en los nombres de los métodos o clases de la API. Por ejemplo, la librería OpenGL, que utiliza el prefijo "gl" en todos sus métodos.

Otra opción es encapsular los miembros de la API en un namespace. Por ejemplo, la librería utiliza el namespace Louvre y además agrega el prefijo "L" a cada una de sus clases. Esto permite a los desarrolladores referirse a sus elementos de manera clara y evitar posibles conflictos con nombres de otros métodos o clases.

### 2.2.2 Templates

Utilizar templates en C++ permite automatizar la creación de estructuras y brindar mayor flexibilidad al usuario. Por ejemplo, la librería STL utiliza templates en su clase *std::list*, permitiendo al usuario crear listas de cualquier tipo de datos.

Generalmente dentro de la librería se utilizan vectores de dos y cuatro dimensiones para representar puntos y rectas, para lo cual ofrece las clases LPoint, LSize y LRect que utilizan números enteros de 32 bits y sus variantes LPointF, LSizeF y LRectF que utilizan números flotantes de 32 bits. Sin embargo, es posible mediante templates generar variantes que utilicen otros tipos numéricos.

### 2.2.3 Operadores

Utilizar operadores permite en muchos casos reducir la verbosidad y mejorar la legibilidad del código. Por ejemplo, si se tiene una clase “Vector2” que represa un vector numérico de dos dimensiones, resulta más intuitivo para usuarios realizar operaciones entre estos usando operadores (valga la redundancia) que utilizando métodos tal como se muestra en la Figura 4.

#### 1. Utilizando operadores

```
vecD = vecC + (vecA + vecB)/2
```

#### 2. Utilizando métodos

```
vecD = vecC.add(vecA.add(vecB).divideBy(2))
```

Figura 4: Ejemplo de operadores.

Las clases LPoint, LSize, LRect y LRegion y sus variantes implementan varios operadores como suma, resta, multiplicación, división, y otros.

### 2.2.4 Llave const

Utilizar correctamente la llave “const”, permite definir reglas al compilador que impidan al usuario manipular indebidamente miembros de una estructura de datos evitando así que

utilice incorrectamente la API. También al utilizarla en argumentos de métodos, brinda seguridad al usuario de que estos no serán modificados al interior del método.

Por ejemplo, la clase LCompositor proporciona acceso a una lista de clientes conectados al compositor a través del método `LCompositor::clients()`. Este método utiliza la llave "const" para prevenir cualquier modificación de la lista por parte del desarrollador, lo que podría generar errores en la librería.

### 2.2.5 Argumentos por referencia o punteros

Definir argumentos de métodos como punteros o referencia evita que se realicen copias innecesarias de objetos o structs, lo que impacta positivamente en el rendimiento.

La mayoría de los argumentos de métodos en la librería siguen esta práctica exceptuando los argumentos que representan datos primitivos.

Por ejemplo la clase LSurface posee el método `LSurface::setPosC()`, para asignar la posición de una superficie, el cual toma como argumento una instancia de LPoint (representación un vector numérico de dos dimensiones) por referencia, para evitar realizar una copia de innecesaria de esta.

### 2.2.6 Argumentos por defecto

Definir argumentos por defecto evita que se deba completar todos los campos de un método, lo resulta particularmente útil cuando poseen muchos argumentos o deben ser invocados un gran número de veces en el código. También pueden servir al usuario como pista para comprender que tipo valor debe ser ingresado.

Por ejemplo, la clase LTexture cuenta con el método `LTexture::setData()` para asignar contenido a una textura OpenGL. El desarrollador solo necesita proporcionar los dos primeros argumentos, que representan el ancho y el alto del buffer, y el tercero que es un puntero al buffer con el contenido a asignar. Los argumentos restantes indican el formato y tipo de dato de los pixeles y el tipo de buffer, cuyos valores por defecto suelen ser los más comunes al cargar texturas, `GL_BGRA`, `GL_UNSIGNED_BYTE` y `SHM` respectivamente.

### 2.2.7 Bitfields y enums

Utilizar bitfields y enums es una forma eficiente y compacta de representar un número limitado de estados que pueden estar activos simultáneamente dentro de un mismo contexto utilizando típicamente una variable de tipo entero y operadores lógicos binarios.

Por ejemplo, tomando como referencia el enum de la Figura 5 que define los posibles estados de una persona, alguien que esté acostado y comiendo estaría representado por la cadena de bits 101 (conformada por la unión 001 | 100), otra persona que esté acostada y estudiando estaría representada por 011 (conformada por la unión 001 | 010), y la intersección de ambos casos anteriores 001 (conformada por 101 & 011) sería el estado que

poseen en común, o sea comiendo. Notar que los valores del enum utilizan potencias de dos para que cada estado pueda ser representado por un bit distinto.

```
enum EstadoPersona
{
    Acostado      = 1,
    Estudiando    = 2,
    Comiendo       = 4
}
```

Figura 5: Ejemplo de bitfields y enums.

### 2.3 Características de buenos diseños de APIs

Henning (2009) enumera una lista de características que poseen los buenos diseños de APIs de las cuales varias fueron incorporadas durante el desarrollo de la librería y se mencionan a continuación:

**1. Una API debería brindar suficiente funcionalidad para que el usuario logre cumplir su tarea:**

Henning se refiere a este punto como “obvio”, sin embargo, sugiere realizar una lista de chequeo para revisar que todas las funcionalidades requeridas por el usuario hayan sido implementadas.

**2. Una API debe ser minimalista, sin imponer inconveniencia innecesaria al usuario:**

Mientras menos tipos, parámetros, funciones y clases contenga una API más sencillo será para los usuarios aprenderla, y que, por lo tanto, no se debería añadir mayor complejidad si el usuario no lo requiere.

**3. Una API no puede ser diseñada sin comprender su contexto:**

Este punto se ilustra con el ejemplo de una clase en la cual existe un método que retorna un cierto tipo de dato solo si este existe. Dependiendo del contexto, para el usuario podría ser útil que retorne una excepción, un valor por defecto, o nulo (para detectar si la variable existe). Por lo tanto, es necesario saber en qué contexto el usuario lo utilizará para retornar un valor que maximice su utilidad.

#### **4. Las APIs multipropósito deberían ser libres de políticas y las de propósito específico ricas en políticas:**

Es inevitable que las APIs impongan ciertas políticas a los usuarios como semánticas, estilo de programación, etc. Cuando el diseñador no esté seguro del contexto en el que se utilizará la API, debería limitar las políticas y brindar la mayor flexibilidad posible al usuario. Solo si el diseñador conoce a la perfección el contexto sobre el cual se utilizará, debería incorporar más políticas y eliminar flexibilidad innecesaria en la API.

#### **5. Las APIs deberían ser diseñadas desde la perspectiva del usuario:**

Generalmente los diseñadores tienden a enfocarse en la funcionalidad y olvidan la experiencia de los usuarios. Ilustra este punto usando como ejemplo el siguiente método: **makeTV(false, true)**. Es evidente que el método debe crear un televisor, pero no es claro qué representan sus parámetros. Si el diseñador definiera un enum con los posibles valores de los parámetros, el método quedaría: **makeTV(Color, FlatScreen)**, lo que facilita la comprensión al usuario aún cuando sigue cumpliendo la misma funcionalidad.

#### **6. Las buenas APIs no deben dejar el “trabajo sucio” al usuario:**

Muchos diseñadores no son lo suficientemente firmes al imponer políticas al no tener bien definido el objetivo de sus APIs. Por lo tanto, suelen ofrecer flexibilidad innecesaria, lo cual resulta en funcionalidades poco utilizadas, aumentando así su complejidad. También menciona el ejemplo de COBRA C++, una librería caracterizada por su alta eficiencia, pero que deja en manos de los usuarios el seguimiento, alocación y de-alocación de memoria.

#### **7. Las APIs deberían ser documentadas antes de ser implementadas:**

Es un error documentar una API luego de ser implementada ya que para entonces el desarrollador se encuentra demasiado familiarizado con esta y por lo tanto tiende a omitir información relevante, dando por hecho que comparte el mismo conocimiento previo con el usuario.

### **2.4 Evaluación de usabilidad de APIs**

Para evaluar la usabilidad de la librería se utilizó un formulario basado en el framework de dimensiones cognitivas (Zghidi, Hammouda, & Hnich, 2018) y adaptado por Steve Clarke para medir la usabilidad de APIs C++ (Clarke, 2006) el cual se puede ver en el Anexo C. Este formulario incluye preguntas para evaluar el nivel de abstracción, estilo de aprendizaje, trabajo por unidad de paso, evaluación progresiva, compromiso prematuro, penetrabilidad, elaboración, viscosidad, consistencia, expresividad y correspondencia de dominio de la API.

### 3. ARQUITECTURA PROPUESTA

En este capítulo se brinda una descripción general de la arquitectura propuesta para la librería Louvre, incluyendo los patrones de diseño utilizados, los diferentes componentes y cómo estos interactúan entre sí. También se explica cómo esta arquitectura facilita el trabajo del desarrollador y reduce la curva de aprendizaje necesaria para crear compositores Wayland.

En los capítulos 4 y 5, se describen con mayor detalle las funcionalidades y propiedades de cada una de las clases, así como los aspectos relacionados con el rendimiento.

#### 3.1 Componentes

Una forma sencilla de ilustrar la arquitectura de la librería es analizando sus componentes principales y como estos interactúan entre sí (ver Figura 6).

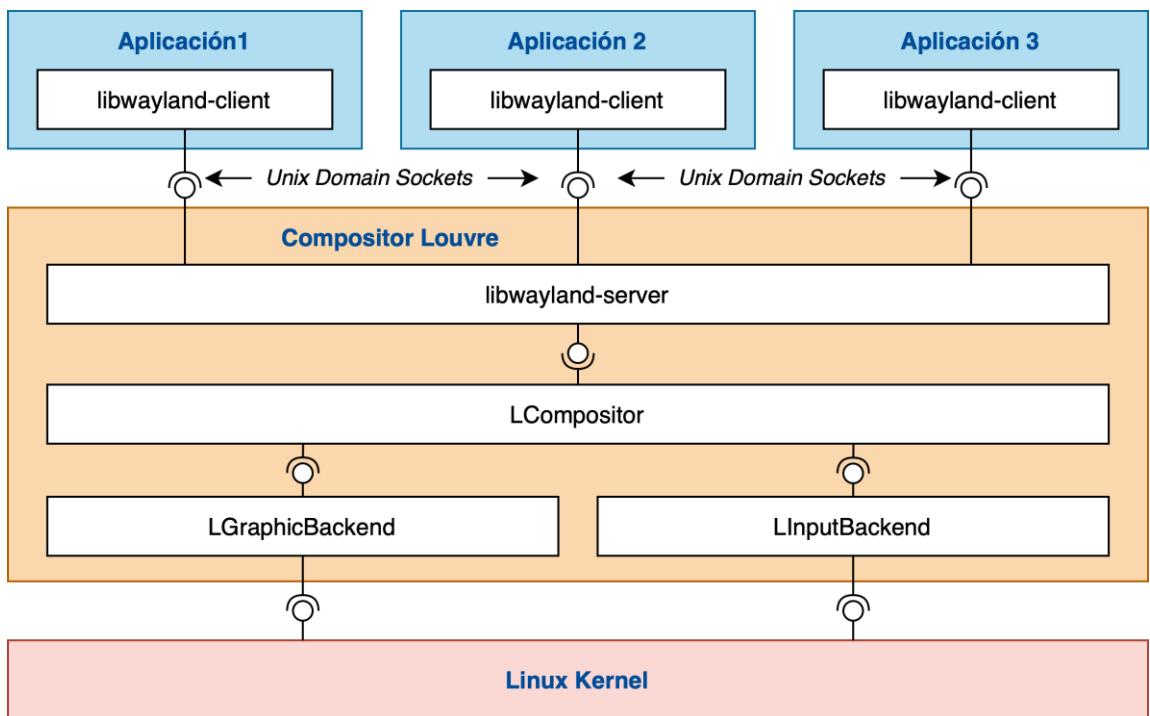


Figura 6: Diagrama de componentes de Louvre.

Las aplicaciones se conectan al compositor a través de la librería de Wayland en su versión para clientes (`libwayland-client`), mientras que el compositor recibe las solicitudes a través de la versión para servidores (`libwayland-server`). La clase `LCompositor` se encarga de procesar y simplificar las solicitudes de los clientes y llamar a métodos virtuales de otras clases de la librería para que el desarrollador pueda manejar las solicitudes de acuerdo a su propia lógica.

La clase LCompositor también interactúa con los backends de input y gráfico, los cuales brindan acceso a los eventos de entrada del sistema y le permiten renderizar en pantalla respectivamente. Estos backends ofrecen una interfaz estándar para el compositor definida en las clases LInputBackend y LGraphicBackend, lo que permite intercambiarlos sin necesidad de modificar el código del compositor.

### 3.1.1 Manejo de solicitudes de clientes

Como muestra la Figura 7, el compositor gestiona las solicitudes de los clientes en dos capas. La primera capa se encarga de recibir y preprocesar las solicitudes utilizando las interfaces C de los protocolos a través de la librería de Wayland. Muchas de las solicitudes, especialmente aquellas que tienen solo una forma de ser implementadas se manejan completamente en esta capa. Las otras solicitudes, por lo general, se preprocesan, simplifican y pasan a la segunda capa mediante la invocación de métodos virtuales de las distintas clases de Louvre.

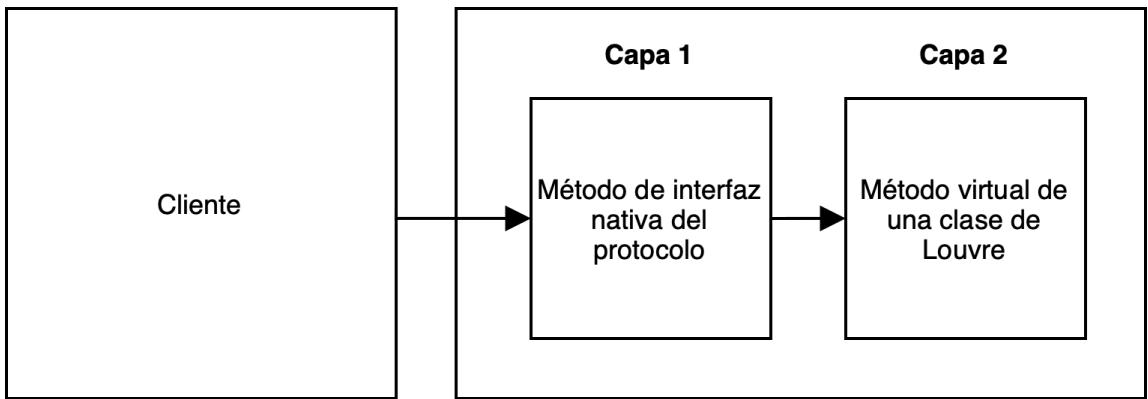


Figura 7: Capas de gestión de solicitudes de clientes en Louvre.

Al ser métodos virtuales, la librería puede proporcionar una forma predeterminada de manejar las solicitudes en la segunda capa y, a su vez, permitir al desarrollador sobrescribirlas si lo desea. El desarrollador solo tiene acceso y puede modificar las solicitudes preprocesadas y simplificadas que recibe la segunda capa.

### 3.1.2 Protocolos

La librería implementa los protocolos Wayland y XDG Shell, los cuales permiten que aplicaciones desarrolladas en toolkits como GTK, Qt, Chromium, SDL y EFL puedan funcionar con el compositor (DeVault, 2022). Además, se ha implementado el protocolo XDG Decoration (Ser, XDG decoration, 2018), que permite al compositor y a los clientes negociar quién renderiza las decoraciones de las ventanas (barra de título, sombras, etc) y el protocolo Pointer Gestures, que permite enviar eventos de touchpads más sofisticados

como pinch zoom o scroll con múltiples dedos (Wayland Explorer, 2022). En las Tablas 2, 3, 4 y 5 se listan todas las interfaces y solicitudes de los protocolos Wayland, XDG Shell, XDG Decoration y Pointer Gestures implementadas por la librería, y se indica cuáles gatillan métodos virtuales en la segunda capa.

Las solicitudes que requieren una respuesta (evento) de la segunda capa para cumplir con las reglas de los protocolos o para el correcto funcionamiento de la librería (por ejemplo, cuando involucran la creación o destrucción de recursos) se resaltan en color rojo. Las solicitudes que no requieren ser implementadas en la segunda capa pero que pueden ser útiles para el desarrollador, se resaltan en naranja. Y las solicitudes con valor "No" en la columna "Implementación en la segunda capa" son completamente manejadas por la librería en la primera capa.

Tabla 2: Solicitudes del protocolo Wayland.

Interfaz	Solicitud	Implementación en la segunda capa
wl_display	sync	No
	get_registry	No
wl_registry	bind	No
wl_compositor	create_surface	<b>Sí (obligatorio)</b>
	create_region	No
wl_shm_pool	create_buffer	No
	destroy	No
	resize	No
wl_shm	create_pool	No
wl_buffer	destroy	No
wl_data_offer	accept	No
	receive	No
	destroy	No
	finish	No
	set_actions	No
wl_data_source	offer	No
	destroy	No
	set_actions	No
	start_drag	<b>Sí (obligatorio)</b>
	set_selection	<b>Sí (obligatorio)</b>
	release	No
wl_data_device_manager	create_data_source	No
	get_data_device	No
wl_surface	destroy	<b>Sí (obligatorio)</b>
	attach	<b>Sí (opcional)</b>
	damage	<b>Sí (opcional)</b>
	frame	No
	set_opaque_region	<b>Sí (opcional)</b>
	set_input_region	<b>Sí (opcional)</b>
	commit	<b>Sí (opcional)</b>
	set_buffer_transform	<b>Sí (opcional)</b>
	set_buffer_scale	<b>Sí (opcional)</b>
	damage_buffer	<b>Sí (opcional)</b>

Tabla 2: Solicitudes del protocolo Wayland (continuación).

	offset	Sí (obligatorio)
wl_seat	get_pointer	No
	get_keyboard	No
	get_touch	No
wl_pointer	set_cursor	Sí (opcional)
	release	No
wl_keyboard	release	No
wl_output	release	No
wl_region	destroy	No
	release	No
	subtract	No
wl_subcompositor	destroy	No
	get_subsurface	Sí (obligatorio)
wl_subsurface	destroy	Sí (obligatorio)
	set_position	Sí (opcional)
	place_above	Sí (opcional)
	place_below	Sí (opcional)
	set_sync	Sí (opcional)
	set_desync	Sí (opcional)

Tabla 3: Solicitudes del protocolo XDG Shell.

Interfaz	Solicitud	Implementación en la segunda capa
xdg_wm_base	destroy	No
	create_positioner	No
	get_xdg_surface	No
	pong	Sí (opcional)
xdg_positioner	set_size	No
	set_anchor_rect	No
	set_anchor	No
	set_gravity	No
	set_constraint_adjustment	No
	set_offset	No
	set_reactive	No
	set_parent_size	No
	set_parent_configure	No
xdg_surface	destroy	Sí (obligatorio)
	get_toplevel	Sí (obligatorio)
	get_popup	Sí (obligatorio)
	set_window_geometry	Sí (opcional)
	ack_configure	No
xdg_toplevel	destroy	Sí (obligatorio)
	set_parent	Sí (opcional)
	set_title	Sí (opcional)
	set_app_id	Sí (opcional)
	show_window_menu	Sí (obligatorio)
	move	Sí (obligatorio)
	resize	Sí (obligatorio)

Tabla 3: Solicitudes del protocolo XDG Shell (continuación).

	set_max_size	Sí (opcional)
	set_min_size	Sí (opcional)
	set_maximized	Sí (obligatorio)
	unset_maximized	Sí (obligatorio)
	setFullscreen	Sí (obligatorio)
	unsetFullscreen	Sí (obligatorio)
	setMinimized	Sí (obligatorio)
xdg_popup	destroy	Sí (obligatorio)
	grab	Sí (obligatorio)
	reposition	Sí (obligatorio)

Tabla 4: Solicitudes del protocolo XDG Decoration.

Interfaz	Solicitud	Implementación en la segunda capa
xdg_decoration_manager	destroy	No
	get_toplevel_decoration	No
xdg_toplevel_decoration	destroy	Sí (opcional)
	set_mode	Sí (opcional)
	unset_mode	Sí (opcional)

Tabla 5: Solicitudes del protocolo Pointer Gestures.

Interfaz	Solicitud	Implementación en la segunda capa
wp_pointer_gestures	get_swipe_gesture	No
	get_pinch_gesture	No
	release	No
	get_hold_gesture	No
	wp_pointer_gesture_swipe	get_toplevel_decoration
	wp_pointer_gesture_pinch	destroy
	wp_pointer_gesture_hold	set_mode

Al calcular la razón entre el número total de solicitudes y las que deben ser implementadas en la segunda capa, se obtiene un 21%. Esto podría interpretarse como que existe un 79% menos de posibilidades de cometer errores de protocolos al utilizar la librería.

### 3.1.3 Backends gráficos

Los backends gráficos en Louvre son librerías dinámicas que permiten desplegar el compositor en diversos contextos. Por ejemplo, sobre un compositor X11, sobre otro compositor Wayland anidado o directamente en pantalla con las APIs proporcionadas en Linux. La librería ofrece dos backends gráficos que se instalan en **/usr/etc/Louvre/backends** los cuales pueden ser cargados dinámicamente durante la ejecución del compositor.

- **LGraphicBackendX11**: este backend permite mostrar el compositor en una ventana fullscreen sobre un compositor X11.

- **LGraphicBackendDRM**: este backend permite mostrar el compositor directamente en pantalla utilizando las APIs DRM, KMS y GBM de Linux.

Por defecto la librería intenta cargar el backend DRM a menos que se especifique otro como se muestra en la Figura 8.

```
#include <LCompositor.h>

using namespace Louvre;

int main()
{
    LCompositor compositor;
    compositor.loadGraphicBackend("/usr/etc/Louvre/backends/libLGraphicBackendX11.so");
    compositor.loadInputBackend("/usr/etc/Louvre/backends/libLInputBackendX11.so");
    return compositor.start();
}
```

Figura 8: Ejemplo de carga de backends.

Los backends gráficos implementan una interfaz estándar definida en la cabecera de la clase **LGraphicBackend**, que exportan a través de un struct con la misma estructura para que el compositor pueda invocar sus métodos tras importarla como una librería dinámica. Esto permite al desarrollador utilizar distintos backends sin tener que modificar el código de sus compositores.

La interfaz descrita en **LGraphicBackend** incluye los siguientes métodos:

1. **initialize()**: este método es llamado por el compositor para inicializar el backend y prepararlo para manejar los demás métodos de la interfaz. El backend debe inicializar las APIs del contexto en el que funciona y determinar qué salidas están disponibles.
2. **uninitialize()**: solicita al backend liberar los recursos en memoria que haya creado durante la inicialización y ejecución.
3. **getAvailableOutputs()**: debe devolver una lista de salidas (**LOutput**) que pueden ser inicializadas.
4. **initializeOutput()**: solicita al backend crear un contexto EGL y OpenGL para una salida, con el fin de poder utilizarla posteriormente para renderizar.
5. **uninitializeOutput()**: solicita al backend eliminar los contextos EGL y OpenGL de una salida.
6. **flipOutputPage()**: solicita al backend cambiar el framebuffer que se muestra en pantalla por el que acaba de ser renderizado.
7. **getOutputEGLDisplay()**: devuelve un puntero al struct **EGLDisplay** de una salida.

8. **getOutputPhysicalSize()**: debe devolver un vector LSize con las dimensiones físicas de una salida en milímetros.
9. **getOutputCurrentBufferIndex()**: devuelve el índice del framebuffer en el que se está renderizando en una salida. Por lo general, los backends utilizan dos framebuffers: mientras renderizan en uno, muestran el otro en pantalla y luego los intercambian en cada frame.
10. **getOutputName()**: retorna un string que identifica el tipo de una salida, por ejemplo "HDMI-1", "eDP-1", "X11 Window", etc.
11. **getOutputManufacturerName()**: retorna el nombre del fabricante de una salida.
12. **getOutputmodelName()**: retorna el nombre del modelo de una salida.
13. **getOutputDescription()**: retorna la descripción de una salida.
14. **getOutputModes()**: retorna una lista de modos disponibles para una salida. Los modos representan la resolución y tasa de refresco posibles de un monitor.
15. **setOutputMode()**: asigna el modo a una salida.
16. **getOutputPreferredMode()**: retorna el modo preferido de una salida. Por lo general el modo preferido es el que posee mayor resolución y tasa de refresco.
17. **getOutputCurrentMode()**: retorna el modo actual de una salida.
18. **getOutputModeSize()**: retorna un vector LSize con las dimensiones de un modo.
19. **getOutputModeRefreshRate()**: retorna la tasa de refresco de un modo en MHz.
20. **getOutputModeIsPreferred()**: retorna un booleano indicando si un modo es el preferido por su salida.
21. **initializeCursor()**: solicita al backend inicializar el cursor.
22. **hasHardwareCursorSupport()**: retorna un booleano indicando si el backend soporta composición de cursores vía hardware.
23. **setCursorPosition()**: asigna la posición del cursor. El compositor debe incluir la transformación dada por el hotspot del cursor en la posición.
24. **setCursorPosition()**: asigna la posición del cursor. El compositor debe incluir la transformación dada por el hotspot del cursor en la posición.

El objetivo de los backends gráficos, a parte de mejorar la modularidad de la librería y portabilidad de los compositores desarrollados, es evitar que el desarrollador deba interactuar directamente con APIs de bajo nivel del Kernel de Linux, las cuales suelen ser complejas y estar brevemente documentadas, requiriendo mucho tiempo y pruebas para poder implementarlas correctamente.

### 3.1.4 Backends de input

Al igual que los backends gráficos, los backends de input son librerías dinámicas que permiten al compositor escuchar eventos de entrada del teclado, ratón, touchpad, etc. La librería ofrece dos backends de input:

- **LInputBackendX11**: este backend permite acceder a los eventos de entrada generados por un compositor X11. Debe ser utilizado en conjunto con el backend gráfico X11.
- **LInputBackendLibinput**: este backend permite acceder a los eventos generados por la API EVDEV de Linux mediante la librería Libinput. Se recomienda utilizarlo en conjunto con el backend gráfico DRM.

Por defecto la librería intenta cargar el backend Libinput a menos que se especifique otro como en la Figura 8.

La interfaz utilizada por los backends de input se encuentra en la cabecera de la clase LInputBackend, la cual posee los siguientes métodos:

1. **initialize()**: este método es invocado por el compositor para inicializar el backend y prepararlo para manejar los demás métodos de la interfaz. El backend debe inicializar las APIs del contexto en el que funciona y determinar qué dispositivos de entrada están disponibles.
2. **uninitialize()**: Sigue al backend liberar los recursos en memoria que haya creado.
3. **getCapabilities()**: debe retornar un bitfield con las capacidades de entrada del backend. Las posibles capacidades son **POINTER**, **KEYBOARD** y **TOUCH**, que aluden a eventos de puntero (ratón, touchpad, etc.), eventos del teclado y eventos de pantallas táctiles, respectivamente.
4. **getContextHandle()**: debe retornar un puntero opaco a un struct o clase del contexto utilizado por el backend. Por ejemplo, en el caso del backend Libinput, retorna un puntero a un struct libinput de la librería Libinput. Y en el caso del backend X11, retorna un puntero a un struct Display de la librería Xlib. El objetivo de este método es permitir al desarrollador acceder a la API nativa del backend.
5. **suspend()**: este método solicita al backend suspender la emisión de eventos.
6. **resume()**: este método solicita al backend resumir la emisión de eventos.
7. **forceUpdate()**: este método obliga al backend a procesar y emitir los eventos que tenga en la cola.

Es importante señalar que tanto esta interfaz como la del backend gráfico son parte de la API privada de la librería y, por lo tanto, los desarrolladores no necesitan interactuar directamente con ellas al crear sus compositores. Los desarrolladores interactúan de manera indirecta con el backend gráfico mediante funciones de mayor nivel presentes en las diversas clases de la librería, como las funciones de renderizado en la clase LPainter o LOutput y el uso de composición de cursores mediante hardware en la clase LCursor. De la misma forma, pueden acceder a los eventos de entrada del backend de input mediante los métodos virtuales de las clases LPointer, LKeyboard y LSeat.

La única situación en la que deberían utilizar estas interfaces es si desean desarrollar backends distintos a los ofrecidos por la librería.

## 3.2 Fábrica de recursos

Como se mencionó anteriormente, para cambiar el comportamiento predeterminado del compositor es necesario sobreescibir los métodos virtuales de las clases que ofrece la librería. Cada clase se encarga de un conjunto específico de funcionalidades, como manejar los eventos de entrada, procesar solicitudes específicas de los clientes, renderizar, etc.

La clase LCompositor se considera la "fábrica de recursos" de la librería, ya que es la encargada de generar y eliminar instancias de dichas clases. Para ello, tiene una lista de constructores y destructores virtuales, que pueden ser sobreescritos para utilizar variantes de dichas clases definidas por el propio desarrollador.

Por ejemplo (siguiendo el diagrama de secuencia de la Figura 9), cuando un cliente desea crear una superficie, envía la solicitud al compositor a través de la interfaz wl\_compositor del protocolo de Wayland. El compositor procesa la solicitud y llama al constructor virtual LCompositor::createSurfaceRequest(), que debe devolver una nueva instancia de la clase LSurface. La implementación predeterminada de LCompositor hace exactamente eso, pero el desarrollador puede sobreescibir LCompositor y dicho constructor virtual para devolver una nueva instancia de su propia subclase de LSurface. De esta manera, puede cambiar la lógica predeterminada que utiliza LSurface para manejar las solicitudes del cliente sobreescribiendo sus métodos virtuales.

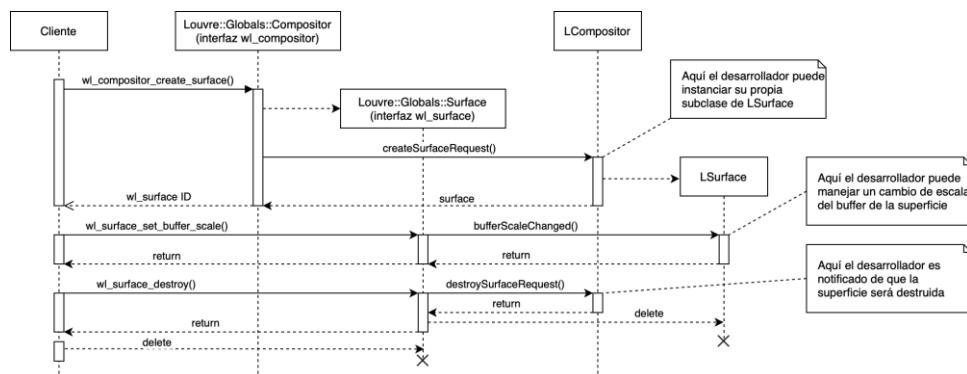


Figura 9: Secuencia de creación, uso y destrucción de una superficie.

La secuencia detallada en la Figura 8 es similar a las demás solicitudes de los protocolos que implican el uso de métodos virtuales en la segunda capa en otras clases de la librería.

## 3.3 Inicialización del compositor

Para iniciar el compositor se debe llamar al método LCompositor::start() como se muestra en la Figura 10. Esto crea una instancia de la clase LSeat mediante el constructor virtual createSeatRequest() la cual es utilizada por los backends para obtener un descriptor de archivo con permisos de lectura y escritura de los dispositivos de entrada y salida de la sesión (sin necesidad de ser superusuario) mediante su método LSeat::openDevice() el cual internamente utiliza la librería libseat (Levinsen, s.f.). Además esta clase, es utilizada

como interfaz para que el desarrollador pueda acceder a los eventos de entrada nativos del backend.

En el constructor de LSeat, se crea una instancia de las clases LPointer y LKeyboard mediante los constructores createPointerRequest() y createKeyboardRequest(), las cuales también sirven como interfaz para que el usuario pueda acceder a los eventos de entrada pero específicamente del puntero y teclado respectivamente.

Después se carga e inicializa el backend gráfico, el cual como ya se mencionó, utiliza el método openDevice() de LSeat para acceder a las GPUs del sistema y detectar las salidas que se encuentren disponibles. Por cada salida disponible, el backend gráfico solicita al compositor crear una instancia de LOutput mediante el constructor virtual createOutputRequest(). Esto permite posteriormente al usuario renderizar en dichas salidas luego de inicializarlas con el método LCompositor::addOutput().

Una vez inicializado el backend gráfico, se crea una instancia de la clase LOutputManager mediante el constructor virtual createOutputManagerRequest(), la cual permite al usuario

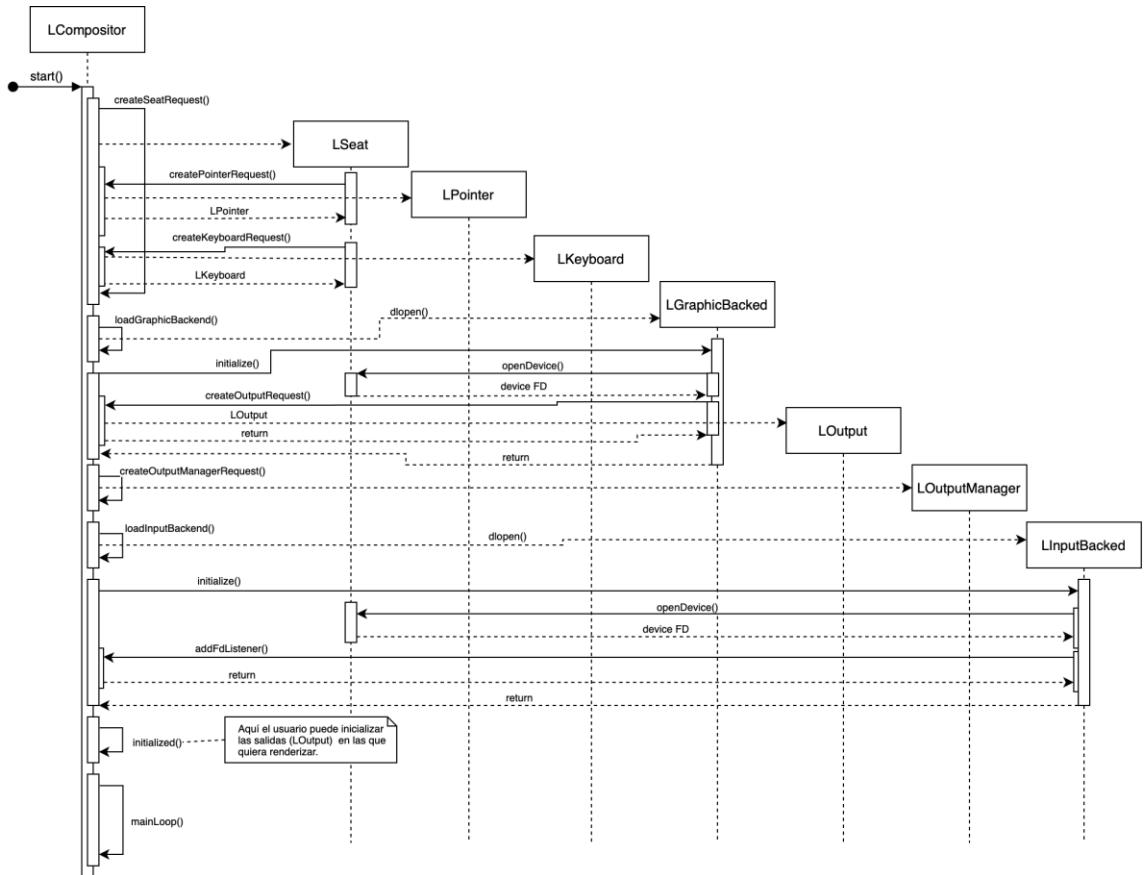


Figura 10: Secuencia de inicialización del compositor.

ser notificado cuando una nueva salida se encuentra disponible o deja de estarlo (por ejemplo, al conectar o desconectar un monitor externo mediante un puerto HDMI o VGA).

Luego se carga e inicializa el backend de input, el cual también utiliza el método `openDevice()` de `LSeat` para acceder a los eventos de entrada de los dispositivos del sistema.

Para poder detectar cuando hay nuevos eventos de entrada, el backend de input añade el descriptor de archivo de los dispositivos a el loop de eventos del compositor con el método `addFdListener()`, esto le permite procesar y despachar los eventos en la cola.

El compositor luego notifica al usuario que ha sido inicializado correctamente con el método `virtual initialized()`. Dentro de este método, se recomienda configurar e inicializar las salidas en las que desea renderizar con el método `LCompositor::addOutput()`, el cual crea un nuevo hilo de renderizado por cada salida.

Finalmente, se entra en el loop `mainLoop()`, el cual se encarga de procesar las solicitudes de clientes, eventos de input y enviar eventos a los clientes.

### 3.4 Dependencias e instalación

Para utilizar la librería, se puede elegir entre compilarla e instalarla manualmente o utilizar paquetes de instalación con binarios precompilados para distribuciones Linux basadas en Debian o Red Hat. El código fuente de la librería se encuentra disponible en un repositorio público en GitHub (<https://github.com/CuarzoSoftware/Louvre>) y los paquetes de instalación se encuentran en la sección *Releases* de dicho repositorio. Si se decide compilar la librería manualmente, se debe utilizar las herramientas Meson o qmake y asegurarse de tener instaladas las librerías enumeradas en la Tabla 6.

Tabla 6: Dependencias de Louvre.

Librería	Versión
Wayland Server	$\geq 1.16$
EGL	$\geq 1.5.0$
GLES 2.0	$\geq 13.0.6$
DRM	$\geq 2.4.85$
GBM	$\geq 22.2.0$
EVDEV	$\geq 1.5.6$
Libinput	$\geq 1.6.3$
XCursor	$\geq 1.1.15$
XKB Common	$\geq 0.7.1$
XFixes	$\geq 6.0.0$
XRandr	$\geq 1.5.1$
Pixman	$\geq 0.40.0$
Libseat	$\geq 0.6.4$

En el Anexo D se detalla una captura de pantalla del segundo capítulo del tutorial, el cual proporciona instrucciones específicas para compilar la librería utilizando Meson tanto en Debian como en Red Hat. En cuanto a los paquetes de instalación, estos se compilan y generan mediante workflows de GitHub Actions, cuya configuración se puede ver en el archivo **`./github/workflows/release.yml`** del repositorio.

Los paquetes de instalación se generan de esta manera para transparentar el proceso y permitir a los usuarios de la librería verificar y asegurarse de que no incluyen código malicioso.

## 4. CLASES

La librería ofrece un variado set de clases para manejar las diversas solicitudes de clientes, renderizar, acceder al input, etc. En esta sección se describe la funcionalidad, métodos y atributos principales de cada una.

La Figura 11 muestra un diagrama de paquetes que agrupa las clases según su funcionalidad.

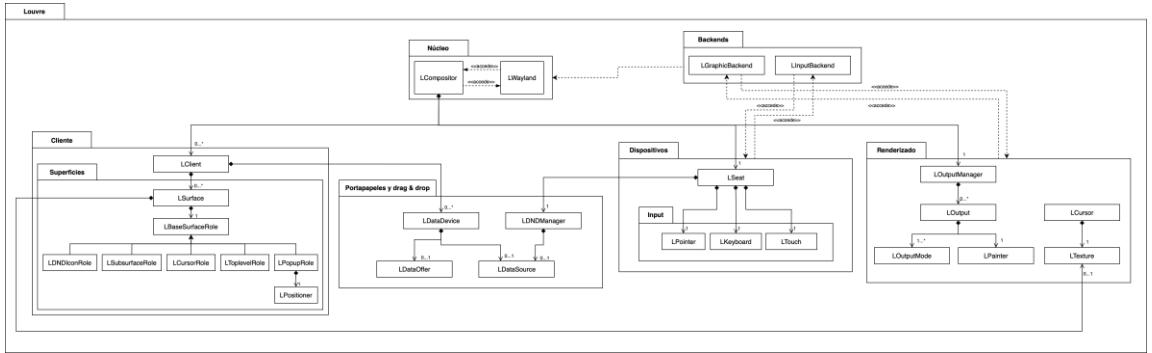


Figura 11: Diagrama de paquetes de Louvre.

### 4.1 Clase LComposer

`LComposer` es la clase principal de la librería y se encarga de iniciar el loop de eventos de Wayland e inicializar los backends de input y gráfico tras llamar al método `start()`. El método virtual `initialized()` es invocado por la librería después de llamar al método `start()` para notificar la correcta inicialización del compositor. Cualquier tipo de configuración inicial debe realizarse dentro de este método o posteriormente para evitar errores de segmentación. Además, `LComposer` funciona como fábrica de recursos siguiendo el patrón de diseño Factory. Tiene métodos virtuales que se utilizan como constructores y destructores virtuales de las clases que ofrece la librería. Esto permite al desarrollador sobrescribir los métodos virtuales de dichas clases y cambiar la implementación predeterminada que ofrece la librería.

Esta clase además como se puede ver en la Figura 12, posee métodos para poder acceder a la instancia global del cursor, al asiento del compositor, a su API privada, a la lista de clientes conectados al compositor, a la lista de salidas inicializadas, a una lista de superficies creadas por clientes ordenadas jerárquicamente, y métodos para reordenarlas.

### 4.2 Clase LOutput

La clase `LOutput` representa una salida en la que se puede mostrar visualmente parte del compositor. A menudo se asocia a un monitor o pantalla, como se muestra en la Figura

LCompositor
<pre>+ void addOutput(LOutput *output) + const list&lt; LClient * &gt; &amp; clients() const + LCursor * cursor() const + virtual void cursorInitialized() + void finish() + Int32 globalScale() const + virtual void globalScaleChanged(Int32 oldScale, Int32 newScale) + bool graphicBackendInitialized() const + LCompositorPrivate * imp() const + virtual void initialized() + bool inputBackendInitialized() const + LCompositor() + LCompositor(const LCompositor &amp;)=delete + bool loadGraphicBackend(const char *path) + bool loadInputBackend(const char *path) + std::thread::id mainThreadId() const + LCompositor &amp; operator=(const LCompositor &amp;)=delete + LOutputManager * outputManager() const + const list&lt; LOutput * &gt; &amp; outputs() const + void raiseSurface(LSurface *surface) + void removeOutput(LOutput *output) + void repaintAllOutputs() + LSeat * seat() const + int start() + const list&lt; LSurface * &gt; &amp; surfaces() const + virtual ~LCompositor() * virtual LOutputManager * createOutputManagerRequest(LOutputManager::Params *params) * virtual LOutput * createOutputRequest() * virtual LClient * createClientRequest(LClient::Params *params) * virtual LSurface * createSurfaceRequest(LSurface::Params *params) * virtual LSeat * createSeatRequest(LSeat::Params *params) * virtual LPointer * createPointerRequest(LPointer::Params *params) * virtual LKeyboard * createKeyboardRequest(LKeyboard::Params *params) * virtual LDNDManager * createDNDManagerRequest(LDNDManager::Params *params) * virtual LToplevelRole * createToplevelRoleRequest(LToplevelRole::Params *params) * virtual LPopupRole * createPopupRoleRequest(LPopupRole::Params *params) * virtual LSubsurfaceRole * createSubsurfaceRoleRequest(LSubsurfaceRole::Params *params) * virtual LCursorRole * createCursorRoleRequest(LCursorRole::Params *params) * virtual LDNDIconRole * createDNDIconRoleRequest(LDNDIconRole::Params *params) * virtual void destroyOutputRequest(LOutput *output) * virtual void destroyOutputManagerRequest() * virtual void destroyClientRequest(LClient *client) * virtual void destroySeatRequest() * virtual void destroyPointerRequest() * virtual void destroyKeyboardRequest() * virtual void destroyDNDManagerRequest() * virtual void destroySubsurfaceRoleRequest(LSubsurfaceRole *subsurface) * virtual void destroySurfaceRequest(LSurface *surface) * virtual void destroyToplevelRoleRequest(LToplevelRole *toplevel) * virtual void destroyPopupRoleRequest(LPopupRole *popup) * virtual void destroyCursorRoleRequest(LCursorRole *cursor) * virtual void destroyDNDIconRoleRequest(LDNDIconRole *icon)</pre>

Figura 12: Miembros de la clase LCompositor.

13, pero también puede ser una ventana en un entorno X11 o Wayland, dependiendo del backend gráfico seleccionado.



Figura 13: Layout de múltiples salidas.<sup>5</sup>

Posee métodos virtuales que permiten inicializar contextos gráficos, renderizar frames y ser notificado cuando cambian las dimensiones del framebuffer como se muestra en la Figura 14.

LOutput
<pre>+ LCompositor * compositor() const + Int32 currentBuffer() const + const LOutputMode * currentMode() const + LCursor * cursor() const + const char * description() const + Int32 dpi() + EGLDisplay eglDisplay() + LOutputPrivate * imp() const + LOutput() + LOutput(const LOutput &amp;)=delete + const char * manufacturer() const + const char * model() const + const list&lt;LOutputMode *&gt; * modes() const + const char * name() const + LOutput &amp; operator=(const LOutput &amp;)=delete + LPainter * painter() const + const LSize &amp; physicalSize() const + const LPoint &amp; posC() const + const LOutputMode * preferredMode() const + const LRect &amp; rectC() const + void repaint() + Int32 scale() const + LSeat * seat() const + void setMode(const LOutputMode *mode) + void setPosC(const LPoint &amp;posC) + void setScale(Int32 scale) + const LSize &amp; sizeB() const + const LSize &amp; sizeC() const + State state() const + virtual ~LOutput() * virtual void initializeGL() * virtual void paintGL() * virtual void resizeGL() * virtual void uninitializeGL()</pre>

Figura 14: Miembros de la clase LOutput.

---

<sup>5</sup> Imagen disponible desde <https://www.eteknix.com/testing-mixed-resolution-amd-eyefinity-6400x1080/>. Accedido el 29 de diciembre de 2022.

#### **4.2.1 Acceso a las salidas disponibles**

El backend gráfico se encarga de crear cada LOutput realizando una solicitud al compositor mediante el constructor virtual LCompositor::createOutputRequest(). El desarrollador puede reimplementar este constructor virtual para utilizar su propia subclase de LOutput e implementar su propia lógica de renderizado. La clase LOutputManager permite acceder a una lista de todas las salidas creadas por el backend gráfico a través del atributo LOutputManager::outputs() y también notifica cuando una nueva salida está disponible o deja de estarlo, por ejemplo al conectar o desconectar un monitor en un puerto HDMI.

#### **4.2.2 Inicialización**

Por defecto, las salidas están inactivas y no se puede renderizar en ellas. Para activar una salida, se debe agregar al compositor mediante el método LCompositor::addOutput(). Esto inicializará su hilo de renderizado, su contexto gráfico e invocará el método virtual initializeGL() de LOutput, en el que se puede realizar una configuración inicial como por ejemplo cargar texturas, compilar shaders, cargar buffers de vértices, etc.

Cada vez que se llame al método repaint(), se programará el siguiente frame de renderizado y la librería invocará el método paintGL(), en el que el desarrollador puede definir su propia lógica de renderizado. Llamar al método repaint() múltiples veces durante el mismo frame no significa que se invocará varias veces el método paintGL(), solo se asegura que se invocará una vez en el siguiente frame.

Por defecto, la librería inicializa todas las salidas disponibles en la implementación predeterminada de LCompositor::initialized().

#### **4.2.3 Modos de una salida**

Cada LOutput puede tener varios modos. Un LOutputMode contiene información sobre la resolución y la tasa de refresco con la que puede funcionar una salida. Se puede acceder a la lista de modos de una salida mediante el atributo modes() y establecer el deseado con setMode(). Por lo general, las salidas utilizan por defecto el modo con mayor tasa de refresco y resolución. Si se cambia a un modo con una resolución distinta a la actual y la salida está inicializada, el backend gráfico invoca el método resizeGL() para notificar un cambio en el tamaño del framebuffer.

#### **4.2.4 Contextos gráficos**

Cada salida tiene su propio contexto OpenGL y su propia instancia de LPainter, que se puede acceder mediante el método painter(). Se puede utilizar las funciones proporcionadas por LPainter para renderizar rectas de colores o texturas o utilizar funciones nativas de OpenGL y shaders/programas propios si se desea.

#### 4.2.5 Layout

Las salidas tienen una posición y dimensiones que permiten organizarlas lógicamente, similar a como lo hace un panel de preferencias (ver Figura 15). Se puede establecer la posición de una salida mediante el método `setPosC()`.

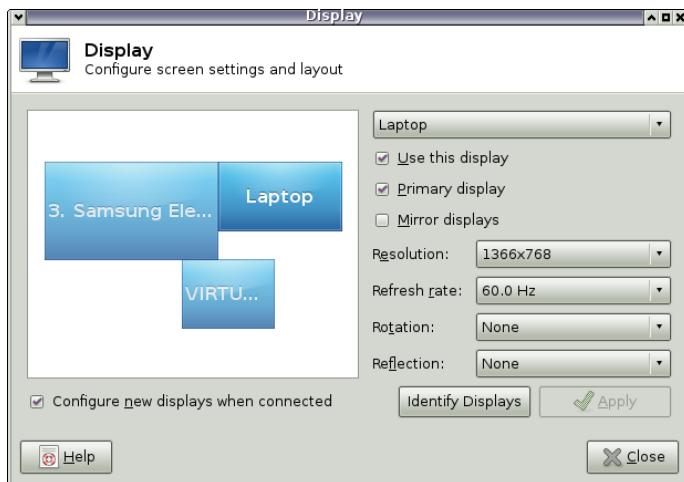


Figura 15: Layout de múltiples salidas en un panel de preferencias.<sup>6</sup>

#### 4.2.6 De-inicialización

Si una salida deja de estar disponible o ya no se desea utilizar, se debe llamar al método `LCompositor::removeOutput()` para eliminarlo del compositor. Esto solo desinicializa su hilo de renderizado, siendo posible volver a inicializarlo posteriormente. Un `LOutput` deja de estar disponible cuando la librería o backend gráfico invoca su destructor virtual (`LCompositor::destroyOutputRequest()`).

### 4.3 Clase LOutputManager

La clase `LOutputManager` permite acceder a la lista de salidas disponibles y generadas por el backend gráfico a través del atributo `outputs()` (ver Figura 16). Esta clase cuenta también con los métodos virtuales `outputPlugged()` y `outputUnplugged()` los cuales son invocados por el backend gráfico para notificar la conexión y desconexión de una salida, por ejemplo cuando se conecta o desconecta un monitor externo a través de un puerto HDMI o VGA. Solo existe una instancia de esta clase, la cual se puede acceder a través de `LCompositor::outputManager()`.

---

<sup>6</sup> Imagen disponible desde <https://eatpeppershotspot.blogspot.com/2016/12/using-external-monitor-with-bumblebee.html>. Accedido el 29 de diciembre de 2022.

LOutputManager
<pre>+ LCompositor * compositor() const + LOutputManagerPrivate * imp() const + LOutputManager(const LOutputManager &amp;)=delete + LOutputManager(Params *params) + LOutputManager &amp; operator=(const LOutputManager &amp;)=delete + const list&lt; LOutput * &gt; * outputs() const + virtual ~LOutputManager() * virtual void outputPlugged(LOutput *output) * virtual void outputUnplugged(LOutput *output)</pre>

Figura 16: Miembros de la clase LOutputManager.

#### 4.4 Clase LOutputMode

La clase LOutputMode representa una configuración posible para una salida (LOutput), específicamente la tasa de refresco y resolución. Cada LOutput tiene uno o más modos que pueden ser accedidos a través de LOutput::modes() y asignados a la salida con LOutput::setMode() como se muestra en la Figura 17. Esta clase no tiene métodos virtuales y, por lo tanto, no es necesario reimplementarla.

LOutputMode
<pre>+ LOutputModePrivate * imp() const + bool isPreferred() const + LOutputMode(const LOutput *output) + LOutputMode(const LOutputMode &amp;)=delete + LOutputMode &amp; operator=(const LOutputMode &amp;)=delete + const LOutput * output() const + UInt32 refreshRate() const + const LSize &amp; sizeB() const + ~LOutputMode()</pre>

Figura 17: Miembros de la clase LOutputMode.

#### 4.5 Clase LPainter

La clase LPainter ofrece funciones básicas para dibujar y renderizar en 2D sin necesidad de utilizar OpenGL directamente. Permite dibujar líneas, rectas y subrectas de texturas o colores, limpiar la pantalla y asignar el viewport (ver Figura 18). Su objetivo es abstraer la API de renderizado y permitir la creación de compositores portables independientes del renderer utilizado. Actualmente, la librería solo ofrece el renderer OpenGL ES 2.0, pero a futuro podrían incorporarse otros como Vulkan o Pixman. Cada LOutput tiene su propio LPainter accesible a través de LOutput::painter(). No es obligatorio utilizar las funciones

de LPainter para renderizar, la librería permite al desarrollador utilizar directamente las funciones de OpenGL, o combinar ambos métodos si desea. En dicho caso, se debe llamar al método bindProgram() antes de utilizar las funciones provistas por LPainter.

LPainter
<pre>+ void bindProgram() + void clearScreen() + void drawColorC(const LRect &amp;dst, Float32 r, Float32 g, Float32 b, Float32 a) + void drawTextureC(LTexture *texture, const LRect &amp;srcB, const LRect &amp;dstG, Float32 alpha=1.f) + LPainterPrivate * imp() const + LPainter(const LPainter &amp;)=delete + LPainter &amp; operator=(const LPainter &amp;)=delete + void setClearColor(Float32 r, Float32 g, Float32 b, Float32 a) + void setViewportC(const LRect &amp;rect)</pre>

Figura 18: Miembros de la clase LPainter.

## 4.6 Clase LOpenGL

La clase LOpenGL proporciona un pequeño conjunto de funciones para facilitar la creación de aplicaciones OpenGL, la compilación de shaders y la carga de texturas a partir de archivos de imagen PNG como se muestra en la Figura 19.

LOpenGL
<pre>+ static bool checkGLError(const char *msg) + static GLuint compileShader(GLenum type, const char *shaderString) + static LTexture * loadTexture(const char *pngFile, GLuint textureUnit=1) + static GLuint maxTextureUnits() + static char * openShader(const char *fileName)</pre>

Figura 19: Miembros de la clase LOpenGL.

## 4.7 Clase LTexture

La clase LTexture es una abstracción de un buffer gráfico que permite generar texturas compatibles con el renderer a partir de buffers en memoria principal con formatos compatibles con Wayland (Freedesktop Organization, 2012-b), o a partir de buffers en la GPU a través de imágenes EGL (ver Figura 20). Esta clase es útil para crear texturas de forma rápida y sencilla sin tener que trabajar directamente con la API de OpenGL.

LTexture
<pre>+ GLenum format() const + GLuint id() + LTexturePrivate * imp() const + bool initialized() + LTexture(const LTexture &amp;)=delete + LTexture(GLuint textureUnit=1) + LTexture &amp; operator=(const LTexture &amp;)=delete + void setDataB(Int32 width, Int32 height, void *data, GLenum format=GL_BGRA, GLenum type=GL_UNSIGNED_BYTE, BufferSourceType sourceType=SHM) + const LSize &amp; sizeB() const + BufferSourceType sourceType() const + GLenum type() + GLuint unit() + ~LTexture()</pre>

Figura 20: Miembros de la clase LTexture.

#### 4.7.1 Mecanismos de intercambio de buffers

La librería implementa actualmente dos mecanismos para que los clientes puedan compartir buffers gráficos con el compositor:

##### Memoria compartida

En este mecanismo, los clientes alojan un buffer utilizando memoria compartida y envían su descriptor de archivo al compositor para que pueda mapearlo y convertirlo a una textura OpenGL. Este método es en la mayoría de los casos ineficiente puesto que el cliente (si renderiza en la GPU) debe transferir el buffer a la CPU, enviársela al compositor, y luego este debe volver a transferirla a la GPU como se muestra en la Figura 21.

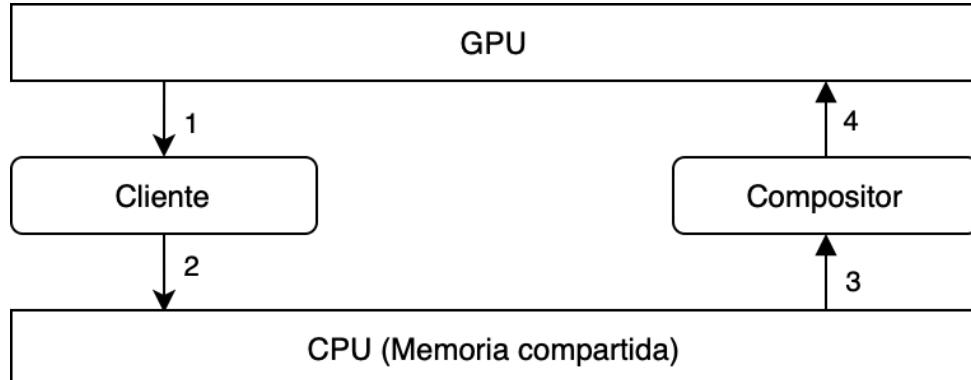


Figura 21: Intercambio de buffers gráficos mediante memoria compartida.

##### Direct Memory Access (DMA)

Este mecanismo implementa la API DMA de Linux mediante la librería EGL (Tomazic, 2020), la cual permite a los clientes compartir sus buffers con el compositor directamente desde la GPU.

Este método es bastante eficiente puesto que no requiere transferir los buffers de la GPU a la CPU y viceversa, como se muestra en la Figura 22.

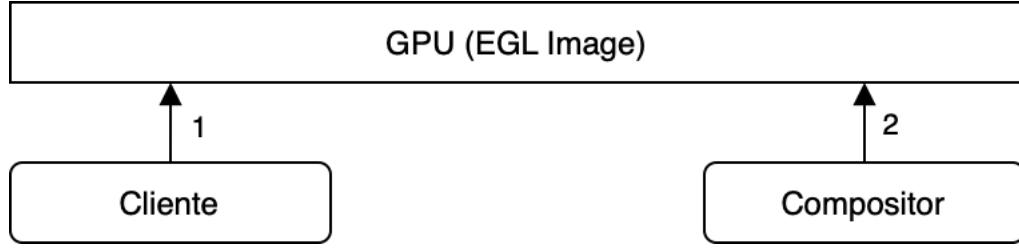


Figura 22: Intercambio de buffers gráficos mediante DMA.

## 4.8 Clase LSurface

La clase LSurface representa una superficie creada por un cliente y es una abstracción de la interfaz wl\_surface del protocolo de Wayland. Las superficies en Wayland son una representación de lo que típicamente se conoce como una ventana de una aplicación y tienen atributos como posición, tamaño, buffers, región de input, región opaca, región de daños y rol (ver Figura 23).

LSurface
+ wl_buffer * buffer() const + Int32 bufferScale() const + const list< LSurface *> & children() const + LClient * client() const + LCompositor * compositor() const + LCursorRole * cursor() const + const LRegion & damagesB() const + const LRegion & damagesC() const + LDNDIconRole * dndIcon() const + bool hasDamage() const + LSurfacePrivate * imp() const + const LRegion & inputRegionC() const + const LRegion & inputRegionsS() const + LSurface(Params &params, GLuint textureUnit=1) + bool mapped() const + bool minimized() const + const LRegion & opaqueRegionC() const + const LRegion & opaqueRegionsS() const + LSurface & operator=(const LSurface &)=delete + const list< LOutput *> & outputs() const + LSurface * parent() const + LPopupRole * popup() const + const LPoint & posC() const + bool receiveInput() const + void replaceOutput() + void reuseNewFrame() + wl_resource * resource() const + LBaseSurfaceRole * role() const + Role roleId() const + const LPoint & rolePosC() const + LSeat * seat() const + void sendOutputEnterEvent(LOutput *output) + void sendOutputLeaveEvent(LOutput *output) + void setMinimized(bool state) + void setPosC(const LPoint &newPos) + void setPosC(Int32 x, Int32 y) + void setXC(Int32 x) + void setYC(Int32 y) + const LSize & sizeB() const + const LSize & sizeS() const + const LSize & sizeC() const + LSubsurfaceRole * subsurface() const + LTexture * texture() const + LToplevelRole * toplevel() const + LSurface * topmostParent() const + const LRegion & translucentRegionC() const + const LRegion & translucentRegionS() const + wl_resource * xdgSurfaceResource() const + virtual ~LSurface() * virtual void damaged() * virtual void roleChanged() * virtual void parentChanged() * virtual void mappingChanged() * virtual void bufferSizeChanged() * virtual void opaqueRegionChanged() * virtual void inputRegionChanged() * virtual void raised()

Figura 23: Miembros de la clase LSurface.

Los clientes dibujan el contenido de sus superficies en un buffer y luego se lo envían al compositor para que este lo asocie a la superficie pertinente y lo muestre en pantalla.

#### 4.8.1 Roles

Las superficies, por sí solas, carecen de funcionalidad suficiente para ser desplegadas en pantalla. Por esta razón, existen roles que amplían sus funcionalidades y definen las reglas para ordenarlas, posicionarlas e interpretar sus geometrías. Actualmente, la librería incluye los siguientes roles:

- **LCursorRole** (definido en el protocolo de Wayland)
- **LDNDIconRole** (definido en el protocolo de Wayland)
- **LSubsurfaceRole** (definido en el protocolo de Wayland)
- **LPopupRole** (definido en el protocolo XDG Shell)
- **LToplevelRole** (definido en el protocolo XDG Shell)

Para acceder al rol de una superficie, se puede utilizar el atributo role() o los métodos cursor(), dndIcon(), popup(), toplevel() y subsurface(), dependiendo de si ya se conoce el rol o se quiere verificar si es alguno de ellos (retornan nullptr si no poseen el rol). Una vez que se asigna un rol a una superficie, este **por lo general** se mantiene durante todo su ciclo de vida. Cuando se le asigna un rol a una superficie o se reemplaza por otro, se notifica a través del método virtual roleChanged(). También, es posible crear roles adicionales a los que ofrece la librería, utilizando la clase LBaseSurfaceRole. Todos los roles incluidos en la librería heredan a esta clase como se muestra en la Figura 24.

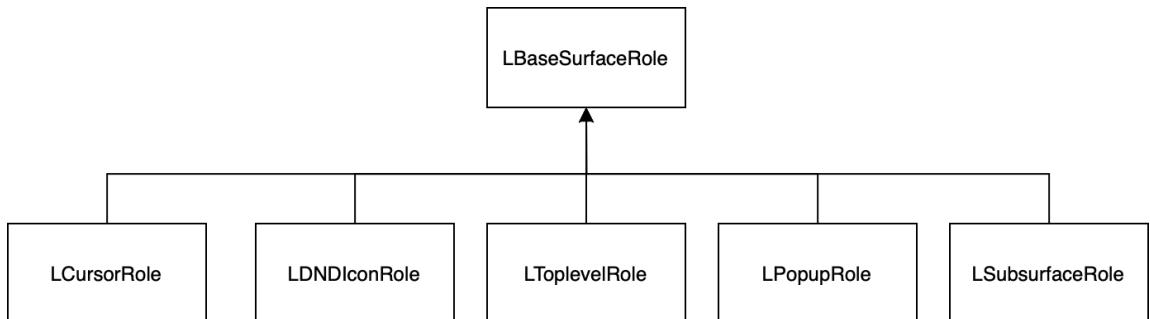


Figura 24: Clases que heredan a LBaseSurfaceRole.

#### 4.8.2 Buffers

La librería convierte automáticamente el buffer de las superficies en una textura utilizable por el renderer. Actualmente, la librería solo ofrece el renderer OpenGL ES 2.0, por lo que el buffer se convierte en una textura OpenGL (LTexture) accesible a través del método texture(). También es posible acceder al recurso Wayland nativo del buffer (wl\_buffer) mediante el método buffer().

#### **4.8.3 Mapeo**

Para poder mostrar una superficie en pantalla, es necesario que tenga asignado un rol y un buffer no nulo. La propiedad mapped() puede usarse para determinar si una superficie es renderizable. Esta propiedad devuelve falso cuando el cliente desea ocultarla mediante alguna regla de su rol o cuando no se cumplen las condiciones previamente mencionadas. El método virtual mappingChanged() puede ser utilizado para detectar un cambio de la propiedad mapped().

#### **4.8.4 Daños**

Para evitar que el compositor vuelva a dibujar el área completa de una superficie en cada frame, los clientes notifican qué regiones han cambiado en sus buffers, lo que se conoce como daños o damages (Ser, Introduction to damage tracking, 2019).

Se puede acceder a esta región mediante los métodos damagesB() o damagesC() y ser notificado cuando hay nuevos daños con el método virtual damaged().

La implementación por defecto del método virtual LOutput::paintGL() (donde se ejecuta el renderizado de una salida) no contempla los daños de las superficies, esto con el fin de facilitar la comprensión del código por parte de los desarrolladores, por lo tanto no renderiza de la forma más eficiente. Sin embargo, en el repositorio GitHub donde se aloja la librería se incluye un compositor de ejemplo llamado louvre-weston-clone, el cual muestra cómo renderizar de forma eficiente tomando en cuenta los daños de las superficies.

#### **4.8.5 Callbacks**

Para evitar que los clientes rendericen el contenido de una superficie a una tasa mayor a la de la salida (LOutput) en la que son visibles o cuando están ocultas por otras superficies, se utilizan callbacks. Los clientes crean un recurso wl\_callback para una superficie en específico (luego de haber renderizado su contenido) y esperan a que el compositor retorne el callback para comenzar a renderizar el próximo frame.

Para responder al callback de una superficie, se debe utilizar el método requestNextFrame(), el cual además limpia la región de daños actual de la superficie.

Si no se llama a requestNextFrame() la superficie no actualiza su contenido.

#### **4.8.6 Orden**

La librería utiliza una lista para llevar registro de todas las superficies creadas por los clientes accesible a través de LCompositor::surfaces(). Esta lista mantiene el orden de las superficies en el eje Z, definido por los protocolos de sus roles. Es posible reordenar las superficies respetando dichas jerarquías con el método LCompositor::raiseSurface(), el cual reubica una superficie (o grupo de superficies) al final de la lista.

Por defecto, las superficies se renderizan en LOutput::paintGL() siguiendo el orden de la lista, por lo que las primeras se ubican al fondo de la pantalla y las últimas al frente. Si el desarrollador desea utilizar su propia lista, puede utilizar los constructores y destructores virtuales de LSurface (LCompositor::createSurfaceRequest() y LCompositor::destroySurfaceRequest() respectivamente) para ser notificado cuando los clientes crean y eliminan una superficie. Al utilizar una lista propia, es importante tener en cuenta que algunos roles requieren que se respete un orden específico entre superficies para funcionar correctamente, por lo que es responsabilidad del desarrollador estudiar e implementar las reglas de sus protocolos correctamente en dicho caso.

#### 4.8.7 Posición

Una característica de los protocolos en Wayland es que por lo general impiden al cliente saber y controlar como el compositor posiciona sus superficies en pantalla. Por esta razón, las reglas de los roles de superficies a menudo definen la posición en base a un offset relativo a otra superficie o a una posición dada por el propio compositor. La librería simplifica el posicionamiento de superficies, permitiendo al desarrollador asignarla mediante el método setPosC() y acceder a la sugerida por su rol a través de rolePosC(). En algunos casos como en los roles LPopupRole o LSubsurfaceRole, la posición asignada mediante setPosC() no se tiene en cuenta, ya que se posicionan de forma relativa a otras superficies. Sin embargo, el desarrollador, tiene total libertad para posicionar y renderizar las superficies como deseé y puede perfectamente ignorar la posición sugerida por los roles.

### 4.9 Clase LBaseSurfaceRole

La clase abstracta LBaseSurfaceRole proporciona una interfaz base para crear roles de superficies compatibles con la librería. Permite implementar la lógica de posicionamiento de la superficie que adquirirá el rol (retornada en LSurface::rolePosC()), asignar el ID del rol, manejar solicitudes de la interfaz wl\_surface, entre otras funcionalidades (ver Figura 25). Todas las clases que representan roles de superficies en la librería heredan esta clase como se muestra en la Figura 24.

LBaseSurfaceRole
<pre>+ LBaseSurfaceRolePrivate * baseImp() const + LCompositor * compositor() const + LBaseSurfaceRole(const LBaseSurfaceRole &amp;)=delete + LBaseSurfaceRole(wl_resource *resource, LSurface *surface, UInt32 roleid) + LBaseSurfaceRole &amp; operator=(const LBaseSurfaceRole &amp;)=delete + wl_resource * resource() const + UInt32 roleid() + virtual const LPoint &amp; rolePosC() const =0 + LSeat * seat() const + LSurface * surface() const + virtual ~LBaseSurfaceRole() # virtual bool acceptCommitRequest(Globals::CommitOrigin origin) # virtual void globalScaleChanged(Int32 oldScale, Int32 newScale) # virtual void handleParentCommit() # virtual void handleParentMappingChange() # virtual void handleSurfaceBufferAttach(wl_resource *buffer, Int32 x, Int32 y) # virtual void handleSurfaceCommit() # virtual void handleSurfaceOffset(Int32 x, Int32 y)</pre>

Figura 25: Miembros de la clase LBaseSurfaceRole.

## 4.10 Clase LCursorRole

La clase LCursorRole es un rol para superficies que permite al compositor utilizarlas como texturas para el cursor. Los clientes crean el rol a través de la solicitud set\_cursor de la interfaz wl\_pointer del protocolo de Wayland y la librería automáticamente invoca el método LPointer::setCursorRequest() cuando un cliente que tiene el foco de puntero desea asignar un LCursorRole como cursor. Además, el método LPointer::setCursorRequest() también se invoca cuando cambia el hotspot del rol, por lo que no es necesario reimplementar esta clase para ser notificado de dichos cambios.

En la Figura 26 se muestran todos los miembros de esta clase.

LCursorRole
+ const LPoint & hotspotB() const + const LPoint & hotspotC() const + virtual void hotspotChanged() + const LPoint & hotspotS() const + LCursorRolePrivate * imp() const + LCursorRole(const LCursorRole &)=delete + LCursorRole(Params *params) + LCursorRole & operator=(const LCursorRole &)=delete + virtual const LPoint & rolePosC() const override + virtual ~LCursorRole()

Figura 26: Miembros de la clase LCursorRole.

## 4.11 Clase LDNDIconRole

La clase LDNDIconRole es un rol para superficies que permite al compositor interpretarlas como íconos para sesiones de arrastrar y soltar (drag & drop). Los clientes crean el rol a través de la solicitud start\_drag de la interfaz wl\_data\_device del protocolo de Wayland. Se puede acceder al rol LDNDIconRole de una sesión drag & drop desde LDNDManager::icon(). En la Figura 27 se muestran todos los miembros de esta clase.

LDNDIconRole
+ const LPoint & hotspotB() const + const LPoint & hotspotC() const + virtual void hotspotChanged() const + const LPoint & hotspotS() const + LDNDIconRolePrivate * imp() const + LDNDIconRole(const LDNDIconRole &)=delete + LDNDIconRole(Params *params) + LDNDIconRole & operator=(const LDNDIconRole &)=delete + virtual const LPoint & rolePosC() const override + virtual ~LDNDIconRole()

Figura 27: Miembros de la clase LDNDIconRole.

## 4.12 Clase LSubsurfaceRole

La clase LSubsurfaceRole es un rol para superficies basado en la interfaz wl\_subsurface del protocolo de Wayland que permite posicionarlas de forma relativa a sus padres. Las subsuperficies siempre son hijas de otras superficies y su posición se determina a partir de la posición de su superficie padre más un offset, el cual se puede acceder a través de los métodos localPosS() o localPosC() (ver Figura 28).

LSubsurfaceRole
+ LSubsurfaceRolePrivate * imp() const + bool isSynced() const + const LPoint & localPosC() const + const LPoint & localPosS() const + LSubsurfaceRole(const LSubsurfaceRole &)=delete + LSubsurfaceRole(Params *params) + LSubsurfaceRole & operator=(const LSubsurfaceRole &)=delete + virtual ~LSubsurfaceRole() * virtual const LPoint & rolePosC() const override * virtual void localPosChanged() * virtual void syncModeChanged() * virtual void placedAbove(LSurface *sibling) * virtual void placedBelow(LSurface *sibling)

Figura 28: Miembros de la clase LSubsurfaceRole.

### 4.12.1 Modos

Existen dos modos en los que las subsuperficies pueden trabajar:

#### Modo síncrono

En este modo, los cambios solicitados a su superficie por el cliente sólo se aplican cuando su padre realice un commit. Un commit es una solicitud de la interfaz wl\_surface del protocolo de Wayland que permite aplicar múltiples cambios en la superficie de manera atómica, como cambios en su escala, buffer, posición, etc. Esto permite a los clientes sincronizar la animación o movimiento de múltiples subsuperficies. La librería se encarga de registrar y aplicar internamente los cambios de la superficie cuando su padre realiza un commit.

#### Modo asíncrono

En este modo, los cambios realizados a la subsuperficie son aplicados de forma independiente a los commits de su padre. Esto es útil para clientes que deseen realizar una com-

posición eficiente de ventanas, como en reproductores de video donde la interfaz de usuario es independiente de la imagen que se está reproduciendo. De esta manera, los cambios a la interfaz de usuario pueden realizarse sin interferir en la reproducción del video y viceversa.

## 4.13 Clase LTopLevelRole

La clase LTopLevelRole es un rol para superficies utilizada por clientes para mostrar ventanas de aplicaciones que tienen un título y botones para cerrar, minimizar y maximizar, como se muestra en la Figura 29.

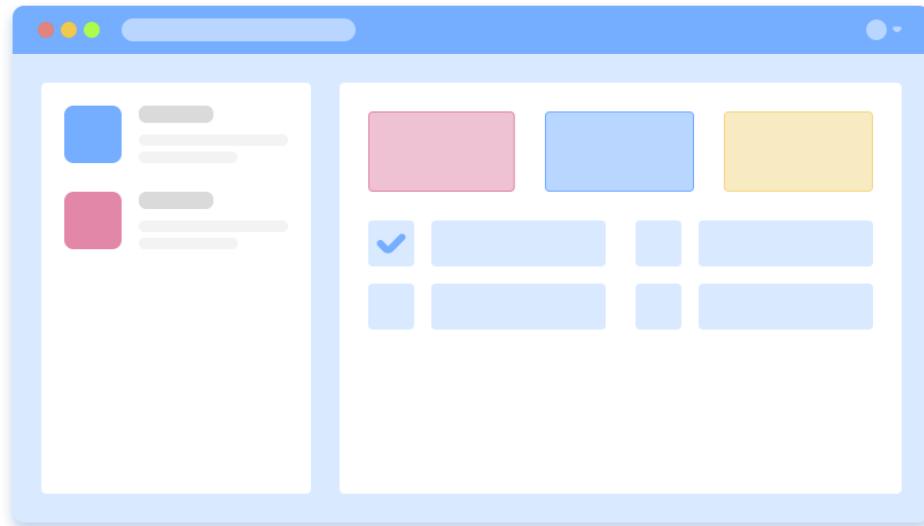


Figura 29: Ejemplo de superficie Toplevel.

Este rol forma parte del protocolo XDG Shell. El protocolo de Wayland también ofrece un rol Toplevel, pero se considera obsoleto, por esta razón no se implementa en la librería. En la Figura 30 se muestran todos los miembros de esta clase.

### 4.13.1 Geometría de ventana

La geometría de la ventana es una recta del buffer de la superficie Toplevel que excluye sus decoraciones (típicamente sombras). La geometría se compone de las coordenadas “x” e “y”, y el ancho y alto de la ventana, como se muestra en la Figura 31.

LTopLevelRole
<pre>+ bool activated() const + const char * appId() const + const LSize &amp; boundsC() const + const LSize &amp; boundsS() const + LSize calculateResizeSize(const LPoint &amp;cursorPosDelta, const LSize &amp;initialSize, ResizeEdge edge) + void close() + void configureBoundsC(const LSize &amp;bounds) + void configureC(const LSize &amp;size, States stateFlags) + void configureC(Int32 width, Int32 height, States stateFlags) + void configureC(States stateFlags) + DecorationMode decorationMode() const + bool fullscreen() const + LToplevelRolePrivate * imp() const + LToplevelRole(const LToplevelRole &amp;)=delete + LToplevelRole(Params *params) + bool maximized() const + const LSize &amp; maxSizeC() const + const LSize &amp; maxSizeS() const + const LSize &amp; minSizeC() const + const LSize &amp; minSizeS() const + LToplevelRole &amp; operator=(const LToplevelRole &amp;)=delete + void ping(UInt32 serial) + void setDecorationMode(DecorationMode mode) + void setWmCapabilities(UChar8 capabilitiesFlags) + States states() const + const char * title() const + const LRect &amp; windowGeometryC() const + const LRect &amp; windowGeometryS() const + UChar8 wmCapabilities() const + virtual ~LToplevelRole() * virtual const LPoint &amp; rolePosC() const override * virtual void startMoveRequest() * virtual void startResizeRequest(ResizeEdge edge) * virtual void configureRequest() * virtual void setMaximizedRequest() * virtual void unsetMaximizedRequest() * virtual void maximizedChanged() * virtual void setMinimizedRequest() * virtual void setFullscreenRequest(LOutput *destOutput) * virtual void unsetFullscreenRequest() * virtual void fullscreenChanged() * virtual void showWindowMenuRequestS(Int32 x, Int32 y) * virtual void pong(UInt32 serial) * virtual void activatedChanged() * virtual void maxSizeChanged() * virtual void minSizeChanged() * virtual void titleChanged() * virtual void appIdChanged() * virtual void geometryChanged() * virtual void decorationModeChanged()</pre>

Figura 30: Miembros de la clase LToplevelRole.

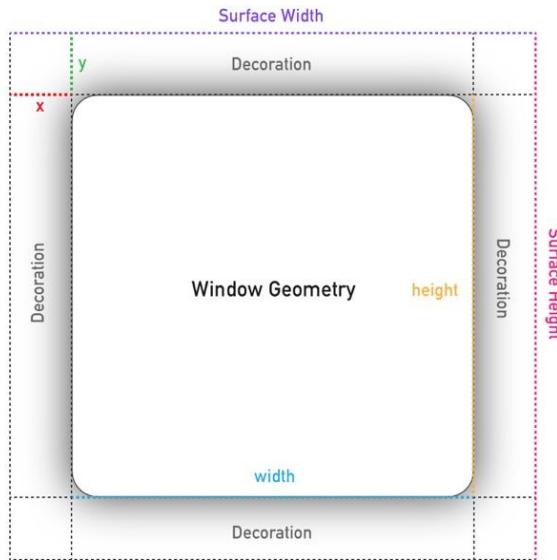


Figura 31: Geometría de ventana de una superficie Toplevel o Popup.

El rol LPopupRole también tiene una geometría de ventana y ambos roles la utilizan en la lógica de posicionamiento de sus superficies.

#### 4.14 Clase LPopupRole

La clase LPopupRole es un rol para superficies utilizada por clientes para mostrar elementos emergentes como menús o tooltips como se aprecia en la Figura 32.

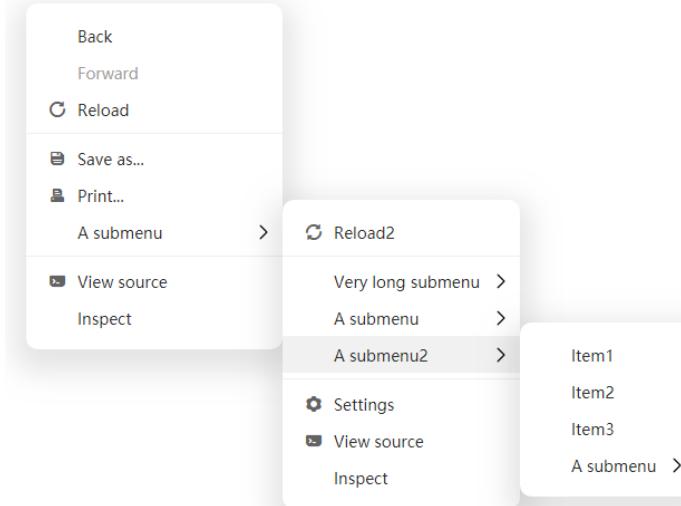


Figura 32: Ejemplo de superficies Popups anidadas.

Las superficies Popup siempre son hijas de otras superficies (otros Popups o Toplevels). Su posición se determina mediante reglas complejas descritas en su instancia de LPositioner. La implementación por defecto de rolePosC() implementa dichas reglas, permitiendo restringir el área donde se debe posicionar el Popup mediante el método setPositionerBoundsC(). El rol de Popup forma parte del protocolo XDG Shell. El protocolo de Wayland también ofrece su propio rol de Popup, pero es considerado obsoleto y por lo tanto no es incluido en la librería. En la Figura 33 se muestran todos los miembros de esta clase.

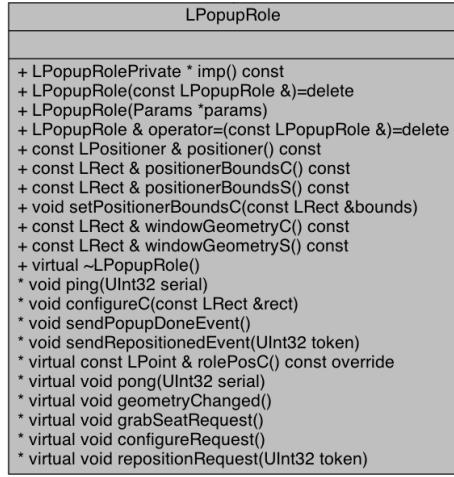


Figura 33: Miembros de la clase LPopupRole.

## 4.15 Clase LPositioner

La clase LPositioner define las reglas para posicionar un Popup en relación al punto de anclaje de su padre. Cada instancia de LPopupRole tiene su propio LPositioner, que puede accederse a través de LPopupRole::positioner(). En la Figura 34 se muestran todos los miembros de esta clase.

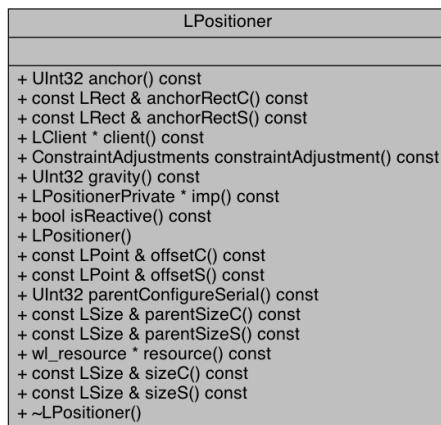


Figura 34: Miembros de la clase LPositioner.

#### 4.15.1 Recta de anclaje

La recta de anclaje (accesible mediante `anchorRectS()` o `anchorRectC()`) es un subrectángulo de la geometría de ventana de la superficie padre del Popup. Dentro de esta recta se define un punto de anclaje (accesible a través de `anchor()`) que el Popup utiliza como referencia para su posicionamiento. El punto de anclaje puede estar ubicado en el centro, en las esquinas o en la mitad de los bordes de la recta de anclaje (puntos grises y azules en la Figura 35). Por ejemplo, en la Figura 35, se posiciona un Popup utilizando el punto de anclaje `AnchorRight` y `AnchorBottomLeft`.

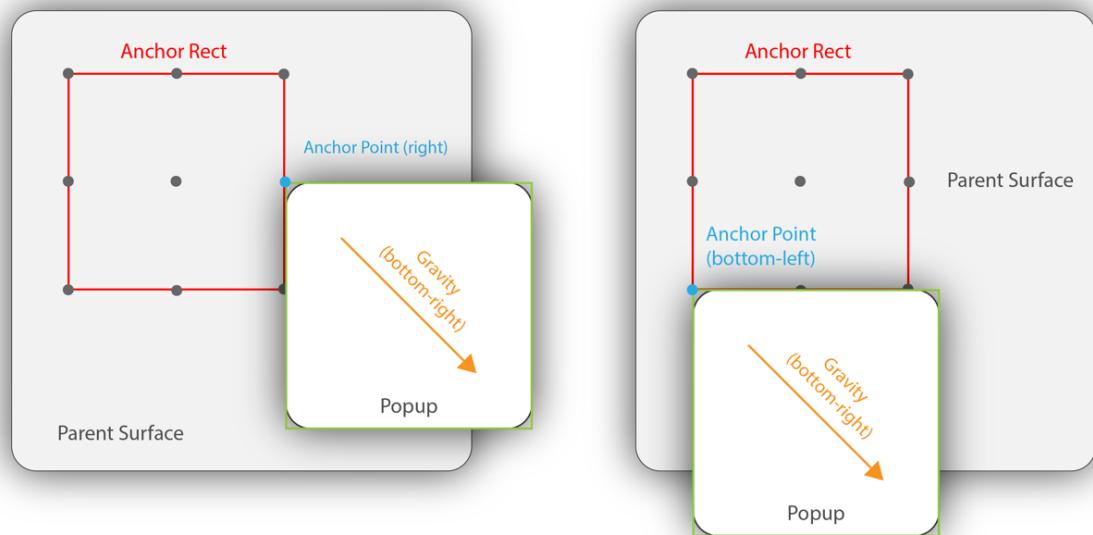


Figura 35: Posición del Popup según el punto de anclaje de la recta de anclaje.

#### 4.15.2 Gravedad

La gravedad del Popup, accesible a través de `gravity()`, indica la dirección hacia la que intenta moverse el Popup. Se puede considerar el punto de anclaje como un clavo y el Popup como un marco compuesto solo por bordes. Si la gravedad es hacia abajo, el borde superior del Popup chocará con el clavo, impidiendo que se siga moviendo. En la Figura 36 se muestra un Popup con gravedad `Gravity::GravityBottomRight` y `Gravity::GravityTopLeft`.

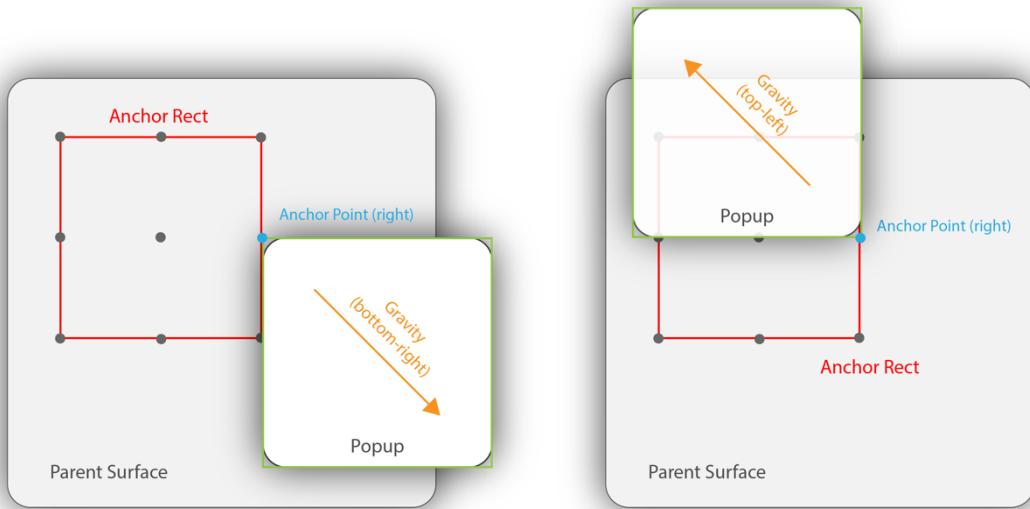


Figura 36: Cambio de posición del Popup según su gravedad.

#### 4.15.3 Ajustes de restricción

En los casos en que la posición inicial de un Popup resulte en una restricción, como quedar fuera del área visible de la pantalla, existen reglas específicas para ajustar su posición. Estas reglas son definidas por el cliente y accesibles mediante el atributo constraintAdjustment(). El proceso para ajustar la posición incluye intentar invertir la gravedad y punto de anclaje en los ejes donde ocurre la restricción, trasladar el Popup si es necesario, y configurarlo para ajustar su tamaño. Si ninguna de estas opciones elimina la restricción, se utiliza la posición restringida originalmente.

### 4.16 Clase LSeat

La clase LSeat representa un asiento del compositor. Un asiento es un conjunto de dispositivos de entrada y salida, como el ratón, teclado, y monitores, utilizados en una sesión. Por lo general, el permiso para acceder a estos dispositivos está limitado a un solo proceso por sesión, que suele ser un compositor Wayland o un servidor X11. Para permitir que el compositor controle los dispositivos del asiento sin necesidad de ser superusuario, los backends DRM y Libinput utilizan el método openDevice() para abrir los dispositivos, el cual internamente implementa la función libseat\_open\_device() de la librería libseat para solicitar los permisos del asiento. La clase LSeat también permite cambiar de sesión (TTY), acceder al portapapeles, a las instancias de LPointer y LKeyboard, escuchar los eventos nativos del backend de input, entre otras funciones. En la Figura 37 se muestran todos los miembros de esta clase.

LSeat
<pre>+ LTopLevelRole * activeToplevel() const + UInt32 backendCapabilities() const + void * backendContextHandle() const + UInt32 capabilities() const + LCompositor * compositor() const + LCursor * cursor() const + LDataSource * dataSelection() const + LDNDManager * dndManager() const + LSeatPrivate * imp() const + LSeatKeyboard * keyboard() const + LSeat(const LSeat &amp;)=delete + LSeat(Params *params) + LSeat &amp; operator=(const LSeat &amp;)=delete + LPointer * pointer() const + void setCapabilities(UInt32 capabilitiesFlags) * virtual ~LSeat() * Int32 setTTY(Int32 tty) * Int32 openDevice(const char *path, Int32 *fd) * Int32 closeDevice(Int32 id) * libseat * libseatHandle() const * bool enabled() const * virtual void initialized() * virtual void setSelectionRequest(LDataDevice *device) * virtual void backendNativeEvent(void *event) * virtual void seatEnabled() * virtual void seatDisabled()</pre>

Figura 37: Miembros de la clase LSeat.

## 4.17 Clase LPointer

La clase LPointer permite acceder y manipular eventos del puntero (ratón, trackpad, etc) generados por el backend de input. Puede utilizarse para mover y escalar superficies TopLevel de manera interactiva, enviar eventos del puntero a superficies de clientes y configurar parámetros como el valor de paso de la rueda scroll del ratón (ver Figura 38). Existe una única instancia de LPointer, la cual puede ser accedida a través de LSeat::pointer().

LPointer
<pre>+ LCompositor * compositor() const + LCursor * cursor() const + LSurface * draggingSurface() const + LSurface * focusSurface() const + LSeat * seat() const + LPointer(const LPointer &amp;)=delete + LPointer(Params *params) + LPointer &amp; operator=(const LPointer &amp;)=delete + LSeat * seat() const + LSeat::getDragSurface(LSurface *surface) + LSurface * surfaceAt(const LPoint &amp;point) + virtual ~LPointer() * void setFocusCLSurface('surface') * void setFocusCLSurface('surface', const LPoint &amp;localPosG) * void sendMoveEvent() * void sendMoveEvent(const LPoint &amp;localPosG) * void sendMoveEvent(const Button button, ButtonState state) * void sendAxisEvent(double x, double y, UInt32 source) const LPoint &amp; scrollWheelStep() const * void setScrollWheelStep(const LPoint &amp;step) * void startResizingTopLevel(CLTopLevelRole *toplevel, LToplevelRole::ResizeEdge edge, Int32 L=EdgeDisabled, Int32 T=EdgeDisabled, Int32 R=EdgeDisabled, Int32 B=EdgeDisabled) * void updateResizingTopLevelSize() * void updateResizingTopLevelPos() * void stopResizingTopLevel() * LToplevelRole * resizingTopLevel() const * const LPoint &amp; movingTopLevelInitPos() const * const LSize &amp; resizingTopLevelInitSize() const * const LPoint &amp; resizingTopLevelInitCursorPos() const * LToplevelRole::ResizeEdge resizingTopLevelEdge() const void startMovingTopLevel(CLTopLevelRole *toplevel, Int32 L=EdgeDisabled, Int32 T=EdgeDisabled, Int32 R=EdgeDisabled, Int32 B=EdgeDisabled) * void stopMovingTopLevel() * LToplevelRole * movingTopLevel() const const LPoint &amp; movingTopLevelInitPos() const const LPoint &amp; movingTopLevelInitCursorPos() const * UInt32 lastPointerLeaveEventSerial() const * UInt32 lastPointerButtonEventSerial() const * virtual void pointerMoveEvent(float dx, float dy) * virtual void pointerPosChangeEvent(float x, float y) * virtual void pointerButtonEvent(Button button, ButtonState state) * virtual void pointerAxisEvent(double x, double y, UInt32 source) * virtual void setCursorRequest(CursorRole *cursorRole)</pre>

Figura 38: Miembros de la clase LPointer.

## 4.18 Clase LKeyboard

La clase LKeyboard permite acceder a eventos del teclado generados por el backend de input. Proporciona métodos para asignar el mapa del teclado, enviar eventos a las superficies de los clientes y configurar parámetros como la tasa de repetición al mantener presionada una tecla (ver Figura 39). Existe una única instancia de LKeyboard, la cual se puede obtener a través de LSeat::keyboard(). Esta instancia es útil para realizar tareas como la detección de teclas presionadas o la obtención de información sobre el estado del teclado.

LKeyboard
+ LCompositor * compositor() const + LSurface * focusSurface() const + LKeyboardPrivate * imp() const + bool isModActive(const char *name) const + virtual void keyEvent(UInt32 keyCode, UInt32 keyState) + Int32 keymapFd() const + Int32 keymapSize() const + xcb_state_t keymapState() const + virtual void keyModifiersEvent(UInt32 depressed, UInt32 latched, UInt32 locked, UInt32 group) + xcb_keysym_t keySymbol(UInt32 keyCode) + LKeyboard(const LKeyboard &)=delete + LKeyboard(Params &params) + const KeyboardModifiersState & modifiersState() const + LKeyboard & operator=(const LKeyboard &)=delete + Int32 repeatDelay() const + Int32 repeatRate() const + LSeat * seat() const + void sendKeyEvent(UInt32 keyCode, UInt32 keyState) + void sendModifiersEvent() + void sendModifiersEvent(UInt32 depressed, UInt32 latched, UInt32 locked, UInt32 group) + void setFocus(LSurface *surface) + void setKeymap(const char *rules=NULL, const char *model=NULL, const char *layout=NULL, const char *variant=NULL, const char *options=NULL) + void setRepeatInfo(Int32 rate, Int32 msDelay) + virtual ~LKeyboard()

Figura 39: Miembros de la clase LKeyboard.

## 4.19 Clase LCursor

La clase LCursor ofrece una forma sencilla de dibujar cursores en la pantalla. Además, aprovecha las capacidades de algunos backends gráficos para renderizar de manera más eficiente utilizando composición vía hardware (Linux Kernel Development Community, s.f.-b). Esta clase se utiliza principalmente para simplificar la visualización de cursores y mejorar el rendimiento. La lista de miembros de esta clase se puede ver en la Figura 40.

LCursor
+ LCompositor * compositor() const + bool hasHardwareSupport() const + const LPointF & hotspotB() const + LCursorPrivate * imp() const + const std::list<LOutput *> & intersectedOutputs() const + LCursor(const LCursor &)=delete + LXCursor * loadXCursorB(const char *cursor, const char *theme=NULL, Int32 suggestedSize=64, GLuint textureUnit=1) + void moveC(float dx, float dy) + LCursor & operator=(const LCursor &)=delete + LOutput * output() const + const LPointI & posC() const + const LRect & rectC() const + void repaintOutputs() + void setHotspotB(const LPointF &hotspot) + void setOutput(LOutput *output) + void setPosC(const LPointF &pos) + void setSizeS(const LSizeF &size) + void setTextureB(LTexture *texture, const LPointF &hotspot) + void setVisible(bool state) + LTexture * texture() const + void useDefault() + bool visible() const

Figura 40: Miembros de la clase LCursor.

#### 4.19.1 Inicialización

Existe una única instancia de LCursor accesible desde LCompositor::cursor(), la cual es creada durante la inicialización de la primera salida (LOutput) añadida al compositor con el método LCompositor::addOutput() y notificada mediante LCompositor::cursorInitialized(). El cursor debe estar siempre asignado a al menos una salida. Para seleccionar la salida actual se utiliza el método LCursor::setOutput(). La librería se encarga automáticamente de asignar la salida actual del cursor en base a su posición, por lo que no es necesario utilizar el método LCursor::setOutput() directamente a menos que se utilice un sistema de coordenadas distinto al espacio compositor (en el Capítulo 5 se describen en detalle los sistemas de coordenadas utilizados en la librería).

#### 4.19.2 Composición vía hardware

Algunos backends gráficos, como el backend DRM, permiten realizar composición de cursores vía hardware, lo que evita tener que volver a repintar salidas cada vez que el cursor cambia de posición. Para determinar si el backend lo soporta, se puede utilizar el método hasHardwareSupport(). Si el backend no lo soporta, se debe renderizar utilizando OpenGL. En ese caso, es posible acceder a la recta del cursor (posición y tamaño) en coordenadas de compositor mediante el método rectC() y a su textura mediante el método texture().

### 4.20 Clase LXCursor

La clase LXCursor representa un pixmap de un cursor X11 cargado utilizando el método LCursor::loadX11Cursor(). Proporciona acceso a su textura OpenGL y su hotspot (ver Figura 41). Los cursores X11 son ampliamente utilizados en Linux y hay una gran variedad de temas disponibles. El método LCursor::loadX11Cursor() permite cargar un cursor X11 especificando su nombre, tema y tamaño sugerido.

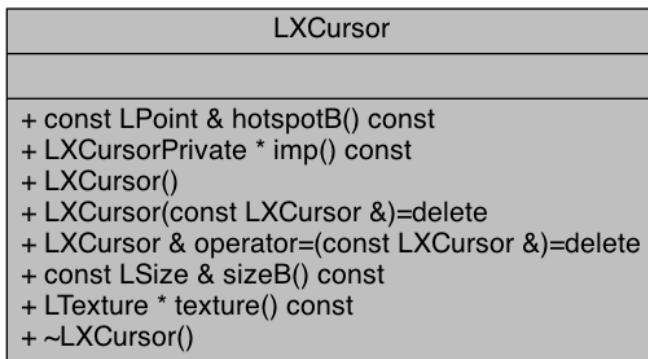


Figura 41: Miembros de la clase LXCursor.

## 4.21 Clase LClient

La clase LClient representa un cliente Wayland conectado al compositor. Brinda acceso a diversos recursos del cliente generados por la librería original de Wayland, a sus superficies, últimos seriales utilizados por los protocolos, entre otros atributos como se muestra en la Figura 42.

LClient
+ wl_client * client() const + LCompositor * compositor() const + wl_resource * compositorResource() const + LDataDevice * dataDevice() const + LClientPrivate * imp() const + wl_resource * keyboardResource() const + LClient(const LClient &)=delete + LClient(Params *params) + LClient & operator=(const LClient &)=delete + const list<wl_resource * > & outputs() const + wl_resource * pointerResource() const + LSeat * seat() const + wl_resource * seatResource() const + const Serials & serials() const + const list<LSurface * > & surfaces() const + wl_resource * touchResource() const + wl_resource * xdgDecorationManagerResource() const + wl_resource * xdgWmBaseResource() const + ~LClient()

Figura 42: Miembros de la clase LClient.

## 4.22 Clase LDataDevice

La clase LDataDevice representa la interfaz wl\_data\_device del protocolo de Wayland y es utilizado por clientes para manejar sesiones drag & drop y compartir información mediante el portapapeles. Solo puede existir un dispositivo de datos por cliente, el cual puede ser accedido desde LClient::dataDevice(). Los clientes que disponen de un dispositivo de datos pueden solicitar asignar la información del portapapeles e iniciar sesiones drag & drop. Para esto, crean una fuente de datos (LDataSource) que contiene la información que desean compartir, la librería luego notifica a otros clientes sobre la presencia de dicha fuente de datos cuando adquieren foco de puntero, teclado o touch mediante una ofrenda de datos (LDataOffer). Finalmente, los clientes pueden solicitar adquirir dicha información, por ejemplo, cuando el usuario presiona “Ctrl+V”.

La librería internamente se encarga de coordinar el intercambio de información entre dispositivos de datos, pero brinda control sobre que cliente puede asignar el portapapeles o

iniciar sesiones drag & drop. En la Figura 43 se muestran todos los miembros de esta clase.

LDataDevice
+ LClient * client() const + LDataOffer * dataOffered() const + LDataDevicePrivate * imp() const + UInt32 lastDataOfferId() const + LDataDevice(const LDataDevice &)=delete + LDataDevice & operator=(const LDataDevice &)=delete + wl_resource * resource() const + LSeat * seat() const + void sendSelectionEvent()

Figura 43: Miembros de la clase LDataDevice.

#### 4.23 Clase LDataSource

La clase LDataSource representa la interfaz wl\_data\_source del protocolo de Wayland y se utiliza para permitir a los clientes compartir información con otros a través del portapapeles o en sesiones drag & drop. Los clientes crean una instancia de LDataSource y especifican los tipos de datos que desean compartir, luego la librería se encarga de notificar a otros clientes sobre la presencia de la fuente de datos mediante una ofrenda de datos (LDataOffer). Los clientes pueden solicitar la información compartida mediante una solicitud de adquisición de datos, y la librería se encarga de coordinar el intercambio de información entre dispositivos de datos. En la Figura 44 se muestran todos los miembros de esta clase.

LDataSource
+ LClient * client() const + UInt32 dndActions() const + LDataSourcePrivate * imp() + LDataSource(const LDataSource &)=delete + LDataSource & operator=(const LDataSource &)=delete + wl_resource * resource() const + const std::list< LSource > & sources() const

Figura 44: Miembros de la clase LDataSource.

#### 4.24 Clase LDataOffer

La clase LDataOffer representa la interfaz wl\_data\_offer del protocolo de Wayland. Se utiliza para enviar información sobre una fuente de datos (LDataSource) a otro cliente

cuando una de sus superficies adquiere foco de puntero, teclado o touch. Esta información incluye las características de la fuente de datos, como el tipo de datos que contiene y cómo está disponible para ser compartida (a través del portapapeles o en una sesión drag & drop). Es principalmente utilizada por la librería y no es necesario interactuar con ella directamente. En la Figura 45 se muestran todos los miembros de esta clase.

LDataOffer
<pre>+ LDataDevice * dataDevice() const + LDataOfferPrivate * imp() const + LDataOffer(const LDataOffer &amp;)=delete + LDataOffer &amp; operator=(const LDataOffer &amp;)=delete + wl_resource * resource() const + LSeat * seat() const + Usage usedFor() const</pre>

Figura 45: Miembros de la clase LDataOffer.

#### 4.25 Clase LDNDManager

La clase LDNDManager permite controlar las sesiones drag & drop de clientes y puede ser accedida a través del método LSeat::dndManager(). Esta clase cuenta con métodos virtuales que notifican cuando un cliente desea iniciar o cancelar una sesión drag & drop, así como también métodos para "soltar" o cancelar una ofrenda de datos (ver Figura 46).

LDNDManager
<pre>+ void cancel() + virtual void cancelled() + bool dragging() const + void drop() + LSurface * focus() const + LDNDIconRole * icon() const + LDNDManagerPrivate * imp() const + LDNDManager(const LDNDManager &amp;)=delete + LDNDManager(Params *params) + LDNDManager &amp; operator=(const LDNDManager &amp;)=delete + LSurface * origin() const + Action preferredAction() const + LSeat * seat() const + void setPreferredAction(Action action) + LDataSource * source() const + virtual void startDragRequest() + virtual ~LDNDManager()</pre>

Figura 46: Miembros de la clase LDNDManager.

## 4.26 Clase LLog

La clase LLog permite imprimir mensajes de depuración en un stream de salida. Se puede controlar el nivel de detalle de los mensajes impresos mediante la asignación de un valor entero a la variable de entorno LOUVRE\_DEBUG. Los diferentes valores disponibles se detallan en la Tabla 7.

Tabla 7: Niveles de verbosidad de LLog.

Nivel	Descripción
LOUVRE_DEBUG=0	Desactiva los mensajes excepto los generados por log().
LOUVRE_DEBUG=1	Imprime los mensajes generados por log() y fatal().
LOUVRE_DEBUG=2	Imprime los mensajes generados por log(), fatal() y error().
LOUVRE_DEBUG=3	Imprime los mensajes generados por log(), fatal(), error() y warning().
LOUVRE_DEBUG=4	Imprime los mensajes generados por log(), fatal(), error(), warning() y debug().

En la Figura 47 se muestran todos los miembros de esta clase.

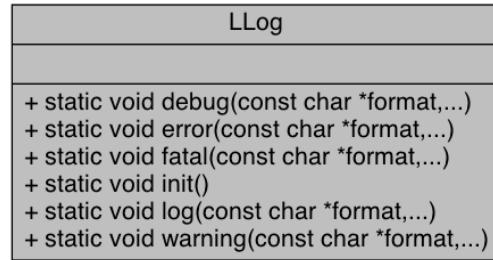


Figura 47: Miembros de la clase LLog.

## 4.27 Clase LTime

La clase LTime es utilizada para acceder tiempo del sistema en milisegundos o microsegundos sin una base de tiempo específica (ver Figura 48). Puede ser utilizada para realizar animaciones y operaciones temporales.

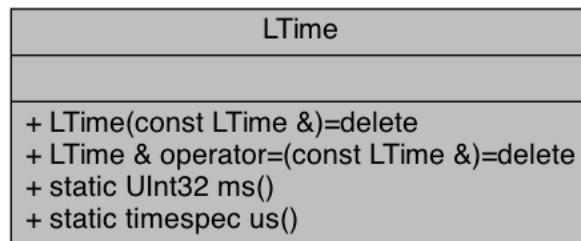


Figura 48: Miembros de la clase LTime.

## 4.28 Clase LPointTemplate

La clase LPointTemplate es una plantilla utilizada para crear vectores de dos dimensiones con diferentes tipos de datos numéricos. La librería utiliza esta plantilla para generar cuatro clases: LPoint y LSize, que trabajan con enteros de 32 bits, y LPointF y LSizeF, que trabajan con flotantes de 32 bits. Estas clases son utilizadas para representar puntos y tamaños en diferentes contextos dentro de la librería. En la Figura 49 se muestran todos los miembros de esta clase/plantilla.

LPointTemplate< TA, TB >
+ TA area() const + LPointTemplate< TA, TB > constrainedToHeight(TA size) const + TA h() const + TA height() const + LPointTemplate() + LPointTemplate(const LPointTemplate< TB, TA > &point) + LPointTemplate(TA n) + LPointTemplate(TA x, TA y) + LPointTemplate(TB n) + LPointTemplate(TB x, TB y) + bool operator!=(const LPointTemplate< TA, TB > &p) const + LPointTemplate< TA, TB > operator*(const LPointTemplate< TA, TB > &p) const + LPointTemplate< TA, TB > operator*(const LPointTemplate< TB, TA > &p) const + LPointTemplate< TA, TB > operator*(TA factor) const + LPointTemplate< TA, TB > operator*(TB factor) const + LPointTemplate< TA, TB > & operator*=(const LPointTemplate< TA, TB > &p) const + LPointTemplate< TA, TB > & operator*=(const LPointTemplate< TB, TA > &p) const + LPointTemplate< TA, TB > & operator*=(TA factor) + LPointTemplate< TA, TB > & operator*=(TB factor) + LPointTemplate< TA, TB > operator+(const LPointTemplate< TA, TB > &p) const + LPointTemplate< TA, TB > operator+(const LPointTemplate< TB, TA > &p) const + LPointTemplate< TA, TB > operator+(TA factor) const + LPointTemplate< TA, TB > operator+(TB factor) const + LPointTemplate< TA, TB > & operator+=(const LPointTemplate< TA, TB > &p) const + LPointTemplate< TA, TB > & operator+=(const LPointTemplate< TB, TA > &p) const + LPointTemplate< TA, TB > & operator+=(TA factor) + LPointTemplate< TA, TB > & operator+=(TB factor) + LPointTemplate< TA, TB > operator-(const LPointTemplate< TA, TB > &p) const + LPointTemplate< TA, TB > operator-(const LPointTemplate< TB, TA > &p) const + LPointTemplate< TA, TB > operator-(TA factor) const + LPointTemplate< TA, TB > operator-(TB factor) const + LPointTemplate< TA, TB > & operator-=(const LPointTemplate< TA, TB > &p) const + LPointTemplate< TA, TB > & operator-=(const LPointTemplate< TB, TA > &p) const + LPointTemplate< TA, TB > & operator-=(TA factor) + LPointTemplate< TA, TB > & operator-=(TB factor) + LPointTemplate< TA, TB > operator/(const LPointTemplate< TA, TB > &p) const + LPointTemplate< TA, TB > operator/(const LPointTemplate< TB, TA > &p) const + LPointTemplate< TA, TB > operator/(TA factor) const + LPointTemplate< TA, TB > operator/(TB factor) const + LPointTemplate< TA, TB > & operator/=(const LPointTemplate< TA, TB > &p) const + LPointTemplate< TA, TB > & operator/=(const LPointTemplate< TB, TA > &p) const + LPointTemplate< TA, TB > & operator/=(TA factor) + LPointTemplate< TA, TB > & operator/=(TB factor) + bool operator==(const LPointTemplate< TA, TB > &p) const + void setH(TA y) + void setHeight(TA y) + void setW(TA x) + void setWidth(TA x) + void setX(TA x) + void setY(TA y) + TA w() const + TA width() const + TA x() const + TA y() const

Figura 49: Miembros de la clase LPointTemplate.

## 4.29 Clase LRectTemplate

La clase LRectTemplate es una plantilla que permite crear vectores de cuatro dimensiones con diferentes tipos de datos numéricos. Estos vectores representan un rectángulo en un espacio 2D, compuesto por una posición (x, y) y un tamaño (ancho, alto).

La librería utiliza esta plantilla para generar la clase LRect, que representa rectángulos con enteros de 32 bits, y la clase LRectF, que representa rectángulos con flotantes de 32 bits. Estas clases se utilizan para realizar operaciones y comparaciones entre rectángulos. En la Figura 50 se muestran todos los miembros de esta clase/plantilla.

LRectTemplate< TA, TB >
+ TA area() const + const LSize & bottomRight() const + const LSize & BR() const + bool clip(const LRectTemplate &rect) + bool containsPoint(const LPointTemplate< TA, TB > &point, bool inclusive=true) const + TA h() const + TA height() const + bool intersects(const LRectTemplate &rect, bool inclusive=true) const + LRectTemplate() + LRectTemplate(const LPointTemplate< TA, TB > &p) + LRectTemplate(const LPointTemplate< TA, TB > &topLeft, const LPointTemplate< TA, TB > &bottomRight) + LRectTemplate(const LPointTemplate< TA, TB > &topLeft, TA bottomRight) + LRectTemplate(const LRectTemplate< TB, TA > &r) + LRectTemplate(TA all) + LRectTemplate(TA topLeft, const LPointTemplate< TA, TB > &bottomRight) + LRectTemplate(TA x, TA y, TA width, TA height) + bool operator!==(const LRectTemplate &p) const + LRectTemplate operator*(const LRectTemplate &r) const + LRectTemplate operator/(TA factor) const + LRectTemplate operator/(TB factor) const + LRectTemplate & operator+=(const LPointTemplate< TA, TB > &p) + LRectTemplate & operator+=(const LRectTemplate &r) + LRectTemplate & operator+=(TA factor) + LRectTemplate & operator+=(TB factor) + LRectTemplate operator+(const LRectTemplate &r) const + LRectTemplate operator+(TA factor) const + LRectTemplate operator+(TB factor) const + LRectTemplate & operator-=(const LPointTemplate< TA, TB > &p) + LRectTemplate & operator-=(const LRectTemplate &r) + LRectTemplate & operator-=(TA factor) + LRectTemplate & operator-=(TB factor) + LRectTemplate operator/(const LRectTemplate &r) const + LRectTemplate operator/(TA factor) const + LRectTemplate operator/(TB factor) const + LRectTemplate & operator/=(const LPointTemplate< TA, TB > &p) + LRectTemplate & operator/=(const LRectTemplate &r) + LRectTemplate & operator/=(TA factor) + LRectTemplate & operator/=(TB factor) + bool operator===(const LRectTemplate &p) const + const LPointTemplate< TA, TB > & pos() const + void setBottomRight(const LPointTemplate< TA, TB > &p) + void setBottomRight(const LPointTemplate< TB, TA > &p) + void setBR(const LPointTemplate< TA, TB > &p) + void setBR(const LPointTemplate< TB, TA > &p) + void setH(TA height) + void setHeight(TA height) + void setPos(const LPointTemplate< TA, TB > &p) + void setPos(const LPointTemplate< TB, TA > &p) + void setSize(const LPointTemplate< TA, TB > &p) + void setSizE(const LPointTemplate< TB, TA > &p) + void setTL(const LPointTemplate< TA, TB > &p) + void setTL(const LPointTemplate< TB, TA > &p) + void setTopLeft(const LPointTemplate< TA, TB > &p) + void setTopLeft(const LPointTemplate< TB, TA > &p) + void setW(TA width) + void setWidth(TA width) + void setY(TA y) + void setX(TA x) + const LSize & size() const + const LPointTemplate< TA, TB > & TL() const + const LPointTemplate< TA, TB > & topLeft() const + TA w() const + TA width() const + TA x() const + TA y() const

Figura 50: Miembros de la clase LRectTemplate.

## 4.30 Clase LRegion

La clase LRegion proporciona una forma eficiente de representar conjuntos de rectas sin solapar sus geometrías y utilizando la menor cantidad de rectángulos posibles. Además, ofrece métodos para realizar operaciones booleanas entre rectángulos o regiones, como uniones, substracciones, intersecciones, etc (ver Figura 51). Esta clase es ampliamente utilizada por la librería para calcular el daño de superficies, definir secciones opacas, translúcidas y las que reciben input de puntero. Para llevar a cabo estas operaciones, utiliza internamente los algoritmos y métodos de la librería Pixman (Freedesktop Organization, 2022-c).

LRegion
+ void addRect(const LRect &rect) + void addRegion(const LRegion &region) + void clear() + void clip(const LRect &rect) + bool containsPoint(const LPoint &point) const + void copy(const LRegion &regionToCopy) + bool empty() const + void intersectRegion(const LRegion &region) + void inverse(const LRect &rect) + LRegion() + LRegion(const LRegion &) + void multiply(Float32 factor) + void offset(const LPoint &offset) + LRegion & operator=(const LRegion &) + const vector< LRect > & rects() const + void subtractRect(const LRect &rect) + void subtractRegion(const LRegion &region) + ~LRegion()

Figura 51: Miembros de la clase LRegion.

## 5. RENDIMIENTO

Para evaluar el rendimiento de la librería, se desarrolló un compositor que imita visualmente a Weston (el compositor de referencia de Wayland) para comparar sus tasas de refresco y consumo de recursos de CPU y GPU. Weston se caracteriza por su alta eficiencia, es desarrollado y mantenido por los creadores de Wayland y por la misma razón se ha utilizado como punto de comparación (Wayland Project, 2022).

### 5.1 Benchmark

Para medir el rendimiento de ambos compositores, se desarrolló un benchmark que consiste en un cliente de Wayland que crea varias superficies opacas y transparentes de forma intercalada, las cuales se mueven de manera pseudoaleatoria por la pantalla como se muestra en la Figura 52.

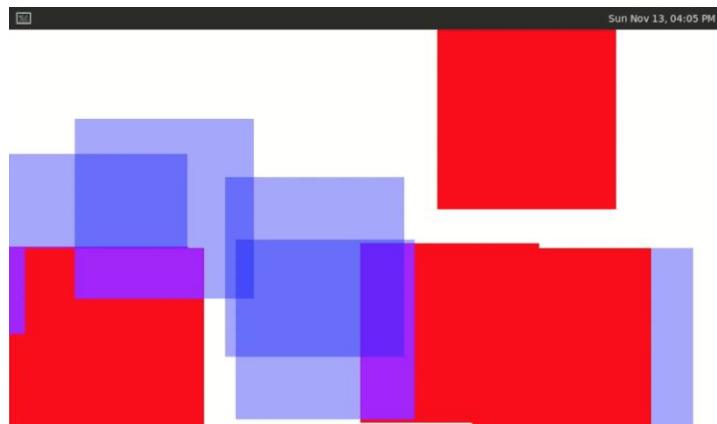


Figura 52: Superficies del benchmark siendo renderizadas en Weston.

La posición, velocidad y dirección inicial de las superficies son generadas aleatoriamente en base a una semilla. Luego, las posiciones se actualizan utilizando funciones sinusoidales en función del tiempo, lo que hace que las superficies se muevan de un extremo a otro de la pantalla tanto vertical como horizontalmente. Como la posición se calcula en función del tiempo, ambos compositores renderizan exactamente el mismo escenario durante una iteración del benchmark, independientemente de la cantidad de cuadros que alcancen a renderizar.

Este benchmark es útil para medir el rendimiento de compositores ya que, en general, estos tratan de renderizar exclusivamente las regiones en pantalla que cambian entre frames y evitan dibujar elementos no visibles, como superficies ocultas por elementos opacos. Calcular las regiones que deben renderizarse implica un elevado número de operaciones booleanas entre rectas, lo que tiene un impacto directo en el consumo de CPU. Por otro lado, si un compositor calcula incorrectamente las regiones cambiantes (dañadas),

podría renderizar zonas innecesarias, lo que se vería reflejado en un aumento del consumo de GPU.

### 5.1.1 Cálculo de consumo de CPU

Para medir el consumo de CPU, el benchmark utiliza el comando *ps* de Linux (Kerrisk, ps command, 2022) al final de cada iteración, el cual retorna el porcentaje de uso de CPU en espacio usuario del compositor desde que se inició. También, para que la comparación entre compositores sea justa, se les fuerza a utilizar exclusivamente el primer núcleo de la CPU con el comando *taskset* (Kerrisk, taskset command, 2022).

### 5.1.2 Cálculo de consumo de GPU

Para medir el consumo de GPU, el benchmark utiliza el comando *intel\_gpu\_top*, el cual retorna el consumo energético promedio en watts de GPUs Intel durante un intervalo de tiempo (Canonical, 2019). En este caso, se lanza al inicio de cada iteración con un intervalo igual a la duración de esta.

### 5.1.3 Cálculo de cuadros por segundo

Para medir los cuadros por segundo que alcanza a renderizar el compositor durante una iteración, se utiliza la interfaz *wl\_callback* de Wayland. Esta interfaz tiene como objetivo evitar que los clientes actualicen su contenido a una tasa superior a la que renderiza el compositor. En este caso, el cliente actualiza las posiciones de sus superficies y envía un recurso *callback* al compositor. El compositor luego renderiza las superficies con sus posiciones actualizadas y retorna el *callback* para notificar al cliente que las ha renderizado. Una vez retornado, el cliente vuelve a actualizar las posiciones de las superficies y repite el proceso. Los cuadros por segundo que alcanza a renderizar un compositor entonces se pueden calcular contando la cantidad de *callbacks* retornados dividido por la duración de la iteración.

### 5.1.4 Ejecución

Cada iteración del benchmark tiene una duración de 10 segundos y registra el consumo de CPU, GPU y cuadros por segundo del compositor al renderizar N número de superficies. En cada benchmark, se realizan en total 20 iteraciones, variando el número de superficies desde una hasta 101 en intervalos de cinco. Para evitar que factores temporales afecten los resultados, las iteraciones para un mismo N se realizan de forma intercalada entre los compositores, con esperas intermedias de un segundo. Además, el benchmark se ejecuta un total de 11 veces para luego promediar los resultados.

### 5.1.5 Hardware de prueba

El benchmark fue ejecutado en dos máquinas cuyas características se muestran en la Tabla 8.

Tabla 8: Características de las máquinas utilizadas en el benchmark.

Máquina	CPU	GPU	RAM	OS	Resolución Monitor	Tasa de Refresco Monitor
Apple Macbook Pro A1398	Intel i7 2,2 GHz 4 Cores	Intel Iris Pro	16 GB	Debian 11 x86_64	2880x1800	60 Hz
Dell Inspiron 15R	Intel i7 2,0 GHz 4 Cores	Intel HD Graphics 3000	6 GB	Debian 11 x86_64	1366x768	60 Hz

## 5.2 Resultados

A continuación, se presentan los resultados obtenidos de los benchmarks en forma gráfica para facilitar su visualización.

### 5.2.1 Cuadros por segundo

En la Figura 53 se muestra la cantidad de cuadros por segundo a la que ambos compositores alcanzan a renderizar en la máquina Dell mostrando una a 101 superficies.

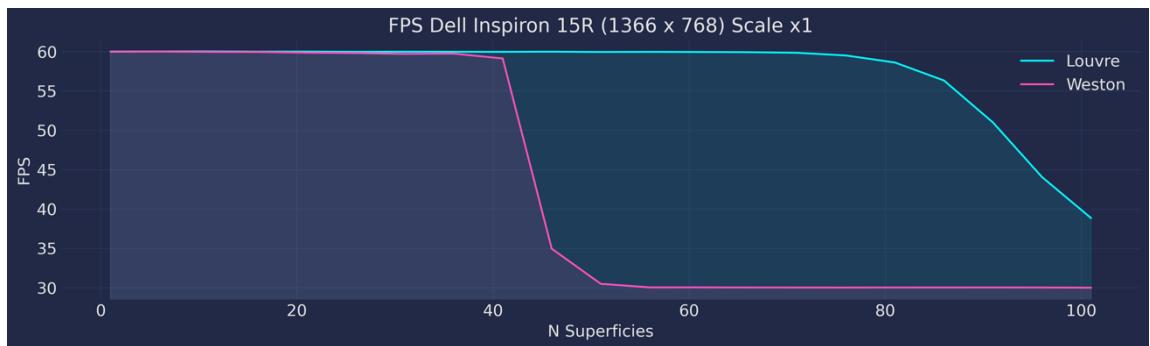


Figura 53: Cuadros por segundo de Louvre y Weston en la máquina Dell.

Weston logra renderizar hasta alrededor de 44 superficies manteniendo una tasa de refresco de 60 cuadros por segundo y luego decae abruptamente a 30.

Louvre en cambio, logra mantener una tasa de 60 cuadros por segundo hasta alrededor de 75 superficies y luego decae de forma menos abrupta a 30 cuadros por segundo.

En la máquina Apple la diferencia es aún más notoria como se aprecia en la Figura 54.



Figura 54: Cuadros por segundo de Louvre y Weston en la máquina Apple.

En este caso, se utiliza doble escala debido a la alta densidad de píxeles y resolución de la pantalla, lo que tiene un impacto negativo en el rendimiento de ambos compositores, especialmente el de Weston.

Weston experimenta una disminución brusca de la tasa de cuadros por segundo a la que alcanza a renderizar debido a la sincronización vertical y porque utiliza un único hilo (Paalanen, 2015). La sincronización vertical es el proceso a través del cual los compositores coordinan el intercambio del framebuffer que desean mostrar en pantalla con la tasa de refresco de la misma. Esto se hace para evitar que aparezca parte del framebuffer anterior en una región de la pantalla y parte del actual en otra, lo que se conoce como el efecto tearing (Wikipedia, 2022-c) y se aprecia en la Figura 55.



Figura 55: Ilustración del efecto tearing.<sup>7</sup>

---

<sup>7</sup> Imagen disponible en <https://www.techspot.com/article/2192-screen-tearing-fix-pc-gaming/>. Consultado el 29 de diciembre de 2022.

Como Weston utiliza un único hilo, tiene “tiempos muertos” mientras espera el momento exacto para hacer el intercambio. Esto significa que, en el peor de los casos, con un monitor de 60 Hz deberá esperar aproximadamente 17 milisegundos (el periodo de 60 Hz). Esto le impide procesar todas las solicitudes de clientes, eventos de entrada y renderizar el contenido para el próximo frame a tiempo. Por esta razón, comienza a omitir un frame y a renderizar a la mitad de tasa de cuadros por segundo del monitor.

Para solucionar este problema, Louvre utiliza múltiples hilos. En el hilo principal se procesan los eventos de entrada y las solicitudes de clientes, y luego cada salida renderiza utilizando su propio hilo. Esto permite al compositor seguir realizando tareas mientras una salida espera hacer el intercambio de un framebuffer y, por lo tanto, tiene más tiempo para renderizar el contenido para el próximo frame.

El procesamiento de solicitudes de clientes y el renderizado sin embargo son sectores excluyentes debido a que un cliente podría desconectarse y eliminar un buffer mientras éste está siendo renderizado en otro hilo. Para evitar este problema, se utiliza un mutex, donde la única parte no excluyente se encuentra en el intercambio de framebuffers de los hilos de renderizado, como se muestra en la Figura 56. Esto asegura que el procesamiento de solicitudes de clientes y el renderizado se realicen de manera segura y sincronizada.

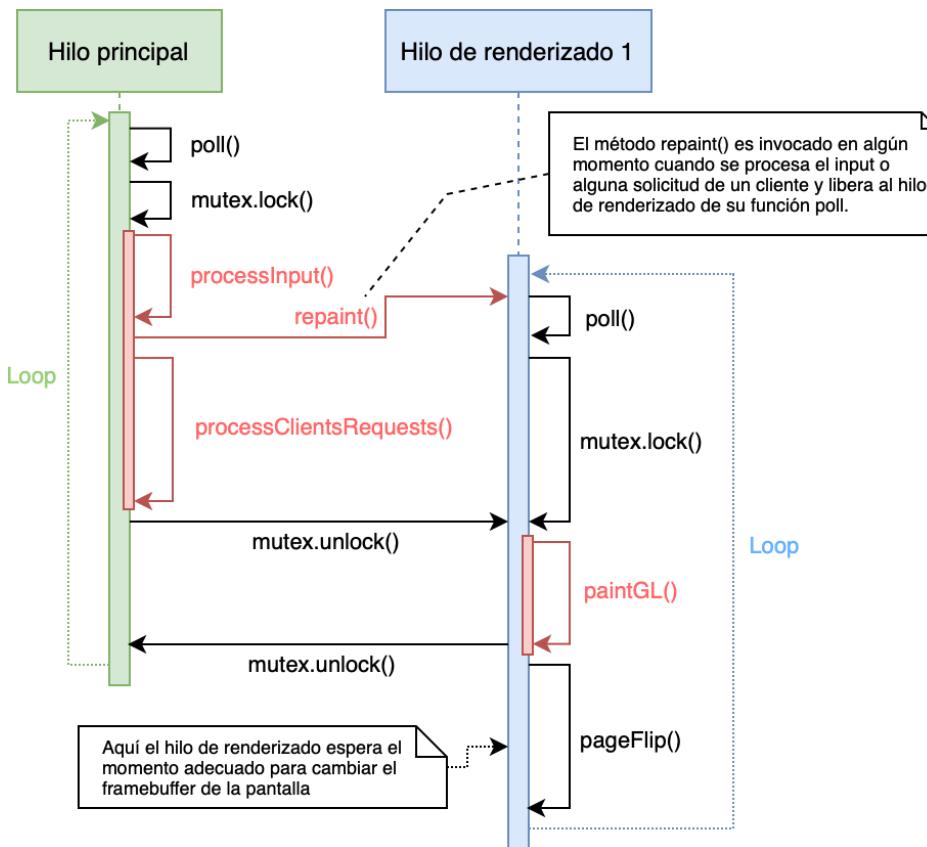


Figura 56: Sincronización de hilo principal con un hilo de renderizado.

### 5.2.2 Consumo de CPU

En la Figura 57 se muestra el porcentaje promedio de CPU que utilizan ambos compositores en espacio usuario en la computadora Dell al mostrar una a 101 superficies.

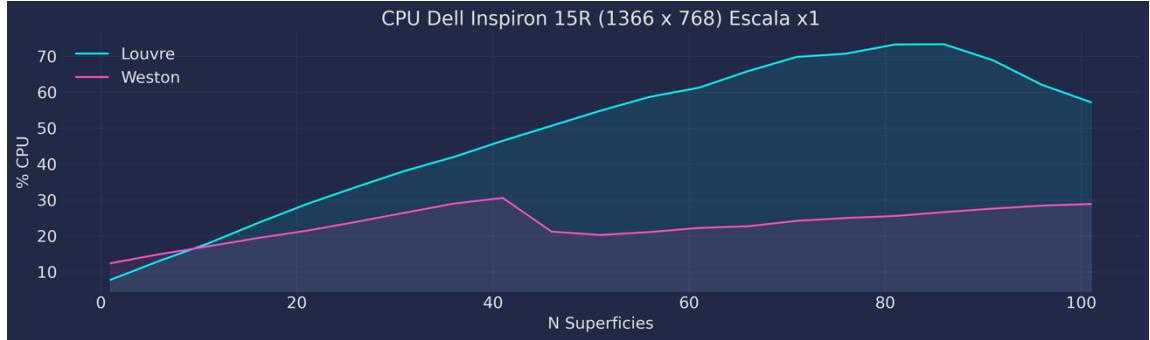


Figura 57: Consumo de CPU de Louvre y Weston en la máquina Dell.

Primero, es importante destacar que Weston comienza a disminuir su tasa de cuadros por segundo con alrededor de 44 superficies, por lo tanto, no es útil comparar el rendimiento de ambos compositores en iteraciones con un número de superficies superior a ese valor. En general, se puede observar que el consumo de CPU aumenta de manera lineal en relación con el número de superficies en ambos casos. Sin embargo, Louvre presenta una pendiente más pronunciada y, por lo tanto, tiene un rendimiento inferior. Ambos compositores utilizan el mismo algoritmo para realizar cálculos booleanos entre rectas, el cual pertenece a la librería Pixman, por lo que es posible que las diferencias en el rendimiento se deban a otros factores como por ejemplo el sistema de coordenadas que utiliza cada compositor para renderizar.

En el protocolo de Wayland se definen dos sistemas de coordenadas:

1. **Coordenadas de superficie:** En este sistema, las dimensiones del buffer de una superficie se dividen por la escala en la que se esté trabajando. Por ejemplo, las dimensiones (100x100) al trabajar en doble escala se convierten en (50x50), mientras que en escala uno se mantienen igual (100x100). Este sistema es consistente al trabajar con múltiples escalas, pero tiene la desventaja de perder granularidad con escalas mayores a uno.
2. **Coordenadas buffer:** En este sistema, las dimensiones no varían según la escala y siempre representan la mayor granularidad posible, sin embargo, tiene la desventaja de ser inconsistente al trabajar con diversas escalas.

Para evitar que los desarrolladores tengan que realizar constantes transformaciones entre los dos sistemas de coordenadas, Louvre utiliza un tercer sistema denominado coordena-

das de compositor. Este sistema transforma las dimensiones a la escala global del compositor (`LComposer::globalScale()`). La escala global del compositor es igual a la de la salida inicializada con mayor escala. Por ejemplo, si se está renderizando en dos monitores, uno con escala uno y otro con escala dos, la escala global del compositor sería dos. Al contrario que en las coordenadas de superficie, si se tienen las dimensiones (100x100) en escala uno y la escala global del compositor es dos, las dimensiones finales serían (200x200). Si se tienen las mismas dimensiones (100x100) pero con escala dos y la escala global también es dos, se mantendrían iguales (100x100).

Este sistema de coordenadas es consistente al trabajar con múltiples escalas y a la vez permite mantener la máxima granularidad posible.

Para saber el sistema de coordenadas con el que trabaja un método o propiedad de una clase, la librería añade los sufijos S, B y C que representan a los sistemas de superficie, buffer y compositor respectivamente. Por ejemplo, el método `LSurface::rolePosC()` retorna la posición de una superficie según su rol en coordenadas de compositor.

Weston renderiza utilizando coordenadas de superficie, es decir, divide las dimensiones de los buffers de las superficies por su escala (Intel Corporation et al, 2019). En cambio, Louvre utiliza coordenadas de compositor, lo que implica multiplicar las dimensiones por la escala global del compositor y dividir por la escala de la superficie. Por lo tanto, es posible que esta multiplicación adicional sea la razón del mayor consumo de CPU en Louvre puesto que estos cálculos se deben realizar un gran número de veces en cada frame.

Louvre no utiliza coordenadas de superficie para renderizar ya que puede generar artefactos indeseados en la pantalla al utilizar escalas distintas a uno. Por ejemplo, al ejecutar el cliente *weston-simple-damage* (Ekstrand & Franzke, 2022) en Weston utilizando doble escala se generan artefactos como se muestra en la Figura 58.



Figura 58: Artefactos generados en Weston debido a falta de precisión.

El cliente *weston-simple-damage*, dibuja un círculo verde el cual se mueve y rebota al colisionar con los bordes de la superficie. Como Weston utiliza coordenadas de superficie no logra repintar con precisión la recta en pantalla donde se encuentra el círculo verde en frame anteriores, lo que deja un rastro de líneas verdes. Esto no ocurre en Louvre gracias a que el sistema de coordenadas de compositor no pierde granularidad.

En cuanto a la máquina Apple, Weston disminuye su tasa de refresco a 30 cuadros por segundo con una cantidad aproximada de 10 superficies. Por lo tanto, nuevamente, conviene comparar el rendimiento antes de ese punto, donde Louvre a simple vista pareciera tener mejor rendimiento, sin embargo, su pendiente es claramente más inclinada, como se muestra en la Figura 59.



Figura 59: Consumo de CPU de Louvre y Weston en la máquina Apple.

### 5.2.3 Consumo de GPU

En cuanto al consumo de GPU en la máquina Dell, se puede apreciar que ambos compositores poseen un rendimiento similar (ver Figura 60). Louvre pareciera consumir un poco más de GPU, sin embargo, si analizamos la Figura 53 podemos intuir que se debe a que Weston está renderizando una menor cantidad de cuadros por segundo.

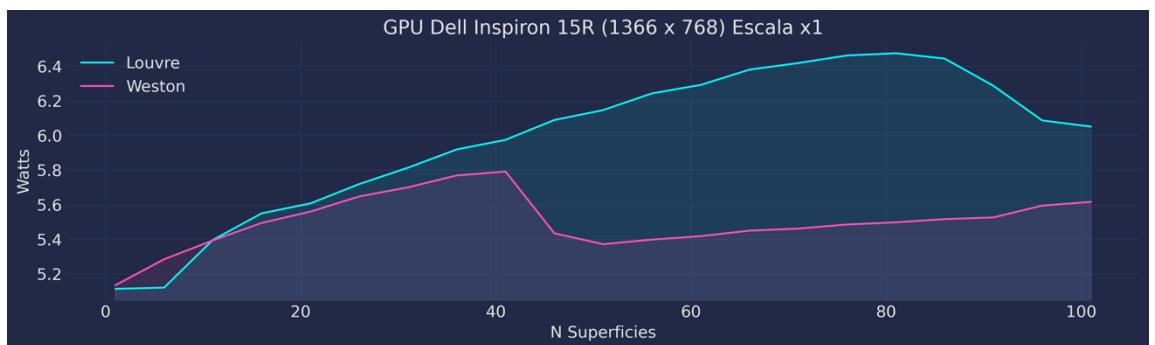


Figura 60: Consumo de GPU de Louvre y Weston en la máquina Dell.

Respecto a la máquina Apple (ver Figura 61), también se puede observar que poseen un rendimiento similar antes de que Weston disminuya su tasa de refresco a 30 cuadros por

segundo. Luego con un número de superficies superior a 40 Weston eleva considerablemente su consumo de GPU aún cuando está renderizando a 30 cuadros por segundo, cuyo motivo no se pudo determinar.



Figura 61: Consumo de GPU de Louvre y Weston en la máquina Apple.

#### 5.2.4 Conclusiones

Los resultados indican que el uso de múltiples hilos es una estrategia efectiva para aumentar la tasa de cuadros por segundo en un compositor, especialmente al activar la sincronización vertical en monitores de alta resolución, ya que en este caso permite a Louvre renderizar un número considerablemente mayor de superficies que Weston manteniendo una alta tasa de refresco.

En cuanto al consumo de CPU, Louvre presentó un consumo más elevado que Weston para un número de superficies mayor a 10, pero realiza una renderización más precisa que evita la aparición de artefactos en monitores que requieren una escala diferente a 1.

Finalmente, ambos compositores demostraron un rendimiento similar en cuanto al consumo de GPU, aunque Weston tuvo problemas al renderizar un gran número de superficies en monitores de alta resolución.

## 6. USABILIDAD

Uno de los objetivos principales de este trabajo es reducir el tiempo y esfuerzo necesarios para crear compositores Wayland. Para evaluar la usabilidad de la API generada a partir del diseño propuesto y determinar qué aspectos pueden mejorarse, se solicitó a dos informáticos que desarrollen su propio compositor utilizando la librería con la ayuda del tutorial y documentación de la API generada con Doxygen, disponible en el sitio GitHub Pages del repositorio del proyecto (<https://cuarzosoftware.github.io/Louvre>), y que respondan el formulario propuesto por Steve Clarke (Anexo C) mediante Google Forms.

### 6.1 Resultados

La evaluación de la librería realizada por los usuarios permitió identificar algunos problemas y áreas a mejorar, así como también confirmar aspectos positivos. Los detalles completos de las respuestas de los usuarios se pueden encontrar en el Anexo E.

Durante la evaluación uno de los usuarios identificó que el backend DRM no funciona en entornos Linux que no cuentan con el servicio *seatd* activo (Levinsen, s.f.), este servicio es necesario para que el compositor obtenga permisos de lectura y escritura de los dispositivos de entrada y salida sin requerir privilegios elevados. Aunque este problema no es directamente causado por la librería, es importante mencionarlo en la documentación para evitar confusiones o problemas para los usuarios.

Otro problema detectado fue que ambos usuarios no pudieron detener el proceso del compositor utilizando la combinación de teclas "Ctrl+Shift+Esc", que en la implementación predeterminada de LKeyboard::keyEvent() invoca al método exit() para cerrar el compositor. En el caso del primer usuario, el compositor se congela forzándolo a reiniciar su computadora. Sin embargo, utilizando el comando *pkill* lograba finalizar el proceso correctamente. Al investigar este problema se determinó que se debía a los procesos hijos del compositor, como el lanzamiento de *weston-terminal* al presionar "F1", los cuales se congelan y no permiten que el compositor termine. Este problema pudo solucionarse llamando al método kill() (Kerrisk, kill, 2022), el cual cierra todos los procesos hijos del compositor permitiendo llamar posteriormente a la función exit() de manera segura.

En el caso del segundo usuario, se pudo determinar que la combinación de teclas "Ctrl+Shift+Esc" no lograba detener el proceso del compositor debido a que al utilizar el backend de input X11, la ventana del compositor en ocasiones perdía el foco del teclado, lo cual impedía que reciba eventos del mismo.

Respecto a los aspectos a mejorar, uno de los usuarios sugirió añadir diagramas en la documentación que permitan visualizar de manera sencilla la conexión entre las distintas clases en el compositor. Además, ambos usuarios expresaron confusión en relación al método initialized() de la clase LCompositor, uno de los usuarios pensó que retornaba un valor booleano indicando si el compositor está inicializado, mientras que el otro no estaba seguro de su función específica. Por lo tanto, se podría considerar cambiar la nomenclatura a algo como initializedEvent() para aclarar su función. Otro aspecto que generó confusión

fue la utilización de los métodos repaint() y paintGL() de la clase LOutput, por lo que se requiere mejorar la documentación y explicación para su uso adecuado.

También se identificaron algunos errores de redacción y detalles menores a mejorar en el tutorial.

En cuanto a los aspectos positivos, ambos usuarios expresaron que la experiencia con la librería fue satisfactoria, destacando que se requirió escribir una cantidad adecuada código para realizar sus tareas, que pudieron validar frecuentemente su progreso y cambios, y que no tuvieron que comprender un gran número de aspectos de la API para comenzar a desarrollar. En cuanto al aprendizaje, el tutorial pareció ser la opción preferida, con revisiones ocasionales a la documentación de la API y pruebas enfocadas especialmente en los métodos relacionados al renderizado.

En una entrevista realizada a uno de los usuarios (antes de solicitarle probar Louvre) cuyo resumen y transcripción se encuentra en el Anexo F, menciona que intentó desarrollar previamente un compositor utilizando wl\_roots pero se vio obstaculizado por la falta de documentación. A pesar de ello, logró comprender algunos conceptos revisando y compilando un pequeño compositor de ejemplo (Tinywl), pero aún así, menciona que tenía muchas líneas de código y que no supo cómo continuar, tampoco recuerda si logró renderizar una aplicación en pantalla. A pesar de que se trata de la opinión de un solo usuario, se puede intuir que crear un compositor con Louvre puede ser más sencillo que con wl\_roots ya que el usuario menciona que logró comprender la API Louvre, renderizar e interactuar con aplicaciones Wayland y pudo modificar la lógica de renderizado por defecto.

## 7. CONCLUSIONES

Del trabajo se puede concluir que el patrón de diseño Factory, con una arquitectura de dos capas para procesar las solicitudes de los clientes de Wayland, que incluye un preprocesamiento en la primera y la utilización de métodos virtuales en la segunda con una implementación predeterminada, es un enfoque eficaz para facilitar el desarrollo de compositores Wayland ya que permitió disminuir en un 79% la probabilidad de que el desarrollador cometiera errores de protocolos (cumpliendo con la meta propuesta para el segundo objetivo específico) y además evita que este deba conocer de antemano e implementar todas las reglas de los protocolos para poder compilar y ver un compositor funcional, disminuyendo considerablemente la curva de aprendizaje al aumentar la evaluación progresiva.

Esta técnica probablemente pueda ser aplicada a cualquier librería que requiera una implementación de múltiples interfaces, con solicitudes interdependientes para lograr un resultado mínimamente funcional.

En cuanto al rendimiento, se pudo concluir que el uso de múltiples hilos es una buena estrategia para aumentar la tasa de cuadros por segundo a la que alcanza a renderizar un compositor al activar la sincronización vertical. Al utilizar varios hilos, el compositor puede seguir realizando tareas mientras espera el intercambio del framebuffer en pantalla, lo que le otorga mayor tiempo para renderizar el próximo frame. Esto se pudo evidenciar al comparar el rendimiento con Weston, el cual utiliza un único hilo, donde se demostró que Louvre es capaz de mantener una tasa de cuadros por segundo elevada al renderizar el doble número de superficies que Weston en un monitor con escala 1 y seis veces más en un monitor de alta definición que utiliza doble escala, superando la meta propuesta para este indicador.

Respecto al consumo de CPU, Louvre presenta un consumo más elevado que Weston, sin embargo, permite renderizar con mayor precisión lo que evita que se generen artefactos en monitores que deben utilizar una escala distinta a uno.

En cuanto al consumo de GPU, ambos compositores demostraron tener un rendimiento similar, aunque Weston parecía tener problemas al renderizar un número de superficies elevados en monitores con alta resolución.

En cuanto al objetivo específico relacionado al rendimiento, se cumplió la meta para los indicadores de cuadros por segundo y consumo de GPU, pero no así para el consumo de CPU.

La librería en cuestión puede ser una excelente herramienta para aquellos interesados en aprender sobre Wayland y en desarrollar sus propios compositores, lo que podría aumentar la innovación en este ámbito y mejorar la experiencia de usuario en Linux. Además, puede ser de utilidad para individuos o empresas que deseen desarrollar interfaces gráficas para sistemas embebidos, como en la industria automotriz, de electrodomésticos o de dispositivos móviles.

## 7.1 Trabajos futuros

Existe una gran cantidad de protocolos presentes en la comunidad que aún se encuentran en fase de desarrollo. Estos protocolos ofrecen una amplia gama de funcionalidades, tales como el compartimiento de pantalla, el acceso global a atajos del teclado, y mecanismos alternativos para intercambiar buffers gráficos. Es por esto que se planea incorporarlos a la librería en cuanto sean considerados estables.

También se planea mejorar la modularidad de la librería. Para ello, se incorporarán módulos encargados de renderizar que permitan utilizar APIs de renderizado adicionales a OpenGL ES 2.0, como Vulkan o Pixman. Esto mejorará la capacidad de la librería para proporcionar un rendimiento óptimo en diferentes situaciones y configuraciones en la que se ejecute un compositor.

En cuanto a la compatibilidad con aplicaciones en Linux, se creará un módulo que facilitará la integración con XWayland, un servidor X11 que simula ser un cliente Wayland. Este módulo permitirá mostrar aplicaciones para X11, lo que mejorará la experiencia de los usuarios al poder seguir utilizando aplicaciones que no hayan sido portadas aún a Wayland.

Finalmente, se tomará en cuenta la retroalimentación dada por los usuarios que probaron la librería para mejorar la documentación y algunos aspectos de la API.

## REFERENCIAS

- Reddy, M. (2011). *API Design for C++*. Burlington, USA: Elsevier.
- Henning, M. (Mayo de 2009). API Design Matters. *Commun. ACM*, 52(5), 46–56.
- Høgsberg, K. (2012). *Chapter 3. Wayland Architecture*. Recuperado el 20 de diciembre de 2022, de [https://wayland.freedesktop.org/docs/html/ch03.html#sect-Wayland-Architecture-wayland\\_architecture](https://wayland.freedesktop.org/docs/html/ch03.html#sect-Wayland-Architecture-wayland_architecture)
- Høgsberg, K. (2012). *Chapter 4. Wayland Protocol and Model of Operation*. Recuperado el 20 de diciembre de 2022, de <https://wayland.freedesktop.org/docs/html/ch04.html#sect-Protocol-Versioning>
- Høgsberg, K. (2022). *Wayland Protocol*. Recuperado el 20 de diciembre de 2022, de <https://wayland.app/protocols/wayland>
- Packard, K., & Johnson, D. (3 de julio de 2007). *Composite Extension*. Recuperado el 20 de diciembre de 2022, de <https://cgit.freedesktop.org/xorg/proto/compositeproto/tree/compositeproto.txt>
- Wikipedia. (27 de octubre de 2022-a). *Servidor gráfico*. Recuperado el 20 de diciembre de 2022, de [https://es.wikipedia.org/wiki/Servidor\\_gráfico](https://es.wikipedia.org/wiki/Servidor_gráfico)
- Wikipedia. (22 de agosto de 2022-b). *Gestor de composición de ventanas*. Recuperado el 20 de diciembre de 2022, de [https://es.wikipedia.org/wiki/Gestor\\_de\\_composición\\_de\\_ventanas](https://es.wikipedia.org/wiki/Gestor_de_composición_de_ventanas)
- The Qt Company. (2022-a). *Qt Wayland Composer C++ Classes*. Recuperado el 20 de diciembre de 2022, de [qt.io: https://doc.qt.io/qt-6/qtwaylandcompositor-module.html](https://doc.qt.io/qt-6/qtwaylandcompositor-module.html)
- The Qt Company. (2022-b). *Signals & Slots*. Recuperado el 20 de diciembre de 2022, de <https://doc.qt.io/qt-6/signalsandslots.html>
- The Qt Company. (2022-c). *QOpenGLWindow Class*. Recuperado el 20 de diciembre de 2022, de <https://doc.qt.io/qt-6/qopenglwindow.html>
- wlroots. (2022). *wlroots*. Recuperado el 20 de diciembre de 2022, de <https://gitlab.freedesktop.org/wlroots/wlroots>
- GamingOnLinux. (2023). *User stats*. Recuperado el 4 de enero de 2023, de <https://www.gamingonlinux.com/users/statistics/#SessionType-top>
- Slant. (17 de diciembre de 2022). *Best Wayland compositors*. Recuperado el 4 de enero de 2023, de <https://www.slant.co/topics/11023/~wayland-compositors>
- Wayland Project. (2022). *Weston*. Recuperado el 20 de diciembre de 2022, de <https://github.com/wayland-project/weston>
- The Libinput Team. (2019). *Libinput*. Recuperado el 20 de diciembre de 2022, de <https://wayland.freedesktop.org/libinput/doc/latest/>
- DeVault, D. (2022). *XDG Shell Basics*. Recuperado el 20 de diciembre de 2022, de Wayland Book: <https://wayland-book.com/xdg-shell-basics.html>
- Ser, S. (2018). *XDG decoration*. Recuperado el 20 de diciembre de 2022, de Simon Ser
- Wayland Explorer. (2022). *Pointer gestures*. Recuperado el 20 de diciembre de 2022, de <https://wayland.app/protocols/pointer-gestures-unstable-v1>

- Freedesktop Organization. (2012-a). *Wayland*. Recuperado el 20 de diciembre de 2022, de <https://wayland.freedesktop.org>
- Freedesktop Organization. (2012-b). *Appendix A. Wayland Protocol Specification*. Recuperado el 20 de diciembre de 2022, de <https://wayland.freedesktop.org/docs/html/apa.html>
- Tomazic, B. (2020). *Inter-Process Texture Sharing with DMA-BUF*. Recuperado el 20 de diciembre de 2022, de <https://blaztinn.gitlab.io/post/dmabuf-texture-sharing/>
- Ser, S. (2019). *Introduction to damage tracking*. Recuperado el 20 de diciembre de 2022, de <https://emersion.fr/blog/2019/intro-to-damage-tracking/>
- Canonical. (2019). *intel\_gpu\_top*. Recuperado el 20 de diciembre de 2022, de [https://manpages.ubuntu.com/manpages/trusty/man1/intel\\_gpu\\_top.1.html](https://manpages.ubuntu.com/manpages/trusty/man1/intel_gpu_top.1.html)
- Paalanen, P. (2015). *Weston repaint scheduling*. Recuperado el 20 de diciembre de 2022, de <https://ppaalanan.blogspot.com/2015/02/weston-repaint-scheduling.html>
- Wikipedia. (2022-c). *Tearing*. Recuperado el 20 de diciembre de 2022, de <https://es.wikipedia.org/wiki/Tearing>
- Intel Corporation et al. (2019). *Weston GL Renderer*. Recuperado el 20 de diciembre de 2022, de <https://coral.googlesource.com/weston-imx/+/refs/tags/3.0.0-4/libweston/gl-renderer.c>
- Clarke, S. (2006). *Describing and Measuring API Usability with the Cognitive Dimensions*. Recuperado el 15 agosto de 2022, de [https://www.researchgate.net/publication/200086113\\_Describing\\_and\\_Measuring\\_API\\_Usability\\_with\\_the\\_Cognitive\\_Dimensions](https://www.researchgate.net/publication/200086113_Describing_and_Measuring_API_Usability_with_the_Cognitive_Dimensions)
- Zghidi, A., Hammouda, I., & Hnich, B. (2018). Towards a Formal API Assessment. *Association for Computing Machinery*(2), 398–399.
- Levinsen, K. (s.f.). *Libseat*. Recuperado el 20 de diciembre de 2022, de <https://sr.ht/~kennylevinsen/seatd/>
- Linux Kernel Development Community. (s.f-b). *Kernel Mode Setting (KMS)*. Recuperado el 20 de diciembre de 2022, de <https://www.kernel.org/doc/html/v4.13/gpu/drm-kms.html>
- Linux Kernel Development Community. (s.f-a). *DRM Userland Interfaces*. Recuperado el 20 de diciembre de 2022, de <https://docs.kernel.org/gpu/drm-uapi.html>
- Freedesktop Organization. (2022-c). *Pixman*. Recuperado el 20 de diciembre de 2022, de <https://github.com/freedesktop/pixman>
- Kerrisk, M. (2022). *ps command*. Recuperado el 20 de diciembre de 2022, de <https://man7.org/linux/man-pages/man1/ps.1.html>
- Kerrisk, M. (2022). *taskset command*. Recuperado el 20 de diciembre de 2022, de <https://man7.org/linux/man-pages/man1/taskset.1.html>
- Ekstrand, J., & Franzke, B. (2022). *simple-damage.c*. Recuperado el 20 de diciembre de 2022, de <https://fossies.org/linux/weston/clients/simple-damage.c>
- Kerrisk, M. (2022). *kill*. Recuperado el 6 de enero de 2023, de <https://man7.org/linux/man-pages/man2/kill.2.html>

## ANEXOS

### A. Código de ejemplo de patrón Observer en QtWayland.

```
class Compositor :public QWaylandCompositor
{
    Q_OBJECT
public:
    Compositor()
    {
        m_xdgShell = new QWaylandXdgShell(this);

        // Se suscribe a la señal QWaylandXdgShell::xdgSurfaceCreated
        connect(m_xdgShell, &QWaylandXdgShell::xdgSurfaceCreated, this, &Compositor::onXdgSurfaceCreated);
    }

public slots:
    void onXdgSurfaceCreated(QWaylandXdgSurface *xdgSurface)
    {
        // Maneja la creación de un recurso xdg_surface
    }

private:
    QWaylandXdgShell *m_xdgShell = nullptr;
}
```

### B. Código de ejemplo de patrón Observer en wl\_roots.

```
static void server_new_xdg_surface(struct wl_listener *listener, void *data)
{
    // Maneja la creación de un recurso xdg_surface
}

int main()
{
    ...

    server.xdg_shell = wlr_xdg_shell_create(server.wl_display);
    server.new_xdg_surface.notify = server_new_xdg_surface;

    // Se suscribe a la señal new_surface de xdg_shell
    wl_signal_add(&server.xdg_shell->events.new_surface, &server.new_xdg_surface);

    ...
}
```

### C. Adaptación de formulario propuesto por Steve Clarke.

Gracias por tomarse el tiempo para completar este cuestionario. Después de haber pasado un tiempo trabajando con Louvre, se encuentra en una excelente posición para reflexionar sobre sus experiencias y brindarnos sus comentarios. Nos gustaría que responda las siguientes preguntas de la manera más completa posible. Para cada pregunta, piense en las diferentes tareas en las que trabajó y los aspectos particulares de la API que hicieron que estas fueran fáciles o difíciles.

1. Proporcione una breve descripción de las tareas en las que trabajó.
2. Proporcione una descripción de cualquier problema que experimentó con la API mientras trabajaba en las tareas.
3. ¿Cómo calificaría su experiencia general con la API?

### Abstracción

1. ¿Pudo implementar la funcionalidad central de una tarea dada usando solo una clase o utilizó más de una? ¿Esperaba usar solo una clase para implementar la funcionalidad central para esta tarea o esperaba usar más de una?
2. ¿Cómo describiría sus experiencias respecto a los tipos de clases con las que tuvo que trabajar? ¿Por qué?
  - ¿Estaban bien?
  - ¿Eran de demasiado bajo nivel?
  - ¿Eran de demasiado alto nivel?

### Estilo de aprendizaje

1. ¿Cómo aprendió a usar la API? Puede marcar más de una opción:
  - Escribiendo un par de líneas de código para tratar hacer que algo funcione y luego intentando comprenderlo a partir de eso.
  - Copiando el código de muestra proporcionado con la API o tutorial.
  - Leyendo una descripción general de alto nivel de la API primero, escribiendo código luego de tener una idea sobre la arquitectura de la API y cómo las clases se relacionan entre sí.

¿Tuvo éxito utilizando este enfoque para aprender a usar la API?

¿Cómo describiría su experiencia respecto al aprendizaje de la API? ¿Estuvo bien, había mucho que leer o aprender, o no había suficiente información? ¿Por qué?

### Trabajo por unidad de paso

1. ¿Cómo describiría la cantidad de código que tuvo que escribir para cada tarea? Si tuvo que dividir la tarea general en sub-tareas más pequeñas, ¿Cómo describiría la cantidad de código que tuvo que escribir para cada una de esas sub-tareas?

### **Evaluación progresiva**

1. ¿Qué tan fácil fue detenerse en medio de la tarea en la que estaba trabajando y revisar su trabajo hasta el momento? ¿Pudo hacer esto cada vez que quería o necesitaba? ¿Si no, porque no?
2. ¿Pudo probar versiones parcialmente completadas del producto? ¿Si no, porque no?
3. ¿Cómo describiría sus experiencias al evaluar su progreso trabajando con esta API? ¿Pudo verificar el progreso cuando lo necesitaba o tuvo que trabajar más de lo esperado para poder verificar el progreso?

### **Compromiso prematuro**

1. ¿Pudo realizar el trabajo en el orden que deseaba, o el sistema lo obligó a pensar en el futuro y tomar ciertas decisiones primero? Si es así, ¿Qué decisiones necesitó tomar por adelantado? ¿Era evidente que necesitaba tomar estas decisiones o lo aprendió a través de prueba y error? ¿Qué tipo de problemas pudo causar esto en su trabajo?

### **Penetrabilidad**

1. ¿Cuántos detalles de la API tuvo que comprender para poder utilizarla con éxito? ¿Cuántos de esos detalles estaban expuestos en la API?
2. ¿Qué tan fácil fue ver o encontrar los detalles de la API mientras la usaba? ¿Por qué? ¿Qué tipo de cosas son más difíciles de ver o encontrar?

### **Elaboración de la API**

- ¿Pudo usar la API exactamente "tal cual" o tuvo que crear nuevos objetos derivando de clases en la API?
- ¿Hasta qué punto pudo ampliar la API para satisfacer sus necesidades?
- ¿Cómo describiría sus experiencias respecto a la extensibilidad de la API? ¿Estuvo bien, no hubo suficientes oportunidades para extenderla o se vio obligado a hacerlo en contra de su voluntad?

### **Viscosidad de la API**

1. Cuando necesitó realizar cambios en su código escrito mediante la API, ¿Qué tan fácil fué realizar el cambio? ¿Por qué?

2. ¿Hubo cambios particulares que fueron más difíciles o especialmente difíciles de hacer? ¿Cuales?

### **Consistencia**

1. ¿Hay lugares de la API que parecen similares, pero resultan ser diferentes? ¿Cuales?

### **Expresividad**

1. Al leer código que hace uso de la API, ¿Es fácil saber qué hace cada sección del código? ¿Por qué?
2. ¿Hay partes que sean particularmente difíciles de interpretar? ¿Cuáles?
3. Al escribir el código, ¿Es fácil saber qué clases y métodos de la API usar?

### **Correspondencia del dominio**

1. ¿Qué tan relacionados están las clases y los métodos expuestos por la API con los objetos conceptuales que piensa que representan? ¿Por qué?
2. ¿Las clases y métodos expuestos por la API se asignan a objetos y comandos que tienen sentido en el dominio particular en el que trabajó?
3. Cuando utiliza la API, ¿Es fácil mapear las ideas en su cabeza al código? ¿Por qué?

### **Preguntas finales**

1. ¿Utilizó la API de formas inusuales o formas que los implementadores podrían no haber previsto? Si es así, ¿Puede dar algunos ejemplos?
2. ¿Hubo algo que quiso hacer con la API que no pudo lograr? ¿Qué cosa?
3. Después de completar esta encuesta, ¿Puede imaginar formas obvias de mejorar el diseño de la API? ¿Qué formas? ¿Serían mejoras específicamente para sus propios requisitos?
4. Si tuviéramos preguntas de seguimiento basadas en algunas de sus respuestas, ¿Podríamos comunicarnos con usted para obtener detalles adicionales? En caso afirmativo, ¿Cuál es la mejor manera de contactarlo (por ejemplo, correo electrónico, teléfono) y cuál es su información de contacto?

## D. Instrucciones de compilación de Louvre (Capítulo 2 del tutorial).

En la sección [Descargas](#) puede encontrar la última versión de **Louvre** pre-compilada para distribuciones basadas en **Debian** o **RedHat**.

Si opta por utilizar esta opción puede omitir la compilación manual y saltar directamente a la sección [Ejemplos](#).

### Compilación Manual

**Louvre** depende de las siguientes librerías:

- **Wayland Server** >= 1.16
- **EGL** >= 1.5.0
- **GLES** 2.0 >= 13.0.6
- **DRM** >= 2.4.85
- **GBM** >= 22.2.0
- **Eudev** >= 1.5.6
- **Libinput** >= 1.6.3
- **Xcursor** >= 1.1.15
- **XKB Common** >= 0.7.1
- **X11 Fixes** >= 6.0.0
- **XRandr** >= 1.5.1
- **Pixman** >= 0.40.0
- **Libseat** >= 0.6.0

Y de la herramienta [Meson](#) para su compilación.

#### Debian (Ubuntu, Mint, etc)

Si su distribución está basada en Debian puede instalar todas las dependencias con los siguientes comandos:

```
$ sudo apt update  
$ sudo apt upgrade  
$ sudo apt install build-essential libegl-dev mesa-common-dev libgles2-mesa-dev libdrm-dev libgbm-dev libevdev-dev libi
```

Y la herramienta Meson con:

```
$ sudo apt install meson
```

También se recomienda instalar [weston-terminal](#), la cual es compatible con Wayland y se utilizará en el transcurso del tutorial.

```
$ sudo apt install weston
```

#### Red Hat (Fedora, CentOS, etc)

Si su distribución está basada en Red Hat puede instalar todas las dependencias con los siguientes comandos:

```
$ sudo dnf update  
$ sudo dnf install @development-tools make automake gcc gcc-c++ kernel-devel libwayland-server wayland-devel libinput-d
```

Y la herramienta Meson con:

```
$ sudo dnf install meson
```

También se recomienda instalar [weston-terminal](#), la cual es compatible con Wayland y se utilizará en el transcurso del tutorial.

```
$ sudo dnf install weston
```

### Compilar Louvre

Ejecute los siguientes comandos para compilar e instalar **Louvre**:

```
$ git clone https://github.com/CuarzoSoftware/Louvre.git  
$ cd Louvre/src  
$ meson setup build --buildtype=release  
$ cd build  
$ meson compile  
$ sudo meson install
```

Esto instalará la librería en:

```
/usr/lib
```

Las cabeceras en:

```
/usr/include/Louvre
```

Los backends en:

```
/usr/etc/Louvre/backends
```

Y ejemplos en:

```
/usr/bin
```

## E. Respuestas del formulario de usabilidad de la API.

Proporcione una descripción de cualquier problema que experimentó con la API mientras trabajaba en las tareas.

1. En mi equipo, al utilizar Ctrl + shift + Esc para terminar el proceso del compositor, se quedaba "pegado" todo el equipo, necesitando un reinicio manual. Luego de descubrir este problema, en reemplazo de la combinación mencionada, se utilizó el comando de linux `pkill` para terminar el proceso, lo cual permitió utilizar la librería sin mayores inconvenientes. 2. En el tutorial, en el Capítulo 5 (Fábrica de Recursos) se instruye la creación de una clase mediante el wizard de creación de clases de QtCreator. Se indica al lector que presione "Next y Finish", lo cual deja a los archivos de código fuente con los nombres mycompositor.h y mycompositor.cpp (todo en minúsculas). Sin embargo, en el resto del tutorial se asume que los nombres de los archivos son MyCompositor.h y MyCompositor.cpp (capitalizado). Esto generó un problema al intentar importar o incluir archivos que no existían con el nombre que estaba descrito en el tutorial. El problema fue menor, ya que luego de unos 2 minutos, me di cuenta del problema y lo solucioné. 3. Un problema muy menor: Se indica al usuario a "marcar las opciones QtCreator y qmake durante la instalación", sin embargo, cuando instalé QtCreator, creo que no aparecía la opción qmake. Seguí con la instalación y no hubo ningún problema.

1 respuesta

Utilizando el backend por defecto (DRM y libinput) no logre hacerla funcionar, pero al cambiar a x11, funcionaba bien, excepto cuando a veces intentaba cerrar el compositor con ctrl-shift-esc, a veces se pegaba. Con X11 pude terminar el tutorial sin problemas.

1 respuesta

¿Cómo calificaría su experiencia general con la API?

Buenísima. He utilizado otras librerías de wayland antes (wlroots) y encontré que era muy difícil para un novato trabajar con ellas. Se asumía la existencia de una serie de conocimientos previos avanzados, y no existía un tutorial oficial para iniciar. El conocimiento estaba esparcido por distintos lados y era difícil saber por dónde comenzar. En cambio, con Louvre, es muy fácil comenzar y el tutorial es amigable con personas novatas en Wayland. Además, es más fácil experimentar, ya que en menos de 3 horas se puede estar modificando código relativo a \*rendering\*, sin la necesidad de escribir cientos o miles de líneas de código como en otras librerías.

---

1 respuesta

Muy buena

---

1 respuesta

¿Pudo implementar la funcionalidad central de una tarea dada usando solo una clase o utilizó más de una? ¿Esperaba usar solo una clase para implementar la funcionalidad central para esta tarea o esperaba usar más de una?

Utilicé solo las clases del tutorial. No desarrollé funcionalidades extra aparte de las expuestas en el tutorial. En el tutorial, a veces se utilizan varias clases para lograr un objetivo, pero se explica para qué es cada clase, por lo que la separación de responsabilidades entre clases es justificada y entendible para el usuario.

---

1 respuesta

por lo general si, a veces se usaban clases como LPoint y otras menores, pero nada que generara una dependencia extraña. Lo que me parecio curioso y que no habia visto antes, es que la implementacion del metodo MyCompositor::createOutputManagerRequest(), se hiciera dentro del archivo MyOutputManager.cpp en vez de MyCompositor.cpp.

---

1 respuesta

¿Cómo describiría su experiencia respecto a los tipos de clases con las que tuvo que trabajar?

Ocultar opciones ^

- Estaban bien.
- Eran de demasiado bajo nivel.
- Eran de demasiado alto nivel.

Estaban bien.

---

2 respuestas

¿Cómo aprendió a usar la API? Puede marcar más de una opción.

Ocultar opciones ^

- Escribiendo un par de líneas de código para tratar hacer que algo funcione y luego intentando comprenderlo a partir de eso.
- Copiando el código de muestra proporcionado con la API o tutorial.
- Leyendo una descripción general de alto nivel de la API primero, escribiendo código luego de tener una idea sobre la arquitectura de la API y cómo las clases se relacionan entre sí.

- Escribiendo un par de líneas de código para tratar hacer que algo funcione y luego intentando comprenderlo a partir de eso.

- Copiando el código de muestra proporcionado con la API o tutorial.

---

1 respuesta

- Copiando el código de muestra proporcionado con la API o tutorial.

- Leyendo una descripción general de alto nivel de la API primero, escribiendo código luego de tener una idea sobre la arquitectura de la API y cómo las clases se relacionan entre sí.

---

1 respuesta

¿Tuvo éxito utilizando este enfoque para aprender a usar la API?

Sí

1 respuesta

Sí

1 respuesta

¿Cómo describiría su experiencia respecto al aprendizaje de la API? ¿Estuvo bien, había mucho que leer o aprender, o no había suficiente información? ¿Por qué?

Estuvo muy bien, al menos para mi nivel de conocimiento. El nivel de información, en general, era el apropiado. Solo una o dos veces quedé con alguna duda conceptual, ahora recuerdo esta: 1. "el método virtual Louvre::LComposer::initialized() que notifica la correcta inicialización del compositor" ¿A quién la notifica? ¿Es una especie de callback? O sea que se ejecutará automáticamente este método cuando el compositor esté inicializado, y el usuario de Louvre debe llenar esta función con el comportamiento que desea que ocurra en ese caso. Eso es lo que entiendo de la función, pero frasearlo como que "la función notifica la correcta inicialización del compositor" hizo que me costara un poco entenderlo.

1 respuesta

En general se entienden bien como funcionan las partes que componen la API y que están presente en el tutorial. Creo que es gracias a que son llamadas de alto nivel. Respecto a la información, se agradece la explicación de conceptos antes de entrar al código, sin embargo, creo que hay falta algunos dibujos sobre como se conecta el compositor con el resto de cosas, que rol tiene usando como ejemplo algo ya conocido por el usuario. También hay un concepto que no me quedó muy claro y es el de Rol, que es un Rol? cuánto abarca?, a pesar de que se menciona en varias partes.

1 respuesta

¿Cómo describiría la cantidad de código que tuvo que escribir para cada tarea?

Ocultar opciones ^

- Justa.
- Demasiado código.
- Poco código.

Justa.

1 respuesta

Poco código.

1 respuesta

Si tuvo que dividir la tarea general en sub-tareas más pequeñas, ¿Cómo describiría la cantidad de código que tuvo que escribir para cada una de esas sub-tareas?

Fue poco código por tarea, pero al menos prefiero que sea así, ya que no tengo conocimientos avanzados en Wayland.

1 respuesta

Generalmente poco

1 respuesta

¿Qué tan fácil fue detenerse en medio de la tarea en la que estaba trabajando y revisar su trabajo hasta el momento? ¿Pudo hacer esto cada vez que quería o necesitaba? ¿Si no, por qué no?

Fue facil, en general redefinir algun metodo no hacia q todo se rompiera, eso si, comparado al resto de trabajo, algunos como LCompositor::initialized o LOutput::paintGL() requerian mas trabajo para volver a funcionar correctamente (mas trabajo en proporcion al resto de modificaciones del tutorial)

1 respuesta

Si, pude hacerlo cada vez, porque el tutorial y el código de ejemplo estaban bien ordenados.

1 respuesta

¿Pudo probar versiones parcialmente completadas del producto? ¿Si no, por qué no?

Si, porque el hecho de que las funciones tengan implementaciones por defecto permiten ir probando versiones funcionales del compositor prácticamente desde el primer minuto.

1 respuesta

si

1 respuesta

¿Cómo describiría sus experiencias al evaluar su progreso trabajando con esta API? ¿Pudo verificar el progreso cuando lo necesitaba o tuvo que trabajar más de lo esperado para poder verificar el progreso?

Una vez solucionado lo del backend, se pudo evaluar fácilmente, lo único molesto era tener q cerrar el compositor con el comando 'kill' cada vez q no salia con ctrl-shift-esc

---

1 respuesta

La evaluación del progreso era instantánea, ya que solo necesitaba compilar, cambiar de tty y ver los resultados de mis cambios en el código.

---

1 respuesta

¿Cuántos detalles de la API tuvo que comprender para poder utilizarla con éxito? ¿Cuántos de esos detalles estaban expuestos en la API?

Bastante pocos, Los del tutorial solamente. A mitad del capítulo me sentí cómodo como para ponerme a experimentar en el código sin sentirme perdido.

---

1 respuesta

Por lo general pocos, se podían utilizar algunas partes sin tener q conocer el resto.

---

1 respuesta

¿Qué tan fácil fue ver o encontrar los detalles de la API mientras la usaba? ¿Por qué? ¿Qué tipo de cosas son más difíciles de ver o encontrar?

Fue muy fácil encontrar los detalles de la API, porque la documentación online de la API es muy buena, y está escrita en un lenguaje muy sencillo. A veces me costó llegar a la definición por defecto de las funciones en el código fuente usando QtCreator, pero es por mi poca costumbre con QtCreator. Sin embargo, en la documentación por defecto están incluidas las implementaciones por defecto de los métodos, por lo que finalmente pude dar con el código que quería ver. Además, se incluyen explicaciones fáciles de entender de la implementación por defecto.

---

1 respuesta

No fue difícil, algo que me parecio nuevo eso si es tener a LCompositor como factory del resto de cosas, quizas se le puedan dar una vuelta mas en el comienzo del tutorial, no lo internalice hasta q lo lei en su entrada de la API

---

1 respuesta

¿Pudo usar la API exactamente "tal cual" o tuvo que crear nuevos objetos derivando de clases en la API?

Solo creé las clases derivadas que se detallan en el tutorial. Para mi experimentación no fue necesario crear más clases.

---

1 respuesta

No me queda muy clara la pregunta.

---

1 respuesta

¿Hasta qué punto pudo ampliar la API para satisfacer sus necesidades?

Solo creé las clases detalladas en el tutorial.

1 respuesta

Creo que en mi caso no corresponde.

1 respuesta

¿Cómo describiría sus experiencias respecto a la extensibilidad de la API? ¿Estuvo bien, no hubo suficientes oportunidades para extenderla o se vio obligado a hacerlo en contra de su voluntad?

Por lo visto en el tutorial, es muy fácil extender la API y crear clases hijas de las clases de Louvre.

1 respuesta

1 respuesta

Cuando necesitó realizar cambios en su código escrito mediante la API, ¿Qué tan fácil fué realizar el cambio? ¿Por qué?

Fue fácil, gracias a la documentación.

1 respuesta

No fue necesario.

1 respuesta

¿Hubo cambios particulares que fueron más difíciles o especialmente difíciles de hacer? ¿Cuales?

No hubieron cambios difíciles.

1 respuesta

No corresponde.

1 respuesta

Hay lugares de la API que parecen similares, pero resultan ser diferentes? ¿Cuales?

No que recuerde

---

1 respuesta

No.

---

1 respuesta

Al leer código que hace uso de la API, ¿Es fácil saber qué hace cada sección del código? ¿Por qué?

En casi todos los métodos que me encontre pareciera que si, sin embargo algunos como LCompositor::initialized() me dio la impresión que retornaba un bool si estuvo inicializada o no. Otro ejemplo es el método paintGL() que no es inmediatamente directo que renderize un frame.

---

1 respuesta

En general, si, porque los nombres de las clases y los métodos son descriptivos.

---

1 respuesta

¿Hay partes que sean particularmente difíciles de interpretar? ¿Cuáles?

No que recuerde

1 respuesta

lo puse arriba

1 respuesta

Al escribir el código, ¿Es fácil saber qué clases y métodos de la API usar?

Sí, es fácil, gracias a la documentación.

1 respuesta

Sí

1 respuesta

¿Qué tan relacionados están las clases y los métodos expuestos por la API con los objetos conceptuales que piensa que representan? ¿Por qué?

En general, es muy claro. Solamente hay algunas veces que es difícil imaginarse el flujo de control entre las funciones. Por ejemplo: repaint -> PaintGL, o quién llama a initialize(). Cosas de ese estilo.

---

1 respuesta

En general esta bien relacionados, se entiende a que abstraccion corresponde cada clase.

---

1 respuesta

¿Las clases y métodos expuestos por la API se asignan a objetos y comandos que tienen sentido en el dominio particular en el que trabajó?

Si.

---

1 respuesta

Sí

---

1 respuesta

Cuando utiliza la API, ¿Es fácil mapear las ideas en su cabeza al código? ¿Por qué?

Sí, excepto en el caso mencionado del flujo de control entre métodos, pero porque eso es complejo en sí, no porque la API esté mal diseñada.

1 respuesta

Al comienzo era difícil por mi falta de conocimiento del dominio, pero a medida que fui entendiendo las partes, Sí.

1 respuesta

¿Utilizó la API de formas inusuales o formas que los implementadores podrían no haber previsto? Si es así, ¿Puede dar algunos ejemplos?

A parte de seguir el tutorial hice un fondo de pantalla de un cuadrado que se mueve, pero al final solo se mueve cuando muevo el cursor.

1 respuesta

Sí. Llamar repaint() desde PaintGL(), que funcionó bien, a pesar de que no me lo esperaba.

1 respuesta

¿Hubo algo que quiso hacer con la API que no pudo lograr? ¿Qué cosa?

No.

1 respuesta

no

1 respuesta

Después de completar esta encuesta, ¿Puede imaginar formas obvias de mejorar el diseño de la API? ¿Qué formas? ¿Serían mejoras específicamente para sus propios requisitos?

No puedo imaginarlas, al menos con mi nivel de conocimiento actual.

1 respuesta

No.

1 respuesta

¿Podríamos comunicarnos con usted para obtener detalles adicionales? En caso afirmativo, ¿Cuál es la mejor manera de contactarlo (por ejemplo, correo electrónico, teléfono) y cuál es su información de contacto?

si. diegosandovalburgos@gmail.com

1 respuesta

felipe.quezada01@outlook.com

1 respuesta

## F. Entrevista de toma de requisitos.

La entrevista fué realizada a Diego Sandoval Burgos, estudiante de Ingeniería Civil en Informática de la Universidad Austral de Chile vía Discord.

El entrevistado menciona que intentó crear un compositor Wayland sin éxito. Su principal motivación fué que no le gustan los entornos de escritorio actuales, y que por lo mismo quiso desarrollar uno propio se adapte a sus necesidades/gustos.

Decidió utilizar la librería wl\_roots y variantes previas de esta pero no logró comprender en totalidad su funcionamiento debido a falta de documentación. Agrega que lo que más le ayudó a comprender su API fue revisar y compilar el código de un pequeño compositor de ejemplo (Tinywl), pero aún así, no supo como continuar implementando su propia lógica, ni tampoco recuerda si pudo desplegar el compositor en pantalla y ver algún tipo de resultado. Esta complejidad lo desmotivó un poco, pero aún así sigue con la intención de desarrollar su propio compositor.

Menciona además, que le hubiera gustado encontrar un tutorial que le explique paso a paso cada uno de los aspectos tanto de la API, como el funcionamiento del input/gráficos del sistema y las extensiones y protocolos de Wayland. Tiene conocimiento respecto al funcionamiento de Wayland a un alto nivel y de algunas de sus interfaces, pero desconoce qué extensiones/interfaces son requeridas por las aplicaciones de toolkits en Linux. Además tiene experiencia desarrollando videojuegos con OpenGL pero aún así le resulta complejo, sobre todo el manejo de texturas y agrega que le parece buena idea que Louvre ofrezca funciones para renderizar de forma más sencilla.

Respecto al uso de métodos virtuales, menciona que le parece buena idea si es que esto permitirá aprender de forma más rápida la API de la librería pero recalca que podría ser un problema o confundir al desarrollador si no se documenta que es lo que hace cada función por defecto.

En cuanto a la integración de las extensiones, menciona que le parece correcto, pero que se debería mencionar en la documentación cuales son y permitir identificarlas en caso de que ocasionen errores. También recalca que se debería poder incorporar extensiones adicionales.

Desconoce que son overlay planes de DRM, pero menciona que su incorporación a la librería no le molestaría, siempre y cuando fuera opcional.

Un aspecto positivo que recalca de wl\_roots es que está escrita en C, lenguaje que para él le resulta más sencillo de utilizar que C++.

Por último, menciona que la documentación debería estar escrita suponiendo que el desarrollador desconoce

totalmente su funcionamiento y el de Wayland. También propone la idea de crear otra librería pero para crear aplicaciones Wayland (clientes).

#### **Hipótesis respecto al personaje**

Se confirmaron todas las hipótesis planteadas con el personaje Paul.

#### **Hipótesis respecto a escenarios problemas**

Se confirmaron todas las hipótesis de los escenarios, aunque parcialmente el de desconocer OpenGL y el de optimizar el compositor utilizando overlay planes de DRM.

En el escenario de OpenGL se menciona que el desarrollador no tiene experiencia y por lo mismo se proponen funciones alternativas y simples para renderizar sin necesidad de estar familiarizado con OpenGL.

El entrevistado menciona que sabe utilizar OpenGL pero de forma básica, por lo tanto esa funcionalidad de todas formas le parece útil y por lo tanto se cumple la hipótesis.

Respecto al escenario de overlay planes de DRM, el entrevistado menciona no saber lo que es, pero que sí permite crear compositores más eficientes y es una funcionalidad opcional, no tendría problemas de que exista (es una funcionalidad opcional por lo tanto se cumple la hipótesis).

En resumen, se pudo observar que al entrevistado le parecen útiles las propuestas, principalmente porque piensa que le ayudarán a comprender cómo funciona Wayland y a crear un compositor de forma más rápida. Sin embargo, enfatiza la importancia de que la librería esté bien documentada y que ojalá cuente con un buen tutorial para aprender cada aspecto.

Un comentario que me pareció relevante fue el que considera C++ un lenguaje más complejo que C, lo cual es técnicamente cierto, pero una vez comprendido el paradigma orientado a objetos, se tiene la ventaja de que permite escribir código menos verboso y trabajar de forma más sencilla con estructuras de datos complejas.

### **Transcripción de la entrevista**

#### **¿Has intentado crear un compositor Wayland?**

Si, me he propuesto crear un compositor, pero la verdad es que he logrado avanzar muy poco, y me di cuenta de que se requiere mucho conocimiento e inversión de tiempo para poder crear uno con las librerías existentes actualmente.

#### **¿Cuál fué la motivación para decidir crear tu propio compositor?**

La motivación fué que a mi no me gusta ninguno de los entornos de escritorio que existen para Linux, o no me satisfacen totalmente, entonces quería crear uno que cumpla con todos mis gustos y necesidades, y como X11 está quedando obsoleto, opté por utilizar Wayland.

#### **¿Qué librerías intentaste utilizar?**

wl\_roots y también revisé superficialmente wlc que utilizaba el compositor Sway antes de que su creador inventara wl\_roots, pero wlc está obsoleta, así que mi intento real fue con wl\_roots.

**¿Cómo intentaste aprender a utilizar la librería?**

Para empezar, busqué tutoriales en Google haciendo búsquedas como "wl\_roots tutorial" o "how to create a Wayland compositor with wl\_roots?", y encontré sitios interesantes, pero nada que me satisfaciera totalmente. Encontré unos artículos escritos por Drew Devault, que es el creador de wl\_roots y Sway, pero estaban obsoletos, también encontré otros post sobre otra librería pero que está hecha en Rust, por lo que no me interesó mucho. Lo que más me acercó a aprender fue ver el código fuente de un compositor creado con wl\_roots que se llama Tinywl, que básicamente implementa las cosas más básicas, pero aún así, tenía muchas de líneas de código y estaba "obsoleto", tuve que modificar algunas funciones para poder compilarlo, pero en el fondo tampoco me ayudó demasiado a entender la librería.

**¿Qué tan complejo/sencillo fue comenzar a desarrollar? ¿Supiste por donde empezar?**

No, la verdad es que luego de buscar en Google y no encontrar documentación, opté simplemente por compilar Tinywl, lo cual me ayudó a entender algunos conceptos, pero no muchos.

**¿Pudiste llevar a cabo tu idea?**

No.

**¿Qué te impidió finalizar tu idea?**

Principalmente, la falta de documentación en línea sobre cómo utilizar wl\_roots, y también sobre cómo se unen todos los componentes, porque por un lado está el protocolo Wayland, por otro lado están las extensiones y módulos de la librería, por otro lado está todo lo que es gráficos de bajo nivel, el acceder al input, entonces todo eso no entiendo como se une y wl\_roots no lo explica de forma detallada.

En el fondo, la falta de un tutorial que te vaya explicando paso a paso que es lo que tienes que ir haciendo y cómo funciona y se interrelaciona cada cosa.

**¿Aún sigues interesado en crear tu propio compositor?**

Sí, pero no tanto como antes, ver lo complejo que es me desmotivó un poco.

**¿Sabes cómo funciona Wayland a un alto nivel?**

Sí.

**¿Conoces alguna interfaz del protocolo de Wayland o de extensiones?**

Tengo conocimiento de algunas interfaces generales del protocolo de Wayland como que es un "Seat", "Surface", "Registry", o conceptos de ese tipo, pero de otras extensiones no.

**¿Sabes qué extensiones de Wayland son requeridas por aplicaciones en Linux?**

Se que hay ciertas cosas que no funcionan si el compositor no tiene algunas extensiones, como el

mecanismo para compartir pantalla, pero no se especificamente cuales.

**¿Lograste crear un compositor básico utilizando alguna de las librerías actuales, o al menos desplegar gráficos en pantalla?**

Pude compilar Tinywl, y mostrarlo en pantalla, pero es prácticamente código que ya estaba escrito, y no recuerdo si pude mostrar aplicaciones en pantalla.

**¿Cuál es tu nivel de experiencia con OpenGL?**

He usado OpenGL para crear prototipos de videojuegos pero en un equipo, y el código base ya estaba listo, pero igual he aprendido por mi cuenta y comprendo cómo funcionan los vertex y fragment shaders, se hacer cosas básicas como mostrar un triángulo, un cuadrado, pero cuando hay que aplicar texturas o cosas por el estilo me cuesta un poco más.

**¿Te parece correcto que la librería ofrezca sus propios programas y funciones para facilitar el renderizado 2D de forma opcional?**

Si, me parece útil.

**¿Te parece correcto que la librería implemente internamente un compositor básico por defecto utilizando métodos virtuales, los cuales luego puedes sobreescribir con tu propia implementación?**

Si eso es para que la gente lo entienda bien, y le sirva para aprender, me parece super bien ya que te daría una cierta velocidad inicial para aprender y evitaría esa frustración inicial de no saber qué hacer o por dónde empezar.

**¿Encuentras algún problema con este acercamiento?**

Un posible problema es que si está todo el código hecho, tal vez el usuario no va a entender que hacen esas funciones por defecto, por eso sería bueno que en la documentación se explique o muestre el código por defecto de cada una.

**¿Te parece correcto que la librería implemente internamente las extensiones requeridas por aplicaciones de escritorio en Linux?**

Me parece correcto, pero se debería especificar en la documentación cuales son y en el caso de que hayan errores, que la librería te deje saber que extensión es la que produce el error. También debería permitir que utilice mis propias extensiones u otras extensiones.

**¿Encuentras algún problema con este acercamiento?**

Lo de no saber qué extensión produce el error, pero de todas formas pienso que el beneficio sería mucho mayor para un principiante, que si no estuvieran implementadas.

**¿Te parece útil que la librería brinde acceso a overlay planes de DRM para realizar composición usando**

**conectores de la GPU?**

La verdad no me siento capacitado para responder esta pregunta, pero si permite una mayor eficiencia y el que exista no genera ningún inconveniente prefiero que esté, o sea, si significa que si no lo utilizo, no me va a afectar, prefiero que esté.

**¿Qué te gustó de las librerías actuales?**

De wl\_roots me gustó que esté escrita en C ya que me parece más sencillo que C++, sobre todo por el paradigma orientado a objetos, creo que para utilizar C++ es necesario aprender muchas más cosas para sacarle provecho.

**¿Qué ideas se te ocurren para mejorar la librería?**

Hacer que la documentación no asuma que ya estás familiarizado con los protocolos de Wayland, que te vaya enseñando como si no supieras nada de Wayland.

Otra idea, que quizás no está muy relacionada, es que la librería o parte de ella también fuera un toolkit gráfico como GTK, para poder crear tanto aplicaciones como compositores.