Master thesis : Integrating Wayland with Flutter

Auteur : Rosca, Alex-Manuel
Promoteur(s) : Mathy, Laurent
Faculté : Faculté des Sciences appliquées
Diplôme : Master en sciences informatiques, à finalité spécialisée en "computer systems security"
Année académique : 2023-2024
URI/URL : http://hdl.handle.net/2268.2/19580

University of Liège - School of Engineering and Computer Science

Integrating Wayland with Flutter

Master's thesis completed in order to obtain the degree of Master of Science in Computer Science by

Alex-Manuel Rosca

Academic year 2023-2024

Promoter : Mathy Laurent

Contents

Abstract

Zenith is a Wayland compositor for Linux mobile devices. It is a user interface designed to run on smartphones running regular Linux distributions. By combining emergent technologies like Wayland and Flutter, we can provide a look-and-feel that competes with Android and iOS. Linux mobile devices are slowly on the rise. There is a desire to bridge the gap between desktop and mobile Linux and provide a user experience on par with popular mobile operating systems.

The idea for this project came to me when I tried other open-source mobile compositors created with GTK and Qt and realized that basic features like scrolling felt unnatural. Knowing that Flutter was designed from the ground up with mobile devices in mind, I thought it would be a great idea to create a mobile user interface using this UI framework.

In this thesis, a free and open-source prototype of a mobile compositor was implemented, explaining

the steps taken to integrate Flutter and Wayland to create a touch-friendly user interface that can be also run on the desktop platform. Documentation on this topic is very scarce, and this thesis will also go through the technical challenges and solutions that I encountered while integrating these two technologies.

As a result of my work, other open-source projects are now interested and have started integrating my implementation of this technology stack into their projects.

Introduction

Making a compositor is a daunting challenge because it takes low-level knowledge of the Linux kernel, the graphics stack, the Wayland protocol, and the Flutter Engine. Not many people on this planet have this knowledge, which makes it even more difficult to acquire because there is barely any documentation on the matter. Lots of times, I had to rely on just the source code to understand how things work.

Documentation got a little better since I started this project. Google has published some examples of how to integrate the Flutter engine in an embedded environment, which could have helped me a lot, even though these examples are still very basic.

This project started as a personal project, and I proposed it to Prof. Laurent Mathy for my thesis because I wanted to dedicate more time to developing it. At the time I presented this project to him, I already had a somewhat working prototype that could display applications and switch between them, but it wasn't very stable. Since my thesis started, I have made lots of changes under the hood. I rearchitectured the codebase, fixed bugs, and implemented new features like app closing, lock screen, app drawer, clipboard support, and more. I'm thankful to Prof. Laurent Mathy for allowing me to work on this project for my thesis because it has been useful to other people and I'm proud of what I've accomplished.

The repository of the compositor is available on GitHub at https://github.com/roscale/zenith

Technical background

Compositors

A compositor is a software component responsible for managing the display of graphical user interfaces (GUIs) and handling the rendering of graphical elements on the screen. Compositors play a crucial role in creating a visually appealing and seamless user experience. Here are some key functions and aspects of compositors in Linux:

- Window Management: Compositors manage the placement and rendering of windows and graphical elements on the screen. They handle tasks such as window stacking, positioning, resizing, and applying visual effects (like transparency or shadows).
- Compositing: Compositors are named after their ability to perform compositing, which involves blending multiple graphical elements or "layers" together to create a single, coherent image. This is essential for creating smooth transitions and effects, like window animations or transparency.
- Hardware Acceleration: Many modern compositors take advantage of hardware acceleration through graphics drivers and GPUs (Graphics Processing Units). This allows for efficient rendering and smoother performance.
- Desktop Effects: Compositors often provide desktop effects such as window animations, wobbly windows, and other visual enhancements. These effects can improve the overall user experience.

Dart

Dart is an open-source programming language developed by Google. It is a general-purpose language designed for a wide range of application development, from web and mobile apps to server-side and command-line applications. Dart follows an object-oriented programming paradigm, making use of objects and classes as core building blocks for software development. This object-oriented nature allows developers to apply principles like inheritance, encapsulation, and polymorphism to create well-structured and maintainable code.

One of the most prominent use cases of Dart is its association with Flutter, an open-source UI framework also developed by Google. Flutter uses Dart as its primary programming language and has gained widespread popularity for building natively compiled applications for mobile, web, and desktop from a single codebase. The ease of use, performance, and ability to create attractive user interfaces have contributed to Flutter's success in the mobile app development space.

Flutter

Flutter is a relatively new cross-platform UI toolkit created by Google and designed from the ground up with mobile devices in mind. Flutter's initial purpose was being able to create Android and iOS apps from a single code base, while providing an identical look and feel compared to native applications. Since its early days, Flutter expanded its support to Windows, Linux, and the Web, allowing developers to run their apps on other platforms. Even the University of Liège adopted Flutter into its mobile development course.

In order for Flutter to run on this many platforms without rewriting a large portion of the code, Google decided to modularize it into 3 parts (see figure 1).

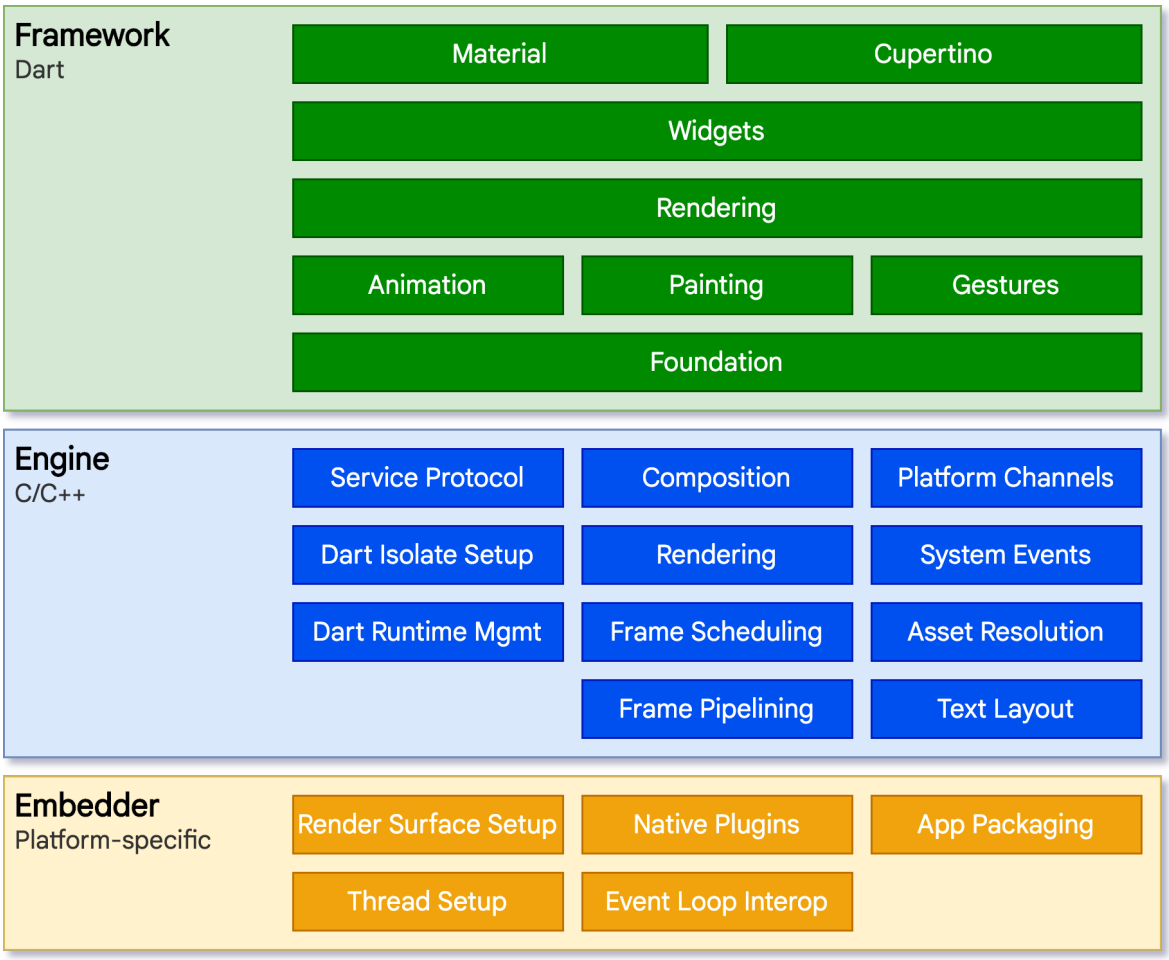

Figure 1: Flutter has 3 layers, and the embedder is the only one that's platform-specific. .

The Flutter framework provides a high-level abstraction written in Dart for building mobile applications. It includes a rich set of widgets for building user interfaces in the style of Android and iOS. This layer also takes care of gesture handling and animations.

The Flutter engine is responsible for rasterization using the Skia graphics library. It also provides low-level features like text layout, accessibility, and networking.

The Flutter embedder is the only platform-specific layer that ties together the Flutter engine and the underlying platform. It has the role of initializing the Flutter engine which in turn initializes the Dart

VM and runs the app. Furthermore, when the engine needs to communicate with the platform, it does so through the embedder. For example, when the engine generates a new frame, it will ask the embedder to display it on the screen.

This sort of communication is done through inversion of control. The Flutter engine is essentially a black box that runs on its own, and the embedder is responsible for implementing all the necessary callbacks that the engine will call when it needs to communicate with the platform. The embedder is also responsible for reading input events (keyboard, mouse, touch) from the platform and forwarding them to the engine. When someone wants to port Flutter to a new platform, they only need to reimplement the embedder. First, they need to provide all the necessary callbacks for the engine to run on that specific platform. Second, they need to listen and forward all user input to the engine if an interactive experience is desired.

Every platform has its own embedder. When you compile a regular Linux Flutter application, the compiler will produce a shared object called `libflutter_linux_gtk.so` which is actually the engine and the embedder implemented using GTK, bundled in a single library. On the Linux platform, the embedder will use the GTK toolkit to spawn a window and communicate with the existing compositor. GTK provides an abstraction over the Wayland protocol which handles input and notifies the compositor when a new frame is available.

Provider, Riverpod, and Freezed

The Provider library in Flutter is a state management solution that simplifies the process of managing and sharing state in a Flutter application. Provider builds on the InheritedWidget and BuildContext concepts to provide a straightforward and efficient way to propagate and consume state across different parts of the widget tree. It introduces the idea of "providers" which are objects responsible for holding and exposing pieces of your application's state.

Riverpod is another state management library for the Flutter framework. It is designed to help manage the state of an application in a clean, scalable, and efficient manner. Riverpod is often considered an alternative to other state management solutions like Provider or Redux and is known for its simplicity and flexibility.

Riverpod provides a way to create reactive state objects, often referred to as "providers". These providers can hold data and notify widgets when their data changes, making it easy to build reactive UIs. It also promotes immutability by design. The state is not mutated directly, but new states are created when changes are needed.

Providers can be organized into families, allowing you to create multiple instances of a provider with

different configurations. This is useful when you need to manage state for multiple instances of the same type, something that was impossible to do with Provider.

Defining immutable classes in Dart is tedious. You have to define a constructor for all your properties, override `toString`, equality, and `hashCode`, and you cannot easily create a copy of a nested object structure.

Freezed is a code generation library for Flutter and Dart that simplifies the process of creating immutable classes, including data classes and union classes. It's particularly useful for managing complex and nested data structures, which are common in mobile app development. Freezed generates code that reduces boilerplate and makes it easier to work with immutable data in your Flutter application.

Flutter platform channels

Flutter platform channels are a mechanism that allows you to establish communication between Flutter, a popular open-source framework for building mobile applications, and native platform code, which can be written in languages like Java, Kotlin, Objective-C, or Swift. This communication is essential for integrating platform-specific features and libraries into your Flutter app, as well as for accessing native APIs and services that are not directly exposed by Flutter itself.

Platform channels provide a way for the Flutter Dart code (which is used to build the UI and application logic) to interact with the native code (written in Java, Kotlin, Objective-C, or Swift) to perform tasks that are outside the scope of what Flutter offers. This includes accessing device-specific sensors, using platform-specific features, or integrating with third-party native libraries.

Platform channels consist of two main parts: method channels, and event channels.

1. Method channels are a way for Flutter to invoke platform-specific methods in the native code. They allow you to send messages from Dart to native code and receive responses back. These messages typically consist of a method name and optional arguments, and the native code processes the request and sends a result back to Flutter.

For example, if you want to use the device's camera, you might define a method channel that communicates with the native camera API, allowing you to take pictures and access camera settings from your Flutter app.

2. Event channels are used for sending data in the opposite direction, from native code to Flutter. They enable the native code to send asynchronous events or data streams to the Flutter code, which can then handle and react to these events.

For example, you can use event channels to stream sensor data (e.g., accelerometer or GPS data) to

your Flutter app in real-time. The native code collects the data and sends it to Flutter using the event channel, where it can be displayed or processed as needed.

Codecs are an important part of Flutter platform channels because they are responsible for serializing and deserializing data when it's sent between the Flutter Dart code and the native platform code. Flutter supports several data types, including primitive types like numbers and strings, as well as more complex data structures like maps and lists. Codecs ensure that data can be properly converted between the Dart and native code representations.

Flutter provides built-in support for common data types and offers a set of standard codecs to simplify the process of encoding and decoding data. The supported codecs include:

- StandardMessageCodec: Used for basic data types like integers, doubles, strings, lists, and maps.
- JSONMessageCodec: Allows you to work with JSON data, which is commonly used for structured data interchange.
- BinaryCodec: Provides a more efficient, lower-level binary data transfer option.

Wayland

Wayland (see figure 2 for its architecture) is an asynchronous object-oriented protocol that clients can use to talk to a compositor in order to make themselves visible and get input from the user. A compositor is a program that manages the display of graphical elements on the screen. It can be a window manager, a desktop environment, or a graphical shell. Clients are programs that want to display something on the screen, like a web browser or a video game. The Wayland protocol is a replacement for X11 which is the de facto standard for graphical applications on Linux. X11 is a very old protocol and it has a lot of legacy features that are not used anymore. Wayland is a modern one that only implements the features that are actually used by modern applications.

The specification of the Wayland protocol is written in XML [37], but the freedesktop group who developed Wayland also created an implementation of the protocol split into 2 C libraries: a client library and a server library.

Alongside the base Wayland protocol, there are also many extensions that provide additional features. One such extension is called `xdg_shell`[41] which provides a set of interfaces for creating windows and popups. Somewhat confusingly, the base Wayland protocol also provides a similar interface called `wl_shell` for achieving the same thing but it has been deprecated in favor of `xdg_shell`. `wl_shell` still exists in the protocol and will probably never be removed because it would break backward compatibility, but it's not recommended to use it anymore.

Unfortunately, these necessary protocol extensions are not implemented by the server library. Freedesk-

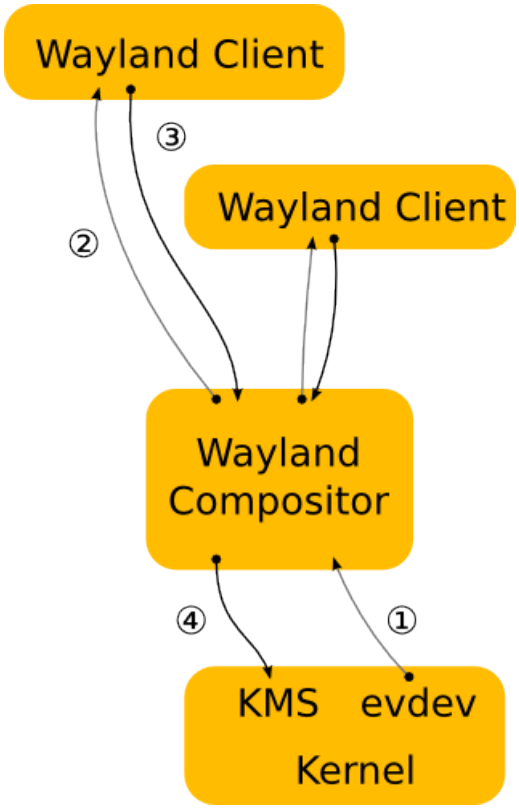

Figure 2: Wayland's architecture, the Wayland server, and the compositor merged into a single component. .

top created a tool that converts an XML specification into C code [4], but this code only generates the appropriate structs and functions for the objects, events, and signals defined in the XML file. Most of the time, protocol specifications also describe in English additional semantics that cannot be translated into code and must be implemented by the compositor. For example, in the `xdg_shell` extension, the protocol defines a list of functions that the client can call to modify the state of a window (minimum size, maximum size, maximized, fullscreen, etc.) and when the client is done he should call a function called `commit`. The protocol specifies in English that compositors should apply these changes atomically on commit, but this is something that cannot be automatically translated into code. Compositors must implement all this bookkeeping themselves.

Smart people found this situation suboptimal because all well-implemented compositors should respect protocol semantics regardless of how different they were. For this reason, another group of people created a library called `wlroots`[40] that does all the bookkeeping for you regarding the accumulation of state, and the ordering of events. This library is unopinionated in the sense that it only does its best to respect the protocol semantics, but it lets you the freedom to handle events however you want.

wlroots

`wlroots`[40] is a modular and composable library that provides an unopinionated foundation for developing Wayland compositors. This foundation contains abstractions around various parts of the Linux kernel, the graphics stack, and the Wayland protocol itself. More precisely, `wlroots` provides a simplified interface to input and display hardware, including `libinput` and KMS/DRM, the Linux subsystems for driving the monitor's output and interfacing with GPUs.

Most user programs are event-driven. The user moves the mouse, the program does something, instructs the graphics card to render a new frame, and then it goes to sleep until some other external event wakes it up. Compositors are no different, although the events a compositor is interested in are quite different from a normal application. It has to listen to keyboard, mouse, touchscreen, and touchpad events. It has to listen to Vertical Synchronization (VSync) events coming from the monitor because, on a 60 Hz monitor, the compositor needs to display something new to the user 60 times per second. It also has to listen to Wayland events because various applications might decide to create or destroy a window or a popup.

This design pattern is called an event loop. The Wayland server library[38] provides an event loop with a friendly API based on callbacks. Although this event loop is mainly used for Wayland events, `wlroots` extends its purpose by attaching more file descriptors that get notified when input or display events are available. You can imagine, for example, when a monitor is plugged into the computer, a callback is called which will handle this particular event.

When a callback is called because a new resource is available, wlroots provides an object to that resource. The user can then query information about that resource and also register some callbacks on events that happen to that resource. For example, when a new keyboard is plugged in, the user can register a callback to be called when a key is pressed. In the code, this observer pattern is expressed as signals and listeners. Resources contain a structure with many signal objects, one per type of event, and each signal object can store a list of listener objects. If the user is interested in a particular event, he can create a listener object, give it a callback (a function pointer), and register the listener to the signal object. When the event happens, the signal object will notify all listeners which will call their corresponding callbacks. In most cases, events carry some data with them, like the key that was pressed for example, but on some occasions, events also carry other resources, creating a chain of resources. One such example is the `wlr_compositor` global object that has a signal called `new_surface` which is emitted when a new surface is created. The callback associated with `new_surface` is passed a pointer to the newly created `wlr_surface` object. This object has another signal called `new_subsurface` and its callback is passed a pointer to a `wlr_subsurface`. Every resource also has a special signal called `destroy` that is emitted just before the resource is destroyed. You might imagine if the user unplugs his monitor, the destroy event on its corresponding `wlr_output` object will be emitted.

epoll, eventfd, and timerfd

epoll (event polling) is a scalable I/O event notification mechanism in the Linux kernel. It is used to efficiently manage I/O events, making it a valuable tool for developing high-performance applications, especially in scenarios where asynchronous I/O or event-driven programming is required.

Unlike traditional mechanisms like select and poll, which become less efficient as the number of monitored file descriptors increases, epoll is designed to efficiently manage a large number of file descriptors simultaneously. This scalability is crucial for applications such as network servers or real-time data processing systems, which often need to monitor multiple network connections concurrently.

One of epoll's key advantages is its support for asynchronous I/O event monitoring. It allows applications to continue performing other tasks and only receive notifications when relevant I/O events occur. This asynchronous approach improves overall system responsiveness and performance.

epoll employs a callback mechanism that ensures that only relevant events are reported. This minimizes the overhead of continuously polling file descriptors that are not ready for I/O. Applications can use the epoll_wait system call to wait for events and process them when they occur. This event-driven programming approach is valuable for creating efficient and responsive applications.

eventfd is another Linux kernel feature that serves as a mechanism for inter-process communication (IPC) and event notification. It allows different parts of an application or even different processes to

efficiently signal events or exchange data. eventfd is particularly useful in scenarios where applications need to communicate or synchronize without relying on traditional mechanisms like signals or message passing.

One of the primary use cases for eventfd is efficient synchronization between different parts of an application, such as between threads or components. It can also be employed for inter-process communication when multiple processes need to signal or share information without the overhead associated with traditional IPC methods.

Unlike signals, which involve context switches and signal handlers, eventfd provides an asynchronous, file descriptor-based method for signaling events. This approach is more efficient in scenarios where applications or processes need to communicate and synchronize frequently.

timerfd is another feature in the Linux operating system that provides a mechanism for creating timer events as file descriptors. It was introduced in Linux kernel version 2.6.25 and is particularly useful for applications that require precise timing, event scheduling, and asynchronous notifications.

Unlike traditional timers, timerfd allows you to work with timers using the same file descriptor interface used for files and sockets. This means you can use standard I/O functions like read, write, and select to interact with timers, making it more convenient to integrate timer functionality into applications.

When a timer expires, timerfd generates an event that you can monitor through the associated file descriptor. This event can be handled like any other I/O event, making it suitable for use with mechanisms like select, poll, or asynchronous I/O frameworks like epoll.

libinput

`libinput`[20] is a library that provides an abstraction over input devices like keyboards, mice, touchscreens, and touchpads. It is a library that sits between the kernel and the compositor. The kernel provides a generic API for input devices called `evdev`, but it's not very ergonomic to use. libinput provides a friendlier API and also some additional features like gesture recognition (scrolling with 2 fingers for example). It also provides functions to configure input devices. libinput is not a Wayland-specific library, but it became the de facto standard for input on compositors.

wlroots uses the libinput library for handling input devices. When a new input device is plugged in, libinput will notify wlroots which will create a new `wlr_input_device` object. This object has a union of pointers to more specific input device types like `wlr_keyboard`, `wlr_pointer`, `wlr_touch`, and `wlr_tablet_pad`. The user can query the type of the input device and access the right pointer. Each of these input devices contains an anonymous struct `events` which contains a list of signals that the user can register listeners to. For example, the `wlr_touch` object has a signal called `down` which is emitted

when a finger touches the screen. The callback associated with this signal is passed a pointer to a `wlr_event_touch_down` struct which contains information about the touch event, like the coordinates of the touch.

freedesktop.org

Freedesktop.org focuses on developing and promoting standards and specifications that enhance the user experience and compatibility across different Linux distributions and desktop environments. Some of the notable achievements of the FreeDesktop.org project include the development of the XDG Base Directory Specification, which helps define where user-specific configuration and data should be stored, and the Desktop Entry Specification, which provides a standardized format for desktop application launchers. The project also plays a crucial role in the development and promotion of the Wayland display server protocol as a potential successor to the X Window System.

DBus

DBus, which stands for Desktop Bus, is a message bus system used for inter-process communication (IPC) and coordination between various software applications within a Linux or Unix-like operating system. It provides a way for different applications or processes to communicate with each other and exchange data, such as method calls, signals, and properties.

Here's a simple example to illustrate how DBus works:

Imagine you have two applications running on your Linux system: a music player and a notification system. The music player wants to send a notification to the notification system whenever a new song starts playing. DBus can facilitate this communication.

- Initialization:
  - Both the music player and the notification system connect to the DBus session bus, which is a system-wide message bus.
- Sending a Signal:
  - When the music player starts a new song, it sends a signal (message) via DBus to the notification system.
  - The signal may include information about the song, such as the song title, artist, and album.
- Receiving the Signal:
  - The notification system, which is also connected to DBus, listens for signals related to music events.
  - When it receives the signal from the music player, it can display a notification with the song information on your desktop.

DBus is used for lots of purposes, including system services, configuration management, and more. The init system systemd, for example, registers itself on DBus and the CLI tool `systemctl` uses DBus to communicate with it.

EGL and OpenGL ES

EGL, which stands for "Embedded-System Graphics Library" is a crucial component in the field of computer graphics, particularly in embedded and mobile systems. EGL is an interface that facilitates the interaction between rendering APIs (such as OpenGL and OpenVG) and the underlying native windowing system of a platform. Its primary purpose is to help applications create and manage rendering contexts and surfaces in a way that is platform-independent.

OpenGL, which stands for "Open Graphics Library" is an open-standard, cross-platform graphics API (Application Programming Interface) that provides a set of functions for rendering 2D and 3D computer graphics. Developed by the Khronos Group, OpenGL is designed to be hardware-agnostic, allowing developers to create graphics applications that work across a wide range of platforms and graphics hardware.

OpenGL ES (OpenGL for Embedded Systems) is a subset of OpenGL designed for mobile and embedded platforms. It is optimized for efficiency and is commonly used in smartphones, tablets, and other embedded devices.

When using OpenGL in an embedded or mobile environment, EGL is often used to initialize OpenGL. EGL creates and manages OpenGL contexts, which are used to store rendering states and resources.

DRM and KMS

DRM is a subsystem within the Linux kernel designed to manage graphics rendering. It provides a framework for efficient, direct, and secure access to graphics hardware, allowing userspace programs to communicate with and control GPUs (Graphics Processing Units) and other graphics devices.

KMS is a component of DRM that specifically focuses on configuring display modes and controlling graphics output. It allows the kernel to take control of the display hardware, setting display parameters such as resolution, refresh rate, and the frame buffer memory used for graphics rendering.

Swap chains

Monitors display frames at a fixed rate called the refresh rate. If a monitor's refresh rate is 60 Hz (which is most of them), it will try to display a new image 60 times per second. The compositor should ideally keep the monitor fed with new frames but sometimes the composition might take longer than

expected. Some fancy effects can be too expensive on the GPU and the frame might be delayed. If the monitor doesn't get a new frame in time, it's best to just repeat the latest one.

It's important to know that monitors don't display frames instantly because we live in a world where physics exists. When the kernel serializes a frame buffer for the monitor, it starts from the top left, going from left to right. After the frame has been sent, the time remaining is called the vertical blank interval. The compositor must be careful because if it swaps the frame buffer with another one while the kernel is sending pixels through the cable, the monitor will end up displaying half of the old frame and half of the new frame, an effect called screen tearing. This method is called **asynchronous page flipping**. Swapping a frame buffer with another one is called page flipping, and it's asynchronous because the compositor does not synchronize with the VBlank interval. This way of page flipping can be sometimes useful if the user wants to see the latest changes immediately, like when playing a competitive game where response time matters. For the other 99% of cases, screen tearing is just unnecessary and ugly.

Usually, compositors will stay in sync with the monitor and won't generate more frames than the refresh rate of the monitor because it's a waste of computing power if some of these frames are not even displayed to the user. This method to stay in sync with the monitor is called **vertical synchronization** or VSync for short.

Swapping between multiple frame buffers is necessary for providing a tear-free experience. This collection of frame buffers is called a *swap chain*. We will now discuss different types of swap chains.

Single buffer

Actually, not a swap chain because you must have at least 2 buffers to swap, but it's the most naïve form. The GPU renders into this frame buffer while the kernel will serialize in parallel from the same frame buffer. Rasterization takes time, especially when the compositor has to render many windows, icons, and probably some fancy effects too. Within this time interval, the frame buffer is incomplete and if the monitor displays an incomplete frame buffer, the user will see in most cases missing UI elements. Even worse, due to the variability of render time, the state of the frame buffer that the user sees is basically random. One frame could be complete and the next could have missing elements, resulting in very annoying flickering. I hope we can all agree that shipping a compositor with epilepsy warnings is not a good idea, and the solution is to use a proper swap chain.

Technically, it could be possible to avoid all problems by starting and finishing the rendering within the same VBlank interval, but nobody has a crystal ball to tell you if the rendering will finish in time. This technique was used in the last century by game consoles like the NES, but nowadays we are expected to run the GPU in parallel with the other components in the system, otherwise, we are

leaving performance on the table.

Double buffer

The GPU can render into a frame buffer while the monitor can display the other one. There are 2 ways to handle the swap. We can do **asynchronous page flipping** and swap the buffers as soon as the GPU completes the frame. This gets rid of the flickering because the monitor will always display pixels from complete frames, but screen tearing will be present. This will work if your goal is to produce frames as fast as possible.

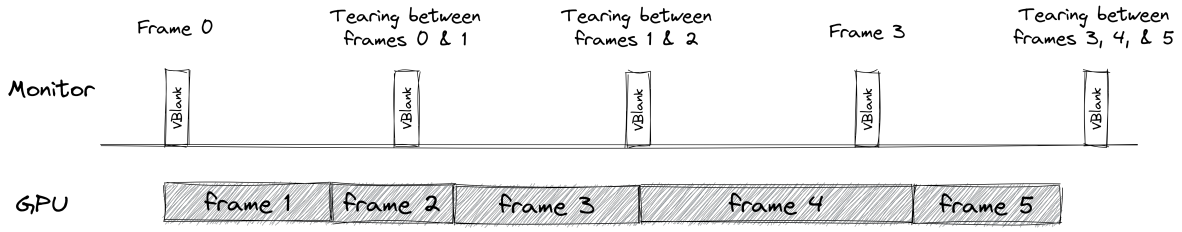

Figure 3: Double buffering with async page flipping . The user sees the newest frames as early as possible at the expense of screen tearing. This method provides the lowest latency.

The second way is to make the GPU synchronize with VBlank intervals before swapping the frame buffers and starting working on a new frame. We have successfully eliminated screen tearing but now we've introduced additional latency. On a 60 Hz panel, the monitor will display frames started 16.6 ms ago. In my opinion, this amount of latency is perfectly acceptable for a casual compositor who's not trying to display video games.

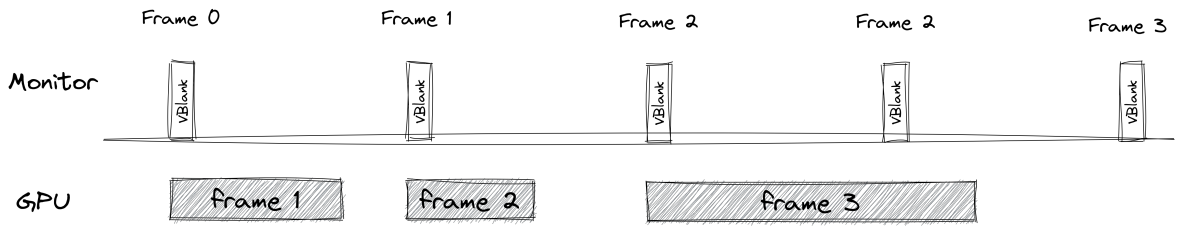

Figure 4: Double buffering with VSync . The animation is the smoothest because the interval between generating a frame and displaying it is constant. The compositor will have to wait for the next vertical blank interval before displaying a new frame, which increases latency.

The latency could be improved by delaying the rendering but at the risk of missing the next VBlank and causing stuttering. But there is another problem with this approach. If we strictly stick to VSync and the GPU can't complete frames within the refresh rate, the framerate will effectively be cut in half,

or possibly even worse if the GPU is taking a really long time.

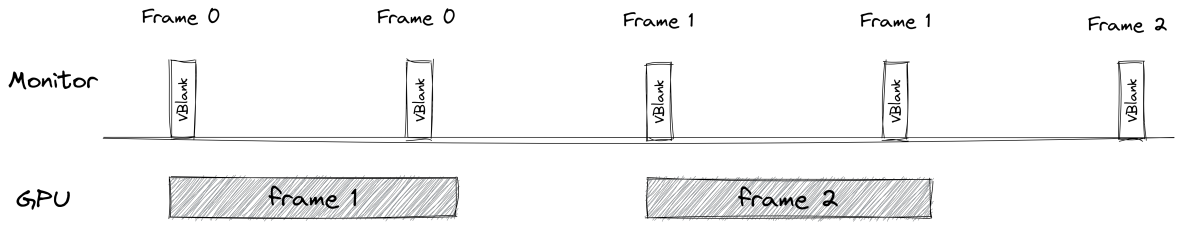

Figure 5: Double buffering with VSync, worst case scenario . With a naïve implementation, if the GPU takes too long to render and misses the vertical blank interval, the framerate will be cut in half because the compositor will only start a new frame at the vblank interval.

Triple buffering

There are 3 buffers. When the GPU finishes rendering into the first buffer and the monitor processes a second buffer, the GPU can switch to the third buffer and begin rendering the next frame immediately without waiting for the monitor. GPU utilization is maxed like in double buffering with async page flipping, but screen tearing is eliminated. The monitor will always display the last frame rendered.

It seems like with this method we've eliminated visual artifacts while still being able to produce frames as fast as we want. While this is true, what we've lost is smoothness. When the user sees a frame, he sees the state of the compositor when the CPU prepares commands for the GPU using the latest state in memory. It's important that the interval between the GPU starting a new frame and the monitor displaying that frame, remains consistent. If we stick to VSync, we know that the GPU always starts working on a new frame one refresh cycle before it is being displayed (16.6 ms at 60 Hz), otherwise, this interval varies from frame to frame and the user will experience jitter. Every frame will be shown correctly, but the motion will not be smooth.

Texture sharing

Textures represent large amounts of data. A single 1920x1080 RGBA texture is over 8 MB uncompressed. For this reason, it's not optimal to send pixels to the compositor over the Wayland protocol. Linux provides 2 APIs for efficiently sharing buffers.

Shared memory (shm[30])

shm is an API that allows a process to share a memory region with another process. These memory regions live in RAM. The Wayland protocol provides support for creating shared memory pools and pixel buffers between the compositor and the client, using `shm` under the hood. When a client creates

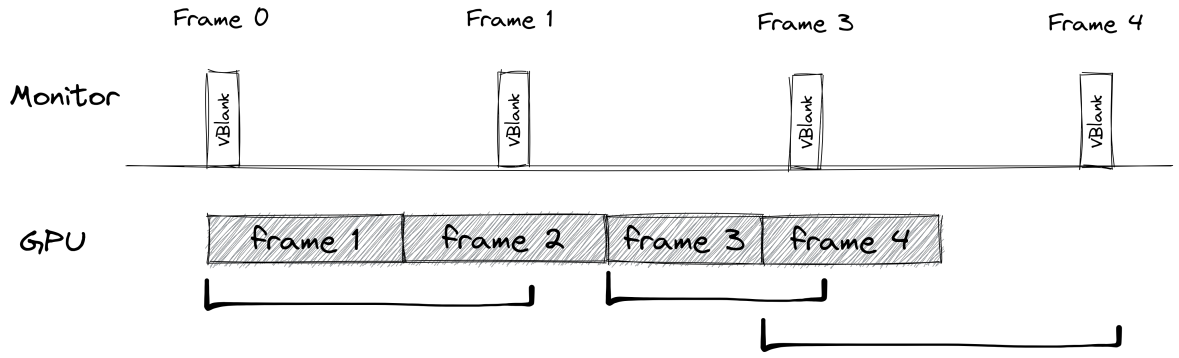

Figure 6: Triple buffering with jitter . The user sees only full frames as in double buffering with VSync, but the animation is not smooth. The interval between starting a frame and displaying it varies from frame to frame. The latency is better than double buffering with VSync but the user experiences jitter.

a memory pool, the compositor must inform the client about all the available pixel buffer formats it supports. Depending on the hardware, the compositor may not be able to efficiently render all pixel formats.

Most likely, the compositor will render the final composition on the GPU for efficiency. Unfortunately, the GPU can't sample the pixel buffer directly from RAM. The compositor has to copy the pixel buffer to VRAM and create a texture out of it. The OpenGL API provides the function `glTexImage2D` for copying pixel buffers from RAM to VRAM. Every time the client produces a new frame, the compositor has to make a copy. Needless to say, this wastes CPU and GPU power which negatively impacts efficiency, a very important parameter on battery-powered devices. It can also impact performance under a full system load because hardware resources are obviously finite and if the compositor is wasting clock cycles, user applications have fewer resources available. The problem can be alleviated by simply not making buffer copies if a window is not visible, but this depends entirely on how the user has arranged its windows on the screen. Mobile devices are less prone because at most one or two apps are shown at a time, but it's still not ideal.

It's important to acknowledge that shm is a method for sharing any kind of buffer, not necessarily pixel buffers. When the compositor creates a texture, it needs more information than just an array of bytes and its length. By looking at the signature of `glTexImage2D`, we also need to know the width, height, and format of the texture. These attributes are not shared through shm, but through Wayland when the client creates a buffer[6].

Direct memory access buffers (dma-buf)

dma-buf is a kernel subsystem that allows device drivers to share and synchronize buffers between each other. If 2 devices want to share a buffer, both device drivers must implement support for dma-buf. When driver `A` wants to share a buffer with driver `B`, `A` is called the exporter, and `B` is the importer. The protocol imposes that the exporter of a buffer manages the lifetime of that buffer, and keeps the buffer alive while the importer is using it. In our use case, both the importer and exporter are the same driver, the graphics driver. Buffers are not shared between 2 devices, but between 2 processes, the Wayland client and the compositor. If a client renders a texture using the GPU, it can export the texture and send the dma-buf file descriptor to the compositor through Wayland. The compositor will import the texture and sample from it during composition. If the Wayland client and the compositor use the same GPU, no copies are made. If they use different GPUs, the texture has to be transferred through the PCIe bus.

Similar to shm, this API was created for generic purposes, not only graphics. For this reason, the client has to send texture properties alongside the dma-buf file descriptor[5]. As a side note, some texture formats require sharing multiple dma-bufs for a single texture. YUV, for example, has a buffer per color component. This segregated memory arrangement is apparently more efficient for video encoding.

Implementation

High-level architecture

The language that I chose to write the compositor is C++. Knowing that I will be using the wlroots library as a building block for my Wayland server, the only sane choices were C or C++. I chose C++ because its standard library provides a lot of useful features like smart pointers, vectors, and hashmaps.

The compositor is split into 3 main components: the Wayland server, the Flutter embedder, and the Flutter application which implements the UI in Dart. The main thread has an event loop that listens and handles Wayland, input (touch, mouse, keyboard), and output (monitor) events. A second thread is spawned which initializes the Flutter engine and runs another event loop that will execute tasks posted by the engine, including callbacks registered for platform messages. The Flutter engine is a black box and spawns many more threads for the Dart VM that I don't know about after it has been initialized and run.

The Wayland server and the Flutter embedder could have been merged into one single thread by I've chosen to separate them for modularity. In a far future, I might want to add a 3D renderer to my compositor in case I want to see my windows in 3D space with a VR headset on my head. As they are

2 different threads, the approach I took for handling communication between them is message passing. I've had bad experiences with mutexes in the past and I'm also seeing a general shift in programming languages from shared memory to message passing. Figure 7 shows the high-level architecture of the compositor.

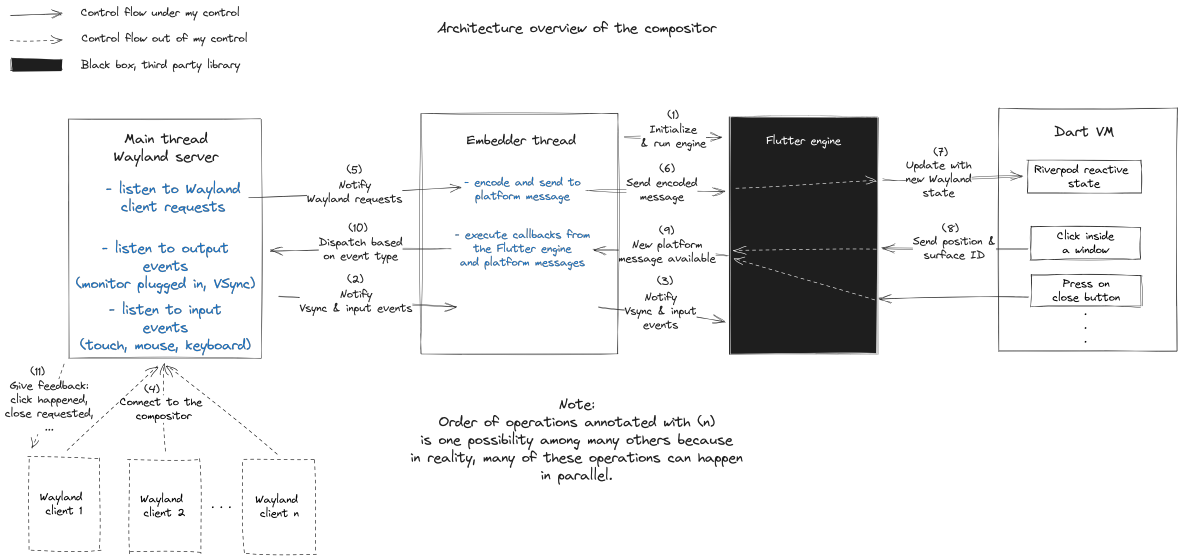

Figure 7: High-level architecture of the compositor. The main thread hosts the Wayland server and another one that runs the Flutter embedder..

Communication between threads

If the Flutter embedder needs some access or to modify some state owned by the Wayland server, the communication goes like this. If the embedder wants to execute some code on the server's thread, it constructs a closure containing the code. This closure is then put in a queue of closures that I'm calling "callable queue". The event loop of the Wayland server will dequeue it when it becomes free, and execute it. Since one thread enqueues and the other dequeues, the queue is protected by a mutex. Modifying the state of the server is easier by sending closures over a queue and executing the code on the server's thread, then having to lock multiple mutexes to protect all the variables that I will modify from the embedder's thread.

Often, the embedder also wants some data in return after accessing the server's variables. Before creating the closure, a channel is created that can be written to one thread and read from another. The embedder creates a channel, creates a closure that captures the channel, enqueues the capture, and then waits for a response on the channel. When the closure executes on the server thread, it will

write the response on the channel, unlocking the embedder's thread.

Callable queues and channels don't come from the standard C++ library. I've implemented them myself using queues, mutexes, condition variables, and eventfd. The event loop of both threads is backed by epoll. The event loop of the server thread listens to queue events through an eventfd file descriptor. When the embedder puts something in the queue, it will write a dummy value to the eventfd file descriptor which will wake up the server thread. Channels are just containers for a single templated value protected by a mutex, and using a condition variable (std::condition_variable) to notify a thread and wait for a value to be written. In this case, I used a condition variable to notify the other thread instead of eventfd mainly because I don't need to integrate it into a polling framework like epoll, and it's simpler to use than eventfd.

These are the steps we'll need to get a working compositor:

- Download the Flutter engine binaries and link the engine to the embedder.
- Start writing the embedder: set up the Flutter engine and run it.
- Successfully render the demo Flutter app on the screen.
- Spawn a Wayland server and forward input events to the Flutter engine.
- Register relevant Wayland event callbacks.
- Send Wayland information to the engine through platform channels.
- Register the OpenGL textures of Wayland clients in the engine.
- Keep track of Wayland's state in Dart, and render clients' textures and other UI elements.

Downloading the Flutter engine binaries

When you compile a Flutter project for Linux, the compiler will produce a shared object called `libflutter_linux_gtk.so` which is the Flutter engine and the GTK embedder bundled in a single library.

The GTK embedder is adequate for when you want to show your Flutter app in a window, but doesn't work anymore when Flutter needs to be the compositor itself. GTK implements a Wayland client, not a server. This is the reason we have to write our own embedder without depending on GTK, and why we can't link to `libflutter_linux_gtk.so`.

Flutter projects can be compiled in 3 modes: debug, profile, and release. Equally, the Flutter engine can also be compiled in these 3 modes and both must be compiled in the same mode, otherwise, the engine will complain and won't run the Flutter project. Furthermore, if a Flutter project was compiled with a version of Flutter, the engine must be the same version, otherwise, the engine will complain again.

Google provides a precompiled dynamic library[7] that only contains the engine without GTK, but only in release mode. I would like to download the engine compiled in all 3 modes because I want to be able to use hot-reload and DevTools during development. Fortunately, Sony provides artifacts compiled in all 3 modes on one of their GitHub repositories[14] so I ended up using those binaries. I could have also manually compiled the Flutter engine myself, but I couldn't justify the effort when precompiled binaries were available.

The archive that I downloaded from Sony contains 3 shared libraries: `libflutter_engine_debug.so`, `libflutter_engine_profile.so`, and `libflutter_engine_release.so`. It also contains a header file called `embedder.h` which is the C API that the engine exposes. Google intended for all developers that write their own embedder to use this API that they promised to keep stable and backward compatible.

This all needs to be automated by the build system. I've chosen Make because it's the build system that I know the best. I've tried CMake in the past, but I didn't have the knowledge to maintain it at that time. I've created 3 different targets: `debug_bundle`, `profile_bundle`, and `release_bundle` that let you choose in which mode you want the Flutter project to be built, and which influences which engine library to link. The build system will automatically download this archive from Sony in `/tmp`, extract it, and copy the shared objects to a directory inside the project called `deps`. It will then link against the right engine shared library depending on the target using the options `-L$(DEPS_DIR) -lflutter_engine_debug`. `-L` is used to specify the directory where the linker should look for shared objects, and `-l` is used to specify the name of the shared object without the `lib` prefix and the `.so` suffix.

I decided to copy the header file `embedder.h` into the directory `src/third_party/`. When I want to use the Flutter engine API, I just include this header file and start calling functions from it.

When I compile my compositor with target `debug_bundle` it will compile the embedder without optimizations and with debug symbols, it will automatically run `flutter build --debug` inside the subdirectory `lib/` where the Flutter project lives, and finally it will copy the embedder executable and the compiled Flutter project to the directory `build/zenith/debug/bundle`. The Flutter CLI tool doesn't support custom embedders, so we have to manually copy all the compiled artifacts into one place.

Setting up the embedder

Google provides a very simplified example on GitHub[3] on how to configure a custom embedder. The Flutter engine is essentially a black box. A few steps are required to set up the engine, but after that, it runs on its own. It will automatically spawn a certain number of threads and will periodically ask the embedder to display a frame on the screen. While the engine is running, the embedder can send

and receive information to and from the engine, like sending input events for example.

The first step to getting a working Flutter engine is to specify the path of the Dart project. Then, a graphics renderer needs to be configured for the engine to be able to draw frames. Finally, we can start the engine by calling `FlutterEngineRun`. This function will give us an opaque handle for the engine. Every time we call other functions on the engine, we need to pass this handle as an argument because the library has been designed to support multiple engine instances running at the same time, even though we aren't going to instantiate more than one.

Specifying the Dart project path

When you build a Flutter project with `flutter build --release` on Linux x86_64, the compiler will produce the executable and some additional shared objects at `build/linux/x64/release/bundle/`.

Let's describe briefly what this bundle contains:

- `executable`: The starting point of the application which calls into the embedder.
- `lib/libflutter_linux_gtk.so`: The Flutter engine and the GTK embedder bundled together.
- `lib/libapp.so`: Our project's Dart code in compiled form.
- `data/icudtl.dat`: Some data for the ICU[21] library which handles unicode.
- `data/flutter_assets/`: Fonts, images, and other types of assets required by the application.

The executable and the GTK embedder are not useful because we are going to replace them with our own, but we need the other files. While configuring the Flutter engine, we have to specify the absolute path of `libapp.so`, `icudtl.dat`, and `flutter_assets/` by populating the struct `FlutterProjectArgs` provided by the engine header file. We need to make sure the program will always work regardless of where we install the executable of the compositor and the compiled Flutter project. We can do this by querying the absolute path of the executable, and then appending the relative paths of those resources. On Linux, the symlink `/proc/self/exe` always points to the executable of the currently running program, and we can use the function `std::filesystem::canonical` in C++ to make sure we get its absolute path. The Dart VM can run in both AOT (ahead of time) or JIT (just in time) modes, and it needs to be specified in the struct. `libapp.so` only exists in AOT mode, so when we compile the compositor using the target `debug_bundle`, we don't want to populate this field in the struct. We can achieve this by conditional compilation `ifdef` macros. Compiling with target `debug_bundle` will define a macro called `DEBUG` which we can use to conditionally populate some fields in this struct.

Configuring the OpenGL renderer

Before running the engine, we need to configure a renderer for it to be able to draw frames.

The Flutter engine supports 4 rendering APIs:

1. The **software** renderer, which is the slowest one and only uses the CPU to render pixels.
2. The **Metal** API, which is GPU accelerated but it's only available on macOS.
3. The **OpenGL** API, which is a cross-platform API. It's relatively old but it's implemented by every platform.
4. The **Vulkan** API, which is a successor of OpenGL. It provides a lower, more explicit API, which can result in better performance.

I've chosen to use the OpenGL renderer because I have the most experience with this API, and the Flutter engine didn't have the Vulkan renderer at the time I started this project. The header file of the Flutter engine provides a struct called `FlutterRendererConfig` which has to be filled with the renderer type that I like to use and provide some function pointers that will serve as callbacks. As described earlier, the Flutter engine is platform-agnostic and uses inversion of control to perform platform-specific actions through callbacks.

These are the fields that I filled in `FlutterRendererConfig` with function pointers to functions I defined myself:

- `make_current`: Called when the engine wants to bind its OpenGL context.
- `clear_current`: Called when the engine wants to unbind its OpenGL context.
- `make_resource_current`: Called when the engine wants to bind its second OpenGL context for asynchronous texture uploads.
- `fbo_callback`: Called when the engine needs a frame buffer to draw.
- `present_with_info`: Called when the engine finishes rendering a frame.
- `gl_external_texture_frame_callback`: Called when the engine needs to render an external texture and needs its OpenGL ID (more on this later).
- `surface_transformation`: Called before beginning a new frame. We return a transformation matrix that is applied to every render operation.
- `populate_existing_damage`: Called when the engine wants the damage regions of the frame buffer.

In the signature of these function pointers, they all have a void pointer as the first parameter, so all these functions that I defined must have this parameter. This void pointer is a way to pass information inside these functions without relying on global variables. This user-provided data can be set when we call `FlutterEngineRun`. Every time the Flutter engine calls back, it will pass the same pointer that we provided as the first argument of the function that is being called. It's a pretty popular pattern employed in C, although you have to be careful when casting the void pointer back into the data type you provided to `FlutterEngineRun` because the compiler won't be able to statically check if the cast is

correct.

Even though I'm writing the embedder in C++ and the Flutter engine is also written in the same language, the engine's functionality is exposed through a C header because Google wanted the engine to be usable from other programming languages too. Most languages support foreign function interfaces for calling C functions, but almost none of them provide a way to interoperate with C++ because its application binary interface (ABI) is too complex.

The `FlutterRendererConfig` struct sometimes contains 2 fields per callback type, for example, `present` and `present_with_info`. The signature of the function pointer for `present` has just the user data void pointer as a parameter, but `present_with_info` provides additional information that the embedder can use to its advantage. The additional parameter is a pointer to the `FlutterPresentInfo` struct which contains the damage regions of the current render. In all cases where you have a choice between a callback with or without additional information, only one of these function pointers needs to be populated and the engine will call the one that is not null. I will talk later about the reasons for choosing `present_with_info` over `present`.

Some structures defined in the engine's header file have a field called `struct_size` which the documentation says that it must always be set to `sizeof(TheStructInQuestion)`. This field is used for versioning. Flutter has the policy to only append new fields to structures and never change or remove existing ones. This way, the size of the struct is essentially the version number of that struct. If I update the shared library of the engine without updating the header file, everything will still work correctly because the engine knows if my struct from the old header file has fewer fields than it's supposed to have. It's a smart design for backward compatibility.

Creating an OpenGL context.

To implement `make_current` and `clear_current`, we need to create an OpenGL context. We have 2 options for doing so. The first option is to call some EGL functions implemented in Mesa, a library that implements OpenGL on Linux. The second option is to use the wlroots library because it provides a renderer that abstracts away the context creation. I've chosen the second option for 2 reasons. The first one is that wlroots actually knows whether the process has been started from another compositor, or directly from a TTY console. They call this mode of operation a backend. When another compositor is running, wlroots will act as a Wayland client and automatically create a window and an EGL context associated with this window. This mode of operation behaves like the official Flutter GTK embedder. If run from a TTY console, wlroots will become the DRM master and will take control of that monitor using the KMS/DRM Linux API. It is very nice to be able to run the compositor in a window for debugging purposes without having to switch TTYs. The second reason I chose wlroots is for all the

bookkeeping it does around the Wayland protocol, as I explained in the technical background.

First, we need to create a Wayland session, a wlroots backend, and a renderer that will store the EGL context. This renderer can technically be used for simple rendering operations because wlroots provides some basic functions for drawing textures, but we are going to offload the rendering part to the Flutter engine, so we are only creating it to get a ready-to-use EGL context.

Unfortunately, wlroots is not modular enough to let me create an OpenGL renderer without also creating a Wayland server first. We are obviously going to need the Wayland server, but not before we can render something to the screen. We'll ignore configuring the Wayland server for now, and we'll come back later to it.

```
// A singleton representing the Wayland session.
display = wl_display_create();
if (display == nullptr) {
    wlr_log(WLR_ERROR, "Could not create Wayland display");
    exit(1);
}

// Connects to an existing compositor if the WAYLAND_DISPLAY environment variable is set.
// Becomes the DRM master if run from a TTY.
backend = wlr_backend_autocreate(display);
if (backend == nullptr) {
    wlr_log(WLR_ERROR, "Could not create wlroots backend");
    exit(2);
}

// Create an EGL context with OpenGL ES2.
renderer = wlr_renderer_autocreate(backend);
if (!wlr_renderer_init_wl_display(renderer, display)) {
    wlr_log(WLR_ERROR, "Could not initialize wlroots renderer");
    exit(3);
}
```

Once we have a renderer, we can get its EGL context by calling `wlr_gles2_renderer_get_egl(renderer)`. I could use this context to implement `make_current` and `clear_current`, but I decided to create another context shared with this one for the Flutter engine. The reason for this is that the engine calls these callbacks on its own managed thread. These EGL contexts can be current in one single thread at

a time and from my experience, wlroots expects its renderer's context to always be bound to the main thread. If I unbind it from the main thread and bind it to the engine's internal thread, wlroots will sometimes give me errors when it's trying to load cursor images for example. To avoid interfering with wlroots, I've decided to create another shared context derived from this one. It's very important that the new context is shared with the old one because the engine will need to render textures imported by wlroots. When a Wayland client shares a pixel buffer with the compositor, wlroots will automatically create an OpenGL texture out of it using its own EGL context. The Flutter engine must sample from this texture when it decides to render, and the easiest way to share resources between contexts is to make them shared. The OpenGL API doesn't return the memory address of a resource when it's created, but instead, it's represented by an integer ID. When 2 contexts are shared, they can both access the same resource by its ID, textures being no exception.

I'm pretty sure I could have also solved this resource-sharing another way. Instead of creating a shared context, I could have used the dma-buf Linux subsystem for sharing buffers across device drivers. The Mesa library which implements EGL and OpenGL lets you export[13] and import[19] a texture from and to a dma-buf file descriptor from one context to another, possibly even between processes. This mechanism is actually used by wlroots internally to import OpenGL textures from Wayland clients in a zero-copy fashion. I decided not to use this method because context sharing is much simpler and using dma-buf seems overkill when I want to share resources between 2 contexts in the same process.

Unsurprisingly, I had to search a bit for how to create a shared context because it's not something the average programmer has to do every day. I stitched together a 50-line function (src/util/egl/create_shared_egl_context.cpp) from code I found on multiple Internet sources. Ironically, all functions that I call in order to create a shared context return no errors and I'm able to successfully create a context even though I see printed on stderr the following error message: `amdgpu: amdgpu_cs_ctx_create2 failed. (-13)`. I've been ignoring this error ever since because I didn't notice any problems, but I'll probably have to look into it one day.

These contexts can easily be bound and unbound from the current thread by calling `wlr_egl_make_current` and `wlr_egl_unset_current`. Since the callbacks `make_current` and `clear_current` are called on the engine's render thread, the context will be bound to that render thread. The engine can work with a single context, but the documentation of the header file says that it prefers having 2 contexts: one for the main rendering operations, and the other one for asynchronous texture uploads. My best guess is that the second context is used when loading compressed images on the disk into OpenGL textures. This can be an expensive operation and would block the command pipeline if done on the render thread with the first context. I've decided to give it one more shared context shared with the other 2 because it was easy to do. I've defined a callback function and filled the field `make_resource_current` with its

function pointer. The engine will call this function whenever it wants to bind this second context, and the implementation is pretty much the same as `make_current` but binding the second context instead.

With context binding in place, the engine can now use the graphics card through the OpenGL API, but it can't render frames just yet. It needs a frame buffer to draw into, with a pixel format compatible with the display output, and it obviously needs the size of this frame buffer because widgets have to be laid out and rendered based on size constraints. The engine can't query this information by itself while still remaining platform agnostic, so the `fbo_callback` callback must be implemented.

Output detection

In `wlroots`, I can register a callback that will run on the the main event loop when a new output is detected. Not so long after the compositor starts, this callback is called once per output connected to the computer. In order to turn the screen on, we must enumerate all modes. Modes are a combination of resolution and refresh rate. We set the mode with the highest resolution and refresh rate because we want the best experience from the start, and then we commit the changes. All this is done in a couple of lines of code using abstraction layers provided by `wlroots`.

In 2015, the Linux kernel enabled atomic mode setting by default. This allows userspace to update the hardware graphics state atomically, eliminating weird graphical glitches while switching resolutions for example. Multiple changes are queued up, and when we commit, all these changes are applied at once. We are using `wlroots` as an abstraction layer, and if the kernel is old enough to not support this feature, it will just apply these changes sequentially.

Swapchain implementation

`wlroots` once again provides abstractions for obtaining a swap chain containing frame buffers ready to be given to the engine to render into. Calling the function `wlr_output_init_render` will create a so-called `wlr_renderer` with a swap chain. Unfortunately, this swap chain, while being triple-buffered, is too simple. `wlroots` expects the rendering to be done sequentially: acquire an unused frame buffer, render into the frame buffer, flip the frame buffer to the screen, and repeat. The Flutter engine doesn't work like that. There is no API to force the engine to start rendering a frame when we want. I've tried very hard to make the default swap chain work with Flutter but I never got it to work flawlessly, so I decided to create my own swapchain. The requirements are:

1) When the engine asks for an unused frame buffer, the swap chain can provide one immediately.
2) The embedder can ask the swap chain for the most recent frame buffer that Flutter completed when a page flip has to occur.

3) Be thread-safe and allow these requests to come from different threads because engine callbacks are called from internal threads.
4) Keep track of which frame buffer is unused, which is being flipped to the screen, and which one is currently being rendered by Flutter.

I've chosen to implement a triple-buffered swap chain because it's the most flexible one. With VSync enabled, it performs exactly the same as the double-buffered swap, the only difference is that there is one idle frame buffer never being used. Additionally, it's nice to be able to disable VSync in case I want to debug something related to frame timing. Triple buffering opens the door to more advanced frame timing techniques. One such technique is sticking to VSync as long as the GPU can produce frames within the refresh cycle, otherwise, ignore VSync and keep the GPU busy to try to recover from a frame that took more than it should. I will not implement these techniques because I haven't experimented with them yet.

Unfortunately, `wlroots` makes it unnecessarily difficult to write your own swapchain. Other than creating a `wlr_renderer`, `wlroots` has no public API to easily allocate some frame buffers in a pixel format that matches the display. At this moment, I had 2 options. The first option was not using `wlroots` for anything graphics related, and manually allocating some frame buffers using the Mesa graphics library. This means I would lose the ability to run my compositor as a Wayland client inside another compositor because it's strongly tied to `wlroots` abstractions. The second option was to copy some code from `wlroots`' source code in order to resurface some private APIs for my own personal use while staying within the bounds of `wlroots`' abstractions. The disadvantage of this method is that I would depend on implementation details of `wlroots` which could change with any new version. I've chosen the second option because I find it invaluable to be able to run the compositor in a window. Besides, `wlroots` won't change versions on Ubuntu LTS, which is the distribution I'm compiling for, for dependency stability reasons. The file `scr/wlr/wlr_helpers.cpp` contains functions and structs copied from the private API of wlroots.

The swapchain is implemented in the class `SwapChain` in `src/swap_chain.tpp`. 4 frame buffers that make the swap chain are passed into the constructor from the outside to avoid coupling it with other `wlroots` abstractions like `wlr_output`, `wlr_renderer`, and `wlr_allocator`. The lifetime of these frame buffers is now bound to the lifetime of the swapchain by wrapping them in unique pointers. `std::unique_ptr` is a smart pointer that automatically deletes the object it points to when it goes out of scope. It ties into the "Resource acquisition is initialization" (RAII) programming idiom in C++ which helps avoid common memory leaks and double frees. I don't know why but if the swapchain is created with 3 buffers instead of 4, there will occasionally be visual glitches on the screen. My theory is that the frame buffers cycle too fast between being rendered by the Flutter engine and being flipped to

the screen, but I didn't investigate any further. Adding one more buffer to the chain fixed the problem.

The swap chain has 4 variables, one per frame buffer:

- `latest_buffer`: The buffer that was most recently rendered by the Flutter engine and ready to be flipped to the screen.
- `write_buffer`: The buffer that is currently being rendered by the Flutter engine.
- `read_buffer`: The buffer that is currently being flipped to the screen.
- `delay_buffer`: The additional buffer I introduced to make the other buffers stay in the chain one more frame before being written to by the engine.

It provides 3 methods:

- `start_write`: Returns a pointer to an available frame buffer ready to be rendered.
- `end_write`: Signal the swap chain that the frame buffer returned earlier is ready to be flipped on the screen.
- `start_read`: Returns a pointer to the latest frame buffer rendered by the Flutter engine in order to flip it to the screen.

These 3 methods use `std::swap` to swap the frame buffers between these 4 variables in order to change their role. For example, the `end_write` will swap `write_buffer` with `latest_buffer`, while `start_read` will swap `latest_buffer` with `read_buffer` if `end_write` was called earlier.

Implementing the `fbo_callback` callback is now trivial. We just have to access the swap chain, call `start_write` to get a new frame buffer, and return the frame buffer's OpenGL ID. The engine will use this ID to bind the frame buffer and render to it.

We can now also implement `present_with_info`. The tricky part is that the engine doesn't call this function when the GPU has finished its work, but after it submits all the commands to the GPU. If we were to flip the frame buffer to the screen right away, the user would probably see an incomplete frame with flickering UI elements. We need to wait for the GPU to finish rendering before calling `end_write` on the swapchain. The OpenGL API provides a function called `glFinish` which essentially blocks the thread until the GPU is done with all the work, so I just called this function before calling `end_write` and it solved all flickering problems. I could have probably done something better than throttling the engine's render thread, but since the engine must produce frames only 60 times per second, this thread will sleep most of the time, so I'm not that concerned about blocking it.

The Flutter engine now has all the necessary callbacks implemented to be able to render frames. In order to see something on the screen we need to commit a new frame buffer at the refresh rate of the monitor.

Presenting a frame

I've talked earlier about how we can register a callback using `wlroots` to detect available outputs. `wlroots` represents outputs as `wlr_output` objects. These output objects contain information about the display like name, model, and refresh rate, but it also has a collection of signals in an anonymous struct called events. These signals basically implement the observer pattern by storing a list of listeners with callbacks, and when there is an event emitted on a signal, all its listeners are notified and their callbacks are called. The `wlr_output` has a signal called `frame` which is emitted every time the output is ready to display a new frame, at the cadence of the refresh rate. In order to display new frames, we can register a callback to this signal, take the latest frame buffer from the swap chain using the `start_read` method, attach it to the output, and commit it.

The render cycle is now complete. The engine can ask for a new frame buffer, render into it, and 60 times per second the newest rendered frame is flipped to the screen. By default, the engine will render as fast as possible saturating the GPU. It will spit out hundreds of frames per second, but only 60 of them will be displayed on the screen in one second. This wastes a lot of GPU resources and we can do better by enabling VSync.

The engine header file has some functionality to enable VSync and give a hint to the engine when it should start rendering a new frame. First, we need to define the `vsync_callback` callback and fill the function pointer of the `FlutterProjectArgs` struct. When this function pointer is non-null at the time the engine is started, the engine won't generate frames as fast as possible anymore. Instead, through a parameter of this callback, it will give us an integer that Google calls "baton". This baton needs to be given back at some point to the engine using the `FlutterEngineOnVsync` function defined in the header file. When this function is called, the engine will understand that it should start rendering a new frame soon. It's also entirely possible that the engine won't start rendering right away if nothing has changed on the screen, and only start rendering when the user interacts with some widgets. When the engine is done rendering, the frame is flipped to the screen as usual, and the engine will give us another baton for the next frame.

Knowing all this, I've implemented `vsync_callback` to just remember the baton. When a page flip has occurred and I have a baton that I haven't given to the engine yet, I will call `FlutterEngineOnVsync` to notify the engine that it can draw a new frame.

Platform channels

It's nice to be able to have widgets on the screen, but the end goal is to be able to display windows and popups which requires communication between Dart and the embedder through platform channels.

Codecs

The header file of the engine has an API for sending and receiving messages from Dart through platform channels, however, these messages are required to already be encoded using one of many formats that Google defined. Unfortunately, Google has left the encoding and decoding of platform messages out of the scope of the engine, and that's something the embedder has to implement. In theory, I see no reason why the engine couldn't provide platform-agnostic implementations of these message codecs, but Google probably knows the reason behind it.

Apart from being able to create my own platform channels for my own uses, there are many predefined platform channels in Flutter for integrating with the underlying platform (see [15]) like text autocomplete, clipboard integration, and more. I've noticed that low-level features like VSync are exposed through the C header file because it's related to the Flutter engine, while other features are exposed through platform channels because they are related to the Flutter framework.

There are multiple ways to encode a message and some of these predefined platform channels expect their messages encoded with a different codec than others. For example, the "flutter/textInput" platform channel is a method channel that exposes control for interacting with the system on-screen keyboard and expects to receive messages encoded in JSON format.

These codecs and documented very briefly on the official Flutter documentation, but I needed far more low-level details if I wanted to implement them. I didn't find this information anywhere on the website, so I had to dig in the official Flutter engine repository for answers. This repository obviously contains the source code of the engine, but not only. It also contains all the embedders for all their supported platforms. All these embedders also use the same header file to interact with the engine, but Google has written a ton more abstractions on top of it for all sorts of bookkeeping, and every embedder also contains an implementation of platform message codecs. After further analysis, I saw that their codecs implementation was very well modularized from the rest of the abstractions, using mostly the C++ standard library and very few includes to other header files outside the directory where all the codecs reside. For this reason, I've decided to copy their implementation into my compositor and use their code instead of reimplementing all the codecs from scratch. In total, I copied a few thousand lines of code from their repository and I put it under `src/third_party/platform_channels`. I have made very small modifications to this code because all the includes had absolute paths that made sense only in the context of the Flutter engine repository. I've also removed some files that were unnecessary for my use case because they were tying into the other abstractions of Google's embedders and I only wanted the codecs implementation, nothing more.

At this point, I was able to send all kinds of messages: numbers, strings, lists, and hashmaps from C++ to Dart using different codecs, but I wasn't able to receive responses from Dart, because that would

require the Flutter engine to somehow notify me when there is a pending message.

Task runner

The Flutter engine has a concept called task runner which is a queue of tasks that have to be executed at a target time on some thread not controlled by the Flutter engine. When the engine has a message or response coming from Dart, it will try to post a task to the platform task runner, which is on the same thread where the engine was started.

The Flutter engine cannot magically schedule work on the event loop of the embedder. If the platform task runner hasn't been configured, all messages from Dart will simply be dropped, so we have to write some code to store tasks posted by the engine and execute them on the main event loop when the target time of each task expires.

The implementation of this task runner is found at `src/flutter_engine/task_runner.cpp`. It's a class containing a priority queue storing `FlutterTask` objects paired with the target time when the task should be executed, ordered by the target time. Tasks that will have to be executed sooner will have a higher priority in the queue. When we set up the engine, before running `FlutterEngineRun`, the struct `FlutterTaskRunnerDescription` has to be filled in order to tell the engine what callback to execute in order to post tasks on the platform task runner. This callback will be called from any internal engine thread with the `FlutterTask` to be posted and the target time in parameters. The callback will simply push this task onto the queue.

Since we have a queue of tasks that are going to be executed on the embedder's thread somewhere in the future, we need a way to efficiently wake up the thread when the target time of the earliest task expires without using a busy loop that continuously checks if it's time to execute a task. We can do that by using timerfd, and attaching the file descriptor to our event loop backed by epoll. When a new task is put in the queue by the engine, we can configure a timerfd file descriptor to become readable at the target time of the earliest task in the queue. If the event loop of the embedder is waiting for events to happen, the kernel will wake up the thread and we can execute all the expired tasks. To execute a task we have to simply run `FlutterEngineRunTask` with the `FlutterTask` whose target time expired.

Even though we set up timerfd to wake up the thread sometime in the future, the kernel will most probably not wake it up exactly at that time, but some microseconds after. We have to take this into consideration when checking the queue for expired tasks because, in that extra time interval, multiple tasks might have expired. Scheduling timerfd in the past is not a problem because the file descriptor will instantly become readable once again, but it's more efficient to just execute all expired tasks at once and make sure to only schedule timerfd in the future. If no tasks are left in the queue, the timer will simply disarm itself and will become rearmed when a new task is put in the queue.

The queue is protected by a mutex because the engine puts tasks in a queue from its internal thread, and these tasks are drained from the queue in the embedder's thread. When the time comes to execute expired tasks, we could lock the mutex for the whole duration of the method, but that would block the queue for longer than it should. Calling `FlutterEngineRunTask` takes an unspecified amount of time depending on what task the engine wants to run. The smarter thing to do is to lock the mutex, collect all the expired tasks in a new vector, unlock the mutex, and then execute the expired tasks. This way, the queue is not blocked while tasks are executing, and the queue is only blocked for the time it takes to iterate it and collect expired tasks.

Once the task runner is implemented, the last piece of the puzzle is to let the engine know what function to call when we get platform messages from Dart. When we configured the engine, there was a function pointer called "platform_message_callback" in the `FlutterProjectArgs` struct that we left on null. This callback, when not null, will be called by the engine on the embedder's thread when a message is received from Dart. In the parameters, we get the channel name where the message came from, the message itself as a byte array, and the length of this byte array. This message is encoded using one of the codecs, but we can use the decoder we copied from Google's repo to decode it and obtain the message.

The code we copied from Google also contains a dispatcher class where we can register a response callback for a specific platform channel name. The dispatcher will know what codec was used to encode the message. It will decode the message and call my message handler with the decoded message.

Registering external textures in Flutter

We have the infrastructure to send and receive messages from Dart which we are going to need in order to transfer simple information about windows and popups like the window title or the position of a popup, but this mechanism is not appropriate for transferring large amounts of data like textures. A single frame of an application can be megabytes in size, and sending this data through platform channels would be very inefficient. The Flutter engine has 2 other mechanisms for making textures available to the engine.

The first one is by letting the Flutter engine have the OpenGL identifier of the texture, and it can draw it on the screen using the Texture widget. Instead of copying the texture, the engine can sample directly from the texture. The second one is to use platform views. Instead of letting Flutter do the draw calls for rendering the texture, the engine will delegate this task to the embedder. This second method is used on Android and iOS when native widgets have to be displayed inside a Flutter app. In my case, I chose the first method out of simplicity because I just wanted to have some textures displayed on the screen and I couldn't justify using platform views for this simple use case when the

first method was created exactly for this reason.

The Flutter engine header contains 3 functions and 1 callback to handle texture registration. When I want Flutter to be aware of a texture, I have to call `FlutterEngineRegisterExternalTexture` with a texture identifier. However, this texture identifier is not the OpenGL ID of the texture. It's just a unique number that the engine will later associate with a texture. I decided to generate incremental numbers starting from 1 every time I had to register a texture in Flutter. Now that a texture is registered, the unique texture number can be used from the Flutter framework in Dart. There is a special widget called `Texture`[16] that takes this unique number in its constructor. When it's time to render a new frame, Flutter will ask the embedder for the OpenGL ID of all textures referenced by the `Texture` widget by calling the `gl_external_texture_frame_callback` function pointer. We define a function and set this function pointer when we configure the OpenGL renderer for the Flutter engine. The embedder remembers the association between the unique texture numbers and the OpenGL IDs using a hashmap, and every time the engine asks for a texture ID, the callback will simply look up inside the hashmap and return the OpenGL ID of that texture.

Welcoming Wayland clients

Now that we have the engine configured properly, we can finally start playing with Wayland and start the quest of displaying our first window.

First, let's see what are the steps that a Wayland client does in order to display a window. A Wayland client needs to connect to the server through the Unix domain socket exposed by the compositor. We already started a Wayland server with wlroots because we wanted to instantiate a renderer. The socket will be created under `/run/user/1000/`, 1000 being the user ID under which the compositor was started. The name of the socket will usually be `wayland-0`, but if there is already another Wayland compositor running, it will try the next name `wayland-1`, and so forth until it can create a socket. When a client wants to connect to a compositor it will look under this directory and will usually connect to the first Wayland server that it finds. In normal circumstances, this is not a problem because a user won't run multiple compositors at a time, but I like to run my compositor nested inside my usual desktop environment (KDE Plasma) because it's more convenient to debug. In this case, if I want to force a client to connect to my compositor, I have to set the environment variable `WAYLAND_DISPLAY` to the name of the socket that my compositor created. Another way would be to set this environment variable inside my compositor and then fork it to spawn a client. This is normally what is done in "production" because processes inherit the environment variables of their parent, so if the compositor provides an app launcher, all the applications will connect to it. However, for debugging purposes, I prefer to manually spawn an application from the terminal and set the environment variable myself

because I can also see the logs on stdout which can be sometimes useful.

Next, the Wayland client will ask the compositor for all the protocol extensions it supports. The core Wayland protocol must always be supported by compositors because it provides the core concepts that all other protocol extensions build upon, like defining what a buffer or a surface is. Luckily, wlroots takes care automatically of all this handshake process.

Listening to surface creation

Once the client has successfully connected and knows what protocol extensions are supported by the compositor, it will usually start sending some requests to the server right away in order to open a window. These are the requests and responses that the compositor has to handle at a minimum in order to satisfy the client. I've denoted the requests from the client to the server with a `->` and responses with a `<-`.

1) -> create a surface
2) <- ID of the newly created surface
3) -> create an xdg surface
4) <- ID of the newly created xdg surface
5) -> create an xdg toplevel surface
6) <- ID of the newly created xdg toplevel surface
7) -> commit the changes
8) <- configuration finished + serial number
9) -> acknowledge the configuration finished with serial number
10) -> create a shm pixel buffer of a certain size and pixel format
11) <- ID of the newly created buffer
12) -> attach the buffer to the surface
13) -> commit the changes

Wayland is an object-oriented protocol. Every request must be made on an object that both the client and the compositor know about. Naturally, the protocol defines a few global objects whose IDs are shared with the client during the handshake. When the client wants to create a surface, it will send the request `create_surface` on the `wl_compositor` global object. The compositor will simply reply with the ID of the newly created surface and both parties can use this ID to refer to the surface in future requests or events. A Wayland surface is an object representing a rectangular area that may be shown on the screen and receive input events.

Wlroots allows us to hook into this client request by registering a callback. Global objects exposed in the Wayland protocols are also accessible in C++. In the case of `wl_compositor`, wlroots puts us

at disposition an instance of the struct `wlr_compositor` which contains one signal object per client request. To hook into the surface creation, we can register a callback for the `new_surface` signal. When a client wants to create a surface, the callback gets called with a `wlr_surface*` object as a parameter. This object is a wrapper around the Wayland surface created by the client. wlroots will manage the lifetime of its own objects by itself, and callbacks of this kind allow us to attach additional information to these objects that can be later retrieved by the compositor. Pretty much all wlroots managed objects have a void pointer called "data" which can be used for any purpose. I personally chose to store in this field a pointer to a heap-allocated instance of `ZenithSurface`, a class I created that stores the callback registrations of further client requests on this surface, a unique ID that I call "view_id", and the pixel buffer size of the previous commit (initialized at (-1, -1) at the start). wlroots doesn't provide an API to retrieve any `wlr_surface` by its ID, so I decided to store these surface pointers in a hashmap in the singleton `ZenithServer`, a class I created that centralizes the state of the entire compositor. It will come in handy later to be able to retrieve any surface by its view ID.

A surface on its own is useless. The compositor won't know what to do with it, and for it to become useful, the client must give this surface a role, for example, a window, a popup, or other roles defined in other protocol extensions. The core Wayland protocol provides another global object called `wl_shell_surface` which lets a client extend the capabilities of a Wayland surface, to become a window or popup, but it has been deprecated in favor of a protocol extension called `xdg_shell` which provides a more complete featureset. Figure 8 shows the hierarchy of surfaces in Wayland.

The `xdg_shell` extension defines a global object called `xdg_wm_base` where the client can extend the capabilities of a regular surface by creating an `xdg_surface` object referencing the original surface. It's in the same spirit of inheritance where you can extend a class to specialize it, but here, the client first creates an instance of the most generic type, then specializes it more and more by creating objects that hold additional properties. In this case, the client first creates an `xdg_surface` having properties that both windows and popups have. Then, it will create an `xdg_toplevel` referencing the previously created `xdg_surface`. Toplevel is a window in Wayland lingo.

Once again, wlroots allows us to hook into the creation of these objects, and possibly attach some additional state to them. I kept the same pattern of creating a class for the xdg surface called `ZenithXdgSurface` and for the toplevel called `ZenithXdgToplevel` that store the callback registrations of their particular client requests. Equally, I created 2 more hashmaps for storing these instances by the view ID of their base surface.

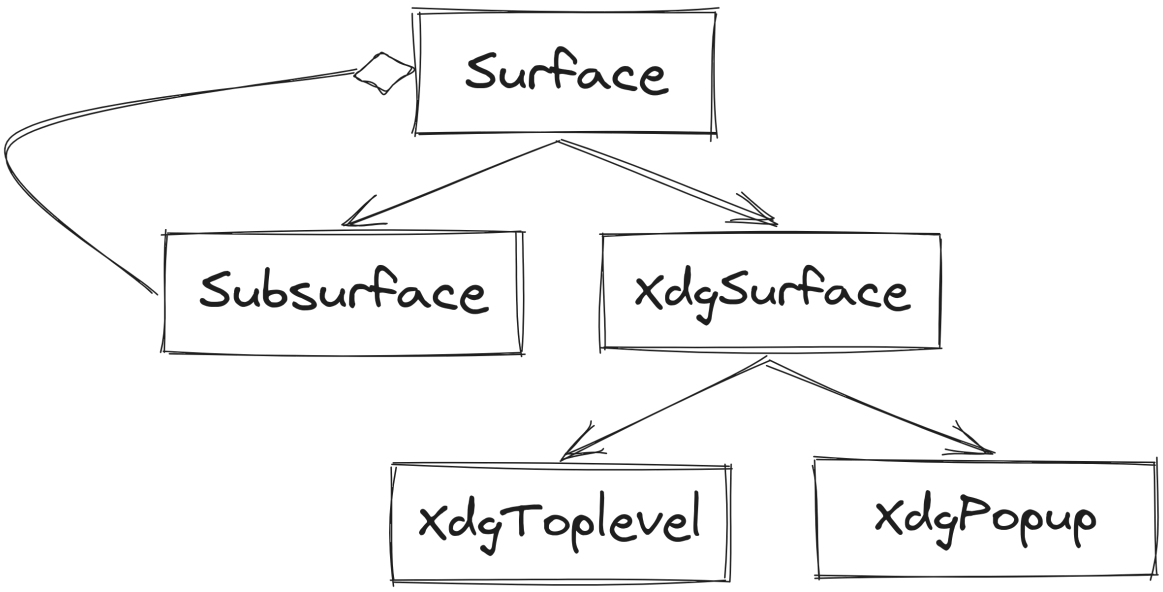

Figure 8: Hierarchy of surfaces in Wayland. Any type of surface can have a tree of subsurfaces that get overlayed on top, a good use case being subtitles in a movie.

Listening to commits and serializing the state for Flutter

Now, the Wayland protocol demands that the client commits these changes by sending a `commit` request on the Wayland surface that was originally created. The compositor will send back a `configure` event on the `xdg_surface` object which contains a serial number. Serial numbers are used in the protocol to identify a specific request or event. When the client acknowledges the configuration event, it has to send back the serial number of the configure event it received. This way, the compositor knows what event the client responds to. When sending the configure event, the compositor can also ask the client for the preferred size of the window, but the client has no obligation to respect this size.

Commits are probably the most important client requests because they mark the end of a series of changes that have to be applied all at once and it is the perfect moment to send the state of our committed surface to Flutter. To do that, we first need to hook into the commit request by defining a callback. When wlroots calls this callback, the pending state accumulated before the commit is already applied to the current state of the surface. This is the list of properties that have to be sent to Flutter through a platform channel:

For the surface:

- `view_id`: The unique ID of the surface.

- `texture_id`: The ID of the registered texture associated with the surface.
- `role`: The role of the surface, whether it's a window or a popup.
- `width`: The width of the surface.
- `height`: The height of the surface.
- `scale`: The scale factor of the surface, used for HiDPI displays.
- `input_region`: The region of the surface that should receive input events.

For the xdg surface:

- `mapped`: Whether the surface is mapped or not. A mapped surface is a surface that has a role and a pixel buffer attached to it.
- `geometry`: The rectangle containing the visible bounds of the surface excluding shadows and margins.

For the toplevel surface:

- `decorations`: Whether the surface should have decorations (title bar, close button, borders) drawn by the compositor.
- `title`: The title of the window.
- `app_id`: A string uniquely identifying the application, usually

The codec I've used to encode these properties before sending them through the platform channel is the standard codec, a binary codec more efficient than JSON defined by Google. In the file `src/flutter_engine/message_structs.hpp`, I have written some DTO structs (data transfer objects) that will hold all the properties I want to encode. The function `commit_surface` from the file `src/flutter_engine/embedder_state.cpp` will take these DTOs, encode them using the standard codec, and send them to Flutter through the platform method channel `platform` and method name `commit_surface`.

From Dart, in order to receive these messages, I just have to set a method call handler on the platform channel with the same name. When a commit message is sent, Dart will call this handler. The first argument is a dynamic type, but it can be casted to a `Map<String, dynamic>` and I can read all the properties I sent from C++. All this deserialization code can be found in the `lib/platform_api.dart` file.

Storing Wayland surface state in Dart

After casting back all the data I received into their own primitive types, I have to store this state somewhere Flutter widgets can read it. Additionally, widgets that depend on this state should automatically be rebuilt and the UI should reflect these changes. This is something that state

management libraries like Provider or Riverpod do very well. I've used Provider in the past and I've had a generally good experience with it, but I decided to try out something new, which is why I went with Riverpod. This library was created by the same guy who created Provider with the goal of fixing the shortcomings of Provider.

My Flutter state is architectured as follows. I have one Riverpod family for the surfaces, one for the xdg surfaces, and the other one for the toplevel surfaces. A family is a way to create a provider that can be parameterized. In my case, I want to create a provider for each surface, so I can pass the view ID as a parameter. If the provider referenced by the view ID doesn't exist, it will be automatically created. In the file `ui/common/state/surface_state.dart`, there is an immutable class called `SurfaceState` created with Freezed which represents an immutable model of a surface. The family of surfaces is represented by the class `SurfaceStates` and contains a method called `commit` which basically shallow copies the state of the surface referenced by the view ID and updates the fields with new values. When a provider modifies its state, Riverpod takes care of rebuilding all the widgets that depend on that provider, and the UI automatically updates to reflect the new committed state of the surface. A widget can depend on a provider by doing `final state = ref.watch(surfaceStatesProvider(viewId))`, very similar to how it works in the Provider library. The `watch` method will rebuild the widget when any field of the surface state has changed, but it's also possible to listen to specific fields for optimization purposes, for example, `final Rect inputRegion = ref.watch(surfaceStatesProvider(viewId).select((s) => s.inputRegion))`. I personally always use this optimization in order to avoid rebuilding widgets more often than necessary.

At this point, the compositor knows that the client has created a window, but it can't show it to the user because the client hasn't yet specified what buffer of pixels it wants to display. Transmitting this pixel buffer over the Wayland socket would be very inefficient the client would have to send a few megabytes of data, 60 times per second on a 60 Hz monitor. The core Wayland protocol defines a global called `wl_shm` which lets Wayland clients share pixel buffers with the compositor through the shared memory API of Linux, no copying involved. This object lets the client create pools of memory from which it can allocate pixel buffers of a certain width, height, and format, as `wl_buffer` objects.

How the client draws into this buffer is entirely up to it, whether it uses the GPU or the CPU, as long as the data gets shared with the compositor. Once the client has created a `wl_buffer`, it can attach it to the surface by sending an `attach` request followed by a `commit`. The motto of Wayland is to be frame perfect, and the commit request exists to make sure the compositor atomically applies all the accumulated changes that have been made to a surface since the last commit. This has the advantage of preventing visual glitches where the client makes a number of requests over a period of time, but the compositor has to generate a frame before waiting for the client to finish sending all the requests.

The way this is implemented in wlroots is by double buffering the state of a surface. Client requests will modify the pending state of the surface, and when a commit arrives, the pending state and the current state pointers are swapped. When the compositor has to generate a frame, it will always read the properties of the current state.

While clients can send all their pixel buffers through `shm`, all compositors nowadays do hardware acceleration rendering using the GPU and if the compositor wants to use this buffer, it will have to copy it into VRAM as a texture. Another copy will have to happen if the client also uses the GPU for rendering because it will have to copy the pixel buffer from VRAM to RAM before sending it to the compositor. Luckily, all modern applications are hardware accelerated, and there is a Wayland protocol extension called `zwp_linux_dmabuf_v1` which lets clients share dmabufs of textures that are already on the GPU with compositors, without any copying involved. In both cases, whether the buffer is a `shm` or `dmabuf`, wlroots will automatically extract the necessary information (width, height, pixel format, and the data itself) and create OpenGL textures ready to be used. In the commit callback when we see that a client has a buffer attached, we register the texture to the flutter engine and we make sure the texture ID is being sent through the platform channel.

The compositor now has all the necessary information to display a window to the user. A surface that should be shown is "mapped", and when it shouldn't be shown anymore, it's "unmapped". Wayland defines a mapped surface to be a surface that has a role and has a pixel buffer attached to it. This property, however, is only defined in the documentation of the protocol and not in the protocol itself, but it clarifies when a surface should be shown to the user. For convenience, wlroots provides a signal to listen when a surface is mapped as if it were a client request. Previously, I simply listened for this signal and sent the state of the boolean through a platform channel which worked relatively well, but I encountered a bug in wlroots where this signal would sometimes not trigger and some surfaces were not visible even when they should. I decided to manually read the state of this boolean property in the commit callback of the surface and send it together with the rest of the surface state, which solved my problems.

Using the Texture widget to display windows

The Texture widget is simply constructed by giving it a texture ID and Flutter will render the texture on the screen at the position of the widget. This widget must be put inside a SizedBox with the dimensions of the texture because apparently, the Texture widget doesn't know the dimensions of the texture, but since we sent all this additional data through the platform channel, we can construct the SizedBox widget with the correct dimensions. I'm expecting a Wayland surface to be represented by many more widgets than simply a texture (input detectors, subsurfaces), so I've created a widget called

Surface which represents a Wayland surface. I've also created 2 more widgets called XdgToplevelSurface and Popup that use Riverpod to listen to changes in state and rebuild themselves accordingly. An XdgToplevelSurface contains a Surface widget as one of its children, but will maybe add a title bar to make it look like a window and be able to drag it around and add some nice animations when the Wayland client unmaps or destroys the window.

Window resizing

When a Wayland client generates a new frame for a surface, we can choose to register a new texture ID in Flutter or tell Flutter that a new frame is available for the already registered texture. At first, I decided to register one single texture ID per surface, and every time the client decided to generate a new frame, I would tell Flutter that a new frame was available. When the window is resized, the commit callback will send the new size through the platform channel but still use the same texture ID. This worked great until the Wayland client decided to resize its window. Flutter does widget layout and rendering on 2 separate threads, and sometimes the render thread is behind the layout thread by 1-2 frames. The widget tree might already be updated with the new data, but the renderer still hasn't finished rendering the previous frames. When a window is resized, this would cause a visual glitch lasting 1-2 frames where Flutter would use the new texture with the old size, and the texture would look stretched. I fixed this issue by registering a new texture ID every time the dimensions of the surface change. This way, a texture ID will always have the same dimensions, and this visual glitch is avoided.

Explicit synchronization

Implicit synchronization is when the GPU driver automatically synchronizes operations on textures shared across processes. This behavior can be both good or bad depending on the point of view.

Simple clients may benefit from it because all synchronization logic is done by the GPU driver, simplifying the code. What you lose is some performance because the driver may over-synchronize some operations because it doesn't have the full picture of what the client is trying to do with it.

From the compositor's point of view, it's another story. Textures shared between the application and the compositor through dma-buf are implicitly synchronized. If a client shares its dma-buf with the compositor before the texture is fully rendered, the OpenGL API will implicitly block the rendering until the texture is ready to be used, if the compositor issues an OpenGL command that tries to use the texture. Simple compositors will benefit from this behavior because they don't have to write more code to make sure they don't use a texture that is not ready yet, but there is a catch to it.

Imagine running a heavy game that issues draw calls which take a while to complete. The compositor

will inevitably have to wait for the game to complete its current frame before it can continue rendering. This can essentially limit the framerate of the compositor to the framerate of the heaviest client, making the whole user interface laggy, or barely usable.

I've been fascinated by this topic because I've seen issues on GitHub/GitLab for other desktop environments that describe this exact situation with a video game in windowed mode. Information is little and scattered on this topic, but fortunately, I have found some excellent resources that helped me understand the problem better. This kernel mail thread [29] and this article written by Collabora [11] were extremely helpful in understanding how the synchronization of textures works on Linux.

I've made an attempt to do explicit sync, and while it works, this was more of an experiment because I haven't encountered any issues with implicit synchronization, so I can't assess the benefits of what I've done.

Linux 5.20 has added a new ioctl request called `DMA_BUF_IOCTL_EXPORT_SYNC_FILE` that allows userspace to extract dma fences from dma-bufs as sync files, which are essentially file descriptors that become readable when the texture behind that dma-buf is ready to be used. These file descriptors can be attached to an epoll-based event loop and asynchronously notify the thread when the texture is ready.

The only place I could find how to use this new ioctl request was in the source code of Mesa, the library that implements the OpenGL and Vulkan graphics libraries. In the windowing system integration part of Vulkan [12] I found how they use this ioctl around the swapchain. Even though I'm using OpenGL and not Vulkan, nothing stops me from doing exactly the same thing because all I want is to handle synchronization manually instead of letting OpenGL do it implicitly for me.

So that's what I did. When I got a new dma-buf from a Wayland client, I extracted the sync file and defined a callback that registers the texture in Flutter when the file descriptor becomes readable. That way, Flutter will only get textures that are always ready to be sampled.

I have only talked about what happens when Wayland clients share textures through dma-buf, but let's not forget that clients can choose to share pixel buffers through shared memory on RAM. In that case, the compositor (done automatically by wlroots) creates an OpenGL texture by copying the buffer from RAM to VRAM. When other OpenGL commands are issued in the same context that the texture was created, OpenGL will implicitly synchronize the operations as always. But our case is different because Flutter renders using its own GL context. The OpenGL ES specification clearly states that textures shared across GL contexts of the same process are not implicitly synchronized and the developer must take care of it. If we don't take care of it, Flutter might try to sample from a texture when the copy from RAM to VRAM hasn't finished yet, causing visual glitches. The solution is to only register the

texture in Flutter when the copy has finished.

OpenGL provides sync objects[28] that lets the application synchronize CPU and GPU operations by putting a fence in the command queue. The function `glClientWaitSync` can then be used to block the thread until the fence is signaled. I didn't want to use this function because that would block the thread hosting the Wayland server while the pixel buffer is copied to VRAM. To be honest, I didn't measure the amount of time the thread stays blocked, but I still thought it's better to avoid blocking it in order to stay responsive to Wayland requests from all other clients. The core OpenGL specification does not provide a way to asynchronously notify a thread when the fence is signaled, but there is an EGL extension called `ANDROID_native_fence_sync`[10] that allows exporting sync objects as file descriptors. Its name starts with "ANDROID" because Google created Android with explicit synchronization in mind, and they needed a way to share sync objects between processes. This extension has since been implemented in Mesa and is also available on any Linux distribution.

What I did was create a sync object by calling `eglCreateSyncKHR`, and then I used the function `eglDupNativeFenceFDANDROID` to extract the sync object as a file descriptor. I then attached this file descriptor to the main event loop powered by epoll. When I get notified, I know I can safely register the texture in Flutter, without needing to call `glClientWaitSync` and block the thread.

Handling raw input

Let's see how we can listen to touch, keyboard, and mouse input and forward it to Flutter and Wayland clients.

The `wlr_backend` object we instantiated in the main function has a signal called `new_input` which is raised when a new input device is connected. We can register a callback to this signal. With the `wlr_input_device` object that is passed in the callback, we can query the device type and register other callbacks for each type of input. Since every time you register a callback you have to instantiate a `wl_listener`, we'll have multiple ones every time a new input device is connected, and those should be stored somewhere. Also, when a device input is disconnected these listeners should be freed from memory. For structural reasons, I've decided to create a `cpp` file and a class for each type of input device. The class would store and set up all the listeners with callbacks defined at the end of the file. The callback `new_input` would instantiate the class corresponding to the device type and store it in a vector just to keep the object alive.

Touch input

The files `src/input/touch_input_device.hpp` and `src/input/touch_input_device.cpp` contain the class `TouchInputDevice`. When a new touch input device is connected, the constructor of this class

will be called and will register callbacks to 4 different events:

1) Touch down - A finger came in contact with the screen.
2) Touch motion - A finger moved while being down.
3) Touch up - A finger was lifted up.
4) Touch cancel - Similar to a touch up event, but the compositor should return to the state before the touch down event occurred without actuating the event.

Sending these events to the Flutter engine is very easy. Its header file contains a function called `FlutterEngineSendPointerEvent` which we can call after filling the struct `FlutterPointerEvent`. This struct aggregates all sorts of attributes for all types of input devices (touch, trackpad, mouse, and stylus), but only a few of them are required for touch devices:

1) `phase` - A value of the enum `FlutterPointerPhase` which can be `kDown`, `kMove`, `kUp`, or `kCancel` for touch events.
2) `timestamp` - The current time in microseconds.
3) `x` - The X coordinates of the touch event in pixels.
4) `y` - The Y coordinates of the touch event in pixels.
5) `device_kind` - The type of device that generated the event. In our case, it's `kFlutterPointerDeviceKindTouch`.
6) `signal_kind` - Some devices like mice might generate scroll signals without moving the pointer. We're setting this to `kFlutterPointerSignalKindNone` because touch input is just fingers interacting with the screen and nothing more.
7) `device` - A unique device ID that Flutter uses to differentiate between fingers on the same touch screen.

When we receive pointer events from wlroots, we get coordinates in a normalized space between 0 and 1. To convert them to pixel coordinates, we just have to multiply these numbers by the screen resolution.

We also get a touch ID from wlroots that stays consistent from the moment a finger touches the screen until it's lifted up. Touch events coming from the same finger will have the same ID, starting from 0 for the first finger, 1 for the second, and so on. I could simply fill the last struct field `device` with this number, but I decided to pass the touch ID + 1 instead. Flutter expects every pointer to have a unique ID, whether it's a finger, mouse, or stylus and I wanted to reserve ID 0 for the mouse.

Once we call `FlutterEngineSendPointerEvent`, the engine delegates the event to the Flutter framework and widgets can react to it.

Only Flutter knows where the window or popup is positioned on the screen, so it makes sense to route touch events through Flutter, to have a widget on top of Wayland surfaces that capture these events, and then send some information to the embedder through a platform channel. Flutter receives global

touch coordinates, and the embedder gets back the Wayland surface the touch event was captured on, and the local coordinates of the touch event relative to the surface.

For this reason, I've created a widget called `ViewInputListener` whose only purpose is to capture touch and mouse events, and then send data back to the embedder. There are two widgets in Flutter that can capture touch events: `GestureDetector` and `Listener`. GestureDetector is not suitable for this use case because it captures high-level gestures like panning. `Listener` would be more appropriate because it captures raw events: down, motion, up, and cancel. However, `Listener` doesn't play in the gesture arena like `GestureDetector`, which made me reluctant to use it because I had planned to implement some system-wide gestures like swiping from the bottom to reveal the task switcher. Implementing these system-wide gestures would mean overlaying an invisible input detector at the bottom of the screen in case the user drags up. Overlaying two `Listeners` on top of each other wouldn't work because they would both capture the same events, and I wanted one to have priority over the other. I was looking for a widget like `Listener` that gives me low-level input events and fights in the gesture arena with other overlaying widgets.

I couldn't find anything like that in Flutter or on `pub.dev`, the package repository for Flutter, so I decided to implement it myself. In short, I managed to implement such a widget in around 200 lines of code by creating a custom `GestureRecognizer`. I've published the package called `arena_listener` on GitHub[26] and `pub.dev`[24].

Using the `ArenaListener` widget I managed to capture these low-level input events and send them to the embedder through the platform channel. The view ID of the surface and the local position of the event are being sent. Once the embedder gets this information, it simply forwards these coordinates to the Wayland client in focus. `src/flutter_engine/platform_api.cpp` contains 4 methods that are called from Dart through the platform channel: `touch_down`, `touch_motion`, `touch_up`, and `touch_cancel`.

Mouse input

Implementing mouse support is a little bit more involved than touch, mainly because the Flutter engine asks for more information from the embedder.

The files `src/input/mouse_input_device.hpp` and `src/input/mouse_input_device.cpp` contain the class `MouseInputDevice`.

If a mouse is present, we can register 4 callbacks for the following events:

1) Relative motion - the mouse moved by some amount of pixels
2) Absolute motion - the mouse jumped to a specific position
3) Button - a button press

4) Axis - scrolling

Again, all events contain a timestamp. Motion events contain normalized XY coordinates, button events contain the button ID being pressed or released, and the axis event contains the orientation of the scroll wheel and a delta for the scroll amount. In this case, Flutter puts more responsibility on the embedder because every time we send a pointer event, it needs a bitmap of all currently pressed buttons. As we capture mouse events in our callbacks, we can accumulate the state of pressed buttons in a set. We then construct a bitmap understandable by Flutter as per the enum `FlutterPointerMouseButtons`, using simple bitwise operations. This button state tracking is done by the class `MouseButtonTracker`.

As like for touch events, we can use the function `FlutterEngineSendPointerEvent` to notify Flutter about mouse events, and once again, mouse events are routed through Flutter and back to the embedder.

Keyboard input

When you press a key on a keyboard, it generates an ID called scancode which is the same for every keyboard regardless of the keyboard layout. In wlroots we can register a callback when the user connects a keyboard, and then connect to two signals: when the user presses a modifier (like CTRL or SHIFT), and when the user presses a key. These two callbacks are defined in `src/input/keyboard.cpp`.

Keyboard events are not routed the same way as touch and mouse events. The Flutter engine has no C API to send keyboard events. The way to do it is to send a JSON string through a platform channel predefined by Flutter called `flutter/keyevent`[35]. There is little documentation on how to format messages sent through this channel but looking at Flutter's deserialization code I managed to figure out what information it expects.

This JSON object must contain the following fields:

- `scanCode`: The ID of the key that was pressed independent of the keyboard layout.
- `keyCode`: The ID symbol of the key that was pressed, taking into account the keyboard layout.
- `modifiers`: A bitmap of all the modifiers that are being pressed.
- `type`: Can be "keydown" or "keyup".
- `keymap`: Hardcoded to `linux` because we're running on Linux.
- `toolkit`: What toolkit is being used, "gtk", or "glfw".

The `toolkit` field needs some explanation. As I currently understand it, different toolkits use different key mappings, especially for encoding modifiers. Because Flutter for desktop uses GTK for windowing API, it expects key mappings encoded in the same way GTK does. GLFW was the old windowing API used by Flutter and that's why the option is still available.

We don't use GTK or GLFW, but we need to choose an encoding. I've decided to go with GTK

key-mappings because it's the one Flutter for Desktop currently uses. This means that we have to translate the raw input events we get from wlroots into GTK key mappings for Flutter to be able to decode them.

There is a popular library on Linux installed on all distributions called `libxkbcommon` which lets you keep track of the keyboard state (keys, modifiers, and LEDs) and translate scancodes into keycodes given a keyboard layout, so that's what I'm going to use. Besides, wlroots also integrates nicely with this library.

Translating from a scan code to a key code consists of just calling the function `xkb_state_key_get_one_sym` with the key sym. Translating modifiers is trickier because there is not a single encoding. wlroots exposes its own enum for modifiers, but Flutter expects the GTK mapping. Looking at the deserialization code I found out that this mapping was almost 1 to 1, only the value of the Win key had to be changed.

This method worked pretty well, but it's not perfect. Some combinations of keys and modifiers are not correctly recognized by Flutter. It's clear that my method of translating wlroots key events into GTK ones is not complete and I'm missing some things, however, I decided not to spend more time on this because I have already spent enough time reading the GTK source code trying to find out how the mapping is done. It's complex with many edge cases, like everything related to languages.

At this point, Flutter widgets can capture keyboard events, but we need to forward these keyboard events to the Wayland client in focus. When sending a keyboard event through the `flutter/keyevent` platform channel, Flutter replies back with a boolean which tells the embedder if the event was handled or not. If there is a `Shortcuts` widget that captures a particular key combination, Flutter will reply with true. Otherwise, if there is no widget interested in that event, it will reply with false. This handled property is very useful for implementing system-wide keyboard shortcuts. Imagine if the user presses the Win key to open the app drawer, Flutter should consume this event, but no Wayland client should be able to receive it because it was handled entirely by the compositor. The embedder can decide based on this property whether it should inform the Wayland client about the keyboard event or not. In our case, if the event was not handled by Flutter, we just forwarded it to the Wayland client in focus.

Sending keyboard input events to Wayland clients is much easier than sending them to Flutter. When a new keyboard is connected, the embedder chooses the English US layout (hardcoded at the moment) and informs the client about this layout through the single `wl_keyboard` object that the Wayland protocol exposes. When the user presses a key, the scan code is sent to the client using the `wlr_seat_keyboard_notify_key` function and that's it. Clients will do themselves the work of interpreting the scan code correctly based on the given keyboard layout.

It's worth noting that, while Flutter is already capable of capturing keyboard events, when you focus on a `TextField` and start typing, it will not work. As I have planned to add an application drawer where you can search for applications by typing, I needed this functionality to work.

It turns out that Flutter expects the embedder to communicate with it through another platform channel called `flutter/textinput`[34]. Flutter will notify the embedder when new text input fields have been created, and the embedder has the responsibility of keeping track of the state of these text inputs (selection extents and the text). It can be seen as a protocol by itself because it's well-documented what fields the JSON messages sent on this channel should contain.

The Flutter engine repo already has some helper classes[1] that represent the state of a text input field, so I decided to copy them and use them in my project because there was no reason to implement them. The directory `src/third_party/text_input` contains the files I copied.

I've also written a class called `TextInputClient` (`src/flutter_engine_text_input_client.cpp`), that encapsulates the the input field model. It constructs the right JSON messages and sends them to Flutter through the `flutter/textinput` platform channel whenever I want to change the text of the text field.

Transforming a key code into a UTF-8 character is very easy because `libxkbcommon` provides a function called `xkb_keysym_to_utf8`, so when the user presses a key, we turn the key code into a character, we append the character to the text input, then we send a JSON message to Flutter with the new text input state.

Implementing a virtual keyboard

A virtual keyboard is necessary for touch devices, otherwise, it's impossible to introduce text. There is a Wayland protocol extension called "Text input"[39] that lets Wayland clients communicate when a text field has been enabled or disabled, and the compositor can send text to the application. This extension provides many more features around text input that make it possible to build a fully functional virtual keyboard with word suggestions and auto-completion, but I've limited myself to the basics of just knowing when a text field becomes focused.

wlroots already implements this protocol extension which makes things a little bit easier. The extension provides a global called `zwp_text_input_manager_v3` which clients can use to create `zwp_text_input_v3` instances. We can hook into the creation of these Wayland objects and register a callback on the `new_text_input` signal on the text input manager. Since every time a text input instance is created we're going to attach more callbacks to it to listen when the client wants to enable or disable a text input, I've wrapped the text input instance in a class called `ZenithTextInput`

which stores signal registrations. You can find this class at `src/input/text_input.cpp`. The virtual keyboard we're going to display will obviously be a Flutter widget, so we need to notify Flutter when a text input is enabled or disabled. As usual, we're going to send this information through the platform channel with the method name `send_text_input_event` and the payload will be the view ID of the surface that has the text input, and whether it's enabled or not.

The virtual keyboard widget was pretty straightforward to implement. All the related files are found at `lib/ui/mobile/virtual_keyboard/`. Using columns and rows I was able to place buttons in a grid and make them look like the Android keyboard. I've also added a shift key to change capitalization and the ability to insert numbers and symbols. Using an Overlay, I was able to stack the virtual keyboard on top of a surface when a text input becomes enabled and hide it when it becomes disabled.

When the user presses a button on the virtual keyboard, it sends the character over the platform channel back to C++ which will then emit a `commit_string` event to the Wayland client whose text input was active. The client would then simply update the text field and the user would see the new character that he typed.

Unfortunately, this Wayland extension is not implemented by all Wayland clients. Applications written in GTK have very good support because GTK has done all the hard work of implementing this protocol extension, and the virtual keyboard works as it should. Firefox also works flawlessly on touch devices. On the other hand, Chromium-based browsers and Qt applications don't implement this protocol extension, so I had to find a workaround to make the virtual keyboard work with them.

For unsupported applications, there is no way to determine when an application expects keyboard input, so I decided to let the user manually enable the virtual keyboard by clicking a button. The text input Wayland protocol was also capable of sending a string of characters to the client, but here, I had to rely on emulating a real physical keyboard. When the user presses a button on the virtual keyboard, the symbol would get translated to a key code based on the current keyboard layout (hardcoded to US English). This key code would then be sent to the Wayland client thinking it was sent by an actual keyboard, and then the client would translate the key code back into a character using the current keyboard layout. It's a hacky solution and it's not perfect, but it works well enough. I've seen this behavior on other mobile compositors like Phosh, so it inspired me to do the same because I couldn't come up with a better idea.

Figure 9 shows the virtual keyboard that I implemented:

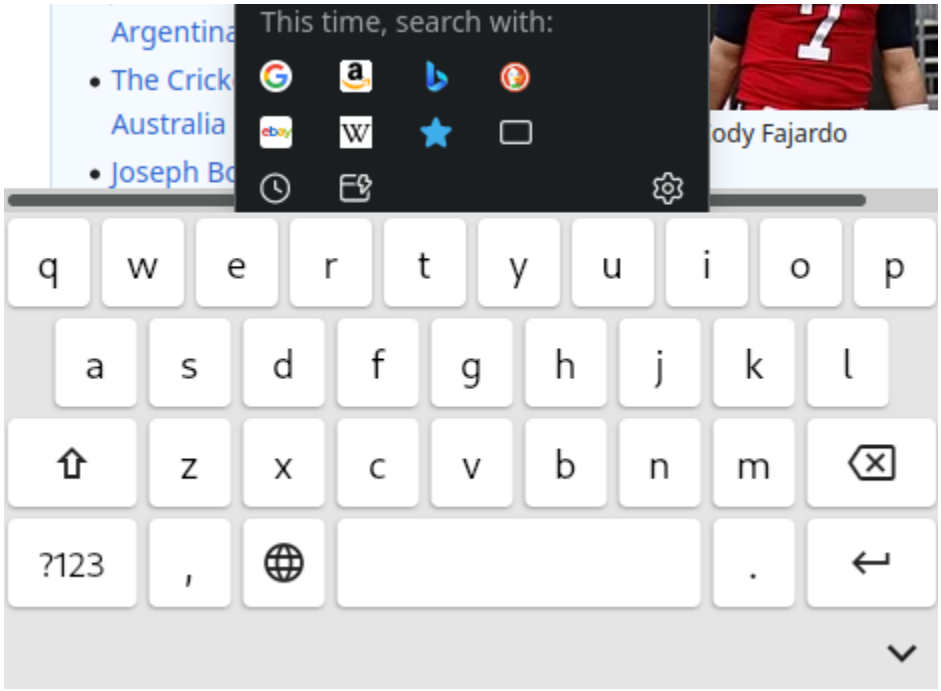

Figure 9: Virtual keyboard sliding up automatically when the user taps on a text field

Changing the screen brightness

The Linux kernel exposes backlight controls in sysfs under `/sys/class/backlight/`. On my phone and laptop, this path contains a directory named differently depending on the hardware, and this directory contains some files for controlling the brightness. There are 2 files we are interested in, which are `brightness` and `max_brightness`. `max_brightness` contains an integer number and it's the maximum value that we can write to the `brightness` file.

To be able to write to files in sysfs you need root permission, but compositors run under the user who's logged in. There are 2 ways I could get my compositor to be able to write to the brightness file. The first one is simply to change the permissions of the file, and the second one is to install a system service (a daemon) that starts at boot and runs as root. The compositor would then communicate with this daemon through DBus and send requests to change the brightness. All desktop environments that I know use the second approach because this daemon could evolve into managing other power-saving features and if more logic is needed, centralizing all functionality into a system service is a good idea. Besides, it's much easier to impose security on sandboxed applications and block them from changing power-saving settings if they have to go through the same DBus Unix domain socket.

In the long term, I plan on creating a daemon, but that's another project on its own and I wanted something quick and dirty to show that it's possible to change the brightness using a slider in Flutter. I decided to just write a script that changes the permissions of the directory in sysfs to the current user, that way I could write some Dart code to read and write the backlight files directly.

While playing around the slider, I noticed that a linear change in brightness written to the brightness file does not result in a linear change in perceived brightness to the eye. Apparently, brightness follows a logarithmic curve where `perceived light = sqrt(measured light)`. I implemented this formula into the code. The slider would show the perceived brightness, but the actual brightness written in the file would be the square of the perceived brightness. The way of calculating the brightness was much better and you could actually perceive the difference in brightness even at high values.

Figure 10 shows the slider in action:

Locking the session and authenticating the user

Like with any phone, I wanted to be able to lock the session when the power button was pressed, so I implemented a lock screen. I decided to go with pin authentication because at the time I started implementing this feature, the virtual keyboard wasn't working for text fields put on the screen by the compositor itself, but only for text fields from Wayland clients. Therefore, I just arranged some round buttons on the screen that would append some digits to a string in memory. Even though my phone

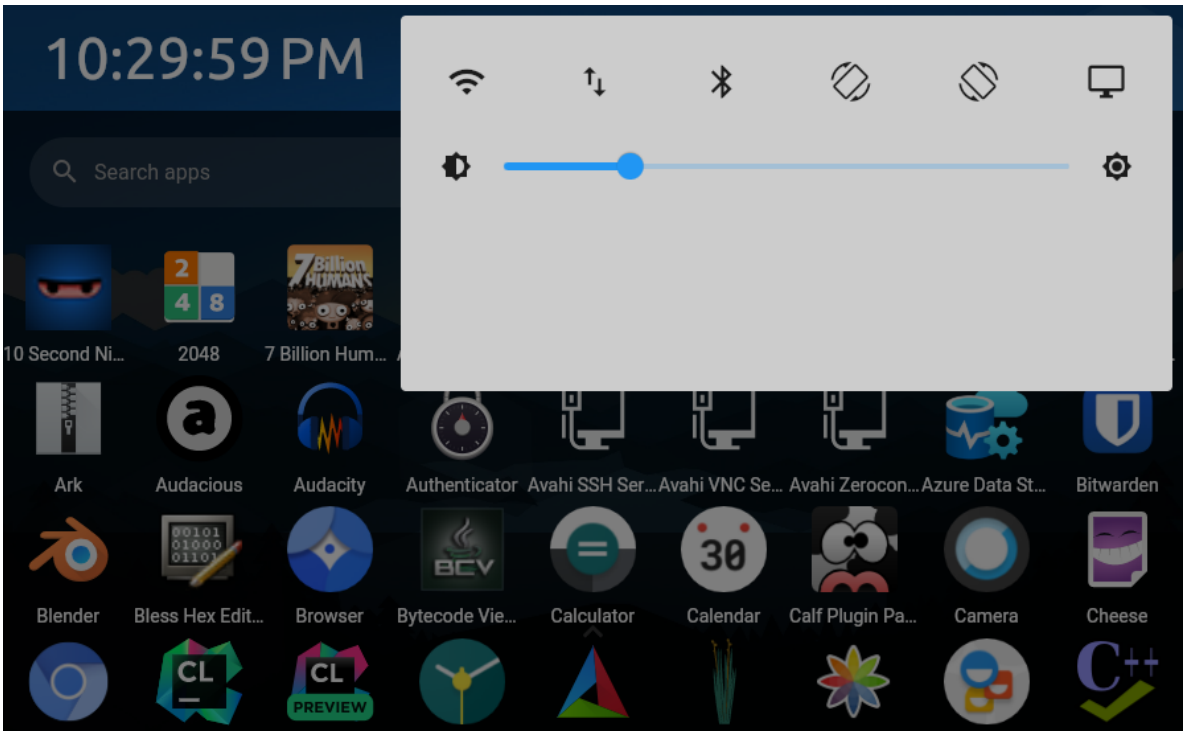

Figure 10: Brightness slider that changes the screen brightness

has a fingerprint sensor, all online resources told me that nobody got it working under a regular Linux distribution, so I didn't even bother to try to make it work.

The way to listen when a power button is pressed is the same as listening to any keyboard key. Flutter's key enumeration contains all the possible keyboard keys and also the more obscure ones like the power button. A regular Flutter application would normally never get notified when the power button is pressed because the compositor is stealing this input event for its own purposes to maybe display a power menu, but since we are the compositor, we receive all input events unfiltered. We can register a global key event callback in Flutter, and it would get called any time the power button is pressed. This callback will modify a provider which will put an overlay showing the lock screen.

When the lock screen is up, the user is presented with a PIN pad where he can enter his PIN. For this to work, the user account must have a PIN set up and not a password. I haven't implemented a way for the user to change his PIN/password using the graphical interface, so this must be done in the terminal prior to starting the compositor.

When the user presses the enter button, the pin stored as a string of characters is sent over to the embedder through a platform channel. Then, the embedder will rely on the PAM library (pluggable authentication module) to verify that the pin is correct. PAM will automatically open `/etc/shadow` for us and compare the hashes. I guess I could have done this work manually, but PAM is the standard way to authenticate users on Linux because it can be configured to use other more complex authentication methods like fingerprint or hardware security key. Once PAM returns the result of the authentication, the embedder will send the result back to Flutter again through the platform channel. The lock screen will then simply check the boolean value and if the authentication is successful, it will remove the overlay and the user will be able to use the phone again. Otherwise, an error message will be shown and the user will have to try again.

But before PAM can be used, a configuration file must be installed in `/etc/pam.d/` which describes what steps will have to be done to authenticate the user. Every application that uses PAM must install its own configuration file, so that's what I did. I created a file called `zenith` which contains these important lines among other less important ones:

```
-auth      [success=1 default=ignore]  pam_systemd_home.so
auth       required                    pam_unix.so          try_first_pass nullok
auth       optional                    pam_permit.so
auth       required                    pam_env.so
```

I've watched some YouTube videos, read some articles, and looked at other config files from other programs to understand the basics of how PAM works. In short, PAM goes through the list of modules

listed in this config file and executes them one by one. These modules can take the user's password and authenticate him in various ways depending on which module is chosen. `pam_unix.so` is the one to read `/etc/shadow` and validate the password. Other modules like `pam_systemd_home.so` may be necessary to ensure the user's directory is mounted before the authentication happens. `pam_permit.so` will permit access because if we get to that point, `pam_unix.so` must have already succeeded. `pam_env.so` will set some environment variables before finishing the authentication process. The `required` keyword means that if this module fails, the authentication process will stop and the user will not be authenticated. `optional` means that if this module fails, the authentication process will continue and the user might still be authenticated.

Before the actual authentication function call is performed using the PAM library, a PAM context has to be created by calling `pam_start` and referencing the name of the config file that we have created so that PAM knows what steps to take to authenticate the user. Then, we can call the `pam_authenticate` function which will return `PAM_SUCCESS` if the authentication was successful, or an error code otherwise. The `pam_end` function must be called to free the PAM context.

Figure 11 shows the lock screen with the pin pad:

Implementing an app drawer

It would be nice to have a way to launch applications using the user interface provided by the compositor, instead of having to spawn a terminal every time. freedesktop.org has created the desktop entry specification[31] which are configuration files describing how a particular program is to be launched, how it appears in menus, etc. These config files are text files with their own file format described by the specification. They are usually stored in `/usr/share/applications` for system-wide applications or `~/.local/share/applications` for user-specific applications, but the `XDG_DATA_DIRS` environment variable contains all the directories where desktop entries might reside. Each desktop entry file contains various key-value pairs that provide information about the application. Some of the essential keys include:

- Name: The human-readable name of the application.
- Exec: The command used to execute the application.
- Icon: The name of the icon file associated with the application.
- Type: The type of the entry (e.g., "Application" or "Link").
- Categories: Categories that describe the type or purpose of the application (e.g., "Office" or "Graphics").
- MimeType: The MIME types supported by the application.
- Comment: A brief description or comment about the application.

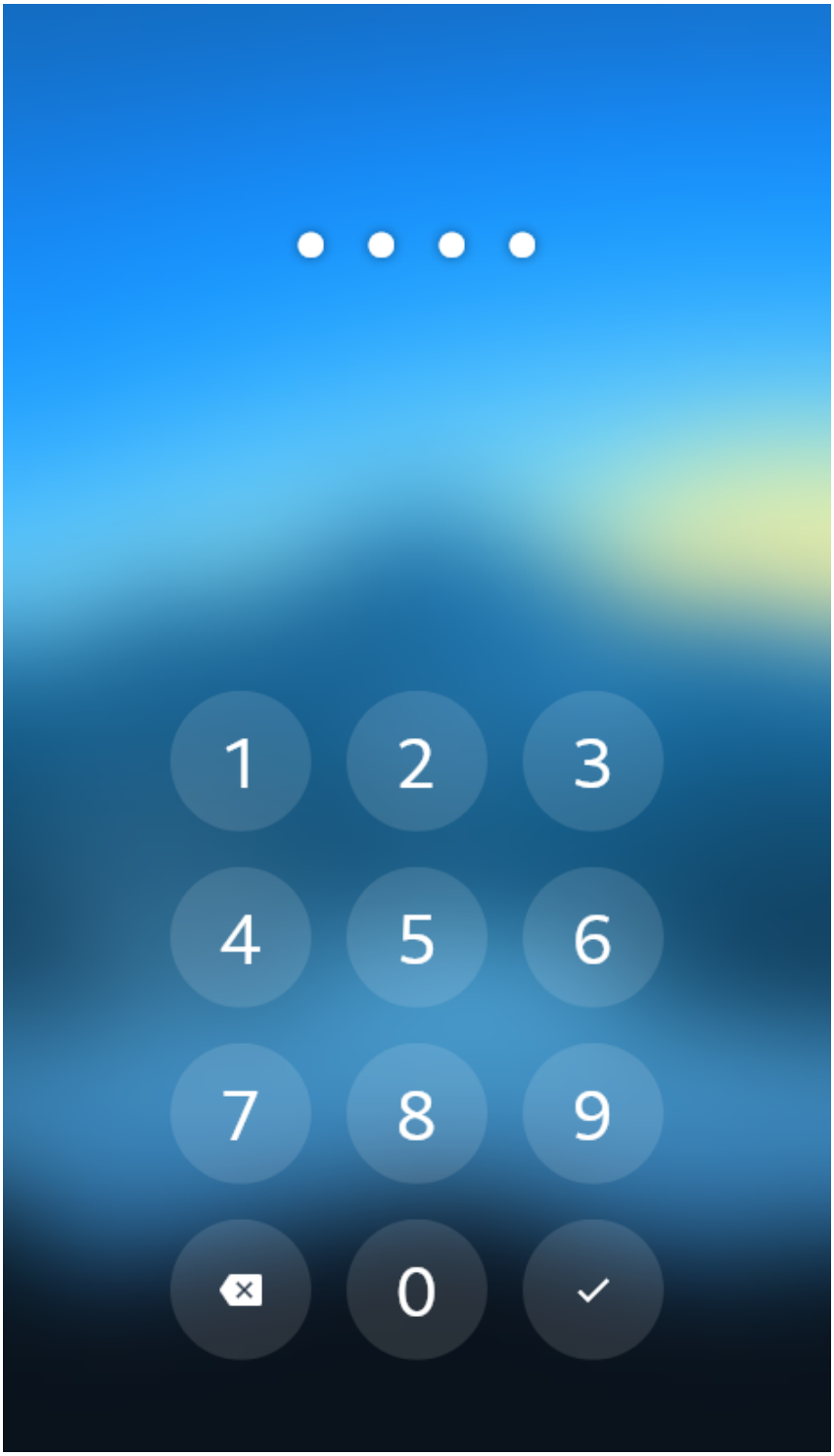

Figure 11: When the power button is pressed, the user is presented with a pin pad where he can enter his pin. If the authentication with PAM is successful, the lock screen overlay disappears.

- StartupNotify: Whether the desktop environment should show a startup notification for the application.
- Terminal: Specifies whether the application should run in a terminal window.

The compositor must be able to parse these files and display the applications in the app drawer in a grid. When the user clicks on an application, the compositor must spawn a new process with the command specified in the `Exec` key. The compositor must also be able to parse the `Icon` key and display the icon in the app drawer.

There are 2 ways I could get a working app drawer. The first one is to use a third-party library that parses desktop entries for me. The second one is to write my own implementation of the specification in Dart. GTK and Qt both have their own implementation of the desktop entry specification, but these libraries are huge because they are essentially entire UI toolkits that do a million other things. I decided not to use them because I would be a fool to depend on another UI toolkit when I'm already using Flutter. I probably wouldn't have chosen this approach if the specification was more complex, but it's simple enough to be implemented in a couple hundred lines of code. I also searched for Dart libraries on `pub.dev` before starting my own implementation, but unsurprisingly, I couldn't find any. The Dart package repository has rarely any packages specific to Linux. I took this as an opportunity to contribute to the Dart ecosystem and publish a useful, albeit niche, package[23].

The desktop entry file is organized in sections. There is a mandatory `[Desktop Entry]` section that must be always present and contains essential information such as name, icon, description, etc. After this section, there might be multiple actions defined, one action being one section. Actions are used to define multiple ways to launch an application. For example, the desktop entry file for Firefox contains 2 actions: `new-window` and `new-private-window`, all having the name translated into multiple languages and the command to launch the application. Usually, desktop environments show these actions when you right-click on the application icon in the app launcher or the dock. Each section has a list of key-value pairs, and each key might have multiple values if the writer of the file has translated the value in multiple languages.

The structure of the file naturally follows a tree because the file contains sections, one section contains multiple keys, and one key may contain multiple values. Because of this, I decided to store all this information inside nested hashmaps. The top-level hashmap maps from the section name to a lower-level hashmap mapping from the key name to another hashmap mapping from the locale to the value, so 3 levels of nesting.

The file format was pretty easy to parse. Using regular expression parsing provided by Dart, I was able to write expressions for sections and key-value pairs with locales and extract all this information inside the nested hashmap.

A user's computer is set to a country and language, and the app drawer usually only wants to show values localized for the user, ignoring all the other localized values. For convenience, I've decided to provide a method that would take this 3-level hashmap and filter out all values that don't match the provided locale. It returns a 2-level hashmap (section -> key -> one single localized value), which is easier to work with because you don't have to provide a locale every time you want to access a value. You localize the entire file once, and then you can access the localized values directly.

Desktop entries contain the icon name of the application, not the absolute path to the icon. Freedeskop has another specification called the Icon Theme Specification[32] which specifies where are icon themes installed, how are they organized in directories, and how to search for the absolute path of an icon from an icon name. This specification exists because the user can install different icon themes in order to change the look of their application icons. Similar to desktop entries, the spec defines the list of directories that can contain icon themes. Some of them are located in the home directory and have priority over themes from system directories, to let the user override icons from system themes.

Theme directories contain other subdirectories, usually called by icon size that they contain, but it's not a rule. For example, there might be a subdirectory called `96x96` containing icons of this exact size. In fact, theme directories must have a file called `index.theme` where every icon subdirectory must be listed with attributes like icon scale and minimum and maximum sizes.

`index.theme` also lets the theme creator specify which themes to inherit from. This can be useful when the creator makes a general theme full of icons, and a smaller, more specific dark mode theme that inherits from the base one and maybe overrides some icons from the base theme.

Again, I could have used a third-party library like GTK to index all the icons, but I didn't want to depend on another UI toolkit, so I decided to implement the specification in Dart myself. The format of `index.theme` is the same as for desktop entries, so I have been able to parse it using regular expressions. However, icon lookup is much more complex. We discussed how the same theme can have directories distributed across home and system directories, and on top of that, themes can inherit other themes. The specification provides some pseudocode that shows in what order to iterate these directories and what rules to follow to find an icon. While this pseudocode is very good at explaining how the icon lookup works, it is a very inefficient algorithm because in lots of places it conditionally checks for the existence of files. If I implement that, every icon lookup would result in tens of thousands of system calls checking if an icon exists at some path.

My first version was exactly the implementation of the pseudocode and, as expected, it was unacceptably slow. In order to reduce the number of system calls, I decided to recursively list all files in theme directories and store the trees as hashmaps in memory. Querying hashmaps was faster than executing system calls when checking for the existence of icons, but it was still pretty slow. When translating

icon paths to hashmaps, naturally, the first level was the theme directory, the second level was the icon size, and only after, the icon filename. Having the icon names at the leaves was a big disadvantage because I had to iterate the hashmaps in order to find the icon I was interested in. The third and final optimization I had to make was to index the paths differently, from the leaf to the root. When I want to find an icon, first, I would access the hashmaps by icon name, then by the size I want, and then take the icon from the first theme directory ordered by priority. The initial indexing takes a few hundred milliseconds for a few thousand icons, but look-up is very fast.

Figure 12 is a screenshot of the app drawer with names and icons parsed from desktop entries and icon themes:

Implementing a task switcher

A task switcher is the part of the compositor that lets the user switch between running applications. I got my inspiration from Android's task switcher where you can swipe from the bottom of the screen to reveal the list of applications, scroll horizontally through them, and tap on one to switch to it. I didn't have a reason to design my own UX and UI because I think Android's task switcher is already very good, so I decided to copy how it works in Flutter and keep the Android look and feel.

Since every application is basically just a Flutter widget at its core, implementing such a task switcher was just a matter of creating a horizontally scrollable widget and putting the window widgets in a list. I didn't go for the usual `ListView` widget because it was too limiting for my use case. When you swipe up from the bottom of the screen, all applications must zoom out so you are able to see and scroll through them. Also, when you swipe up on an application, it should start a fancy dismiss animation when the application quits. These were all things that I couldn't do with the regular `ListView` widget, so I decided to implement my own using a `Stack`. I ended up storing up the 2D positions of all applications in the stack, and then simply lay them out horizontally. This gave me a lot more freedom because I could position the applications anywhere in 2D space, and would let me implement all the fancy animations that I wanted. Using a horizontal gesture recognizer I could detect scrolling gestures which modify the scrolling position of the list.

Due to how Flutter's hit-testing works, I encountered this strange behavior where if you offset the entire `Stack` widget by the scroll position, some applications would be untappable. In Flutter, the bounding box of the child widget is always smaller or equal to the bounding box of the parent. Even though a child widget is allowed to visually overflow the parent, the child becomes untappable outside the parent widget. Flutter has taken this strategy because it makes hit testing more efficient, the complexity is logarithmic (the depth of a branch in the widget tree) instead of linear (every visible widget on the screen). In most UI patterns this behavior is not problematic, but it was for me. The

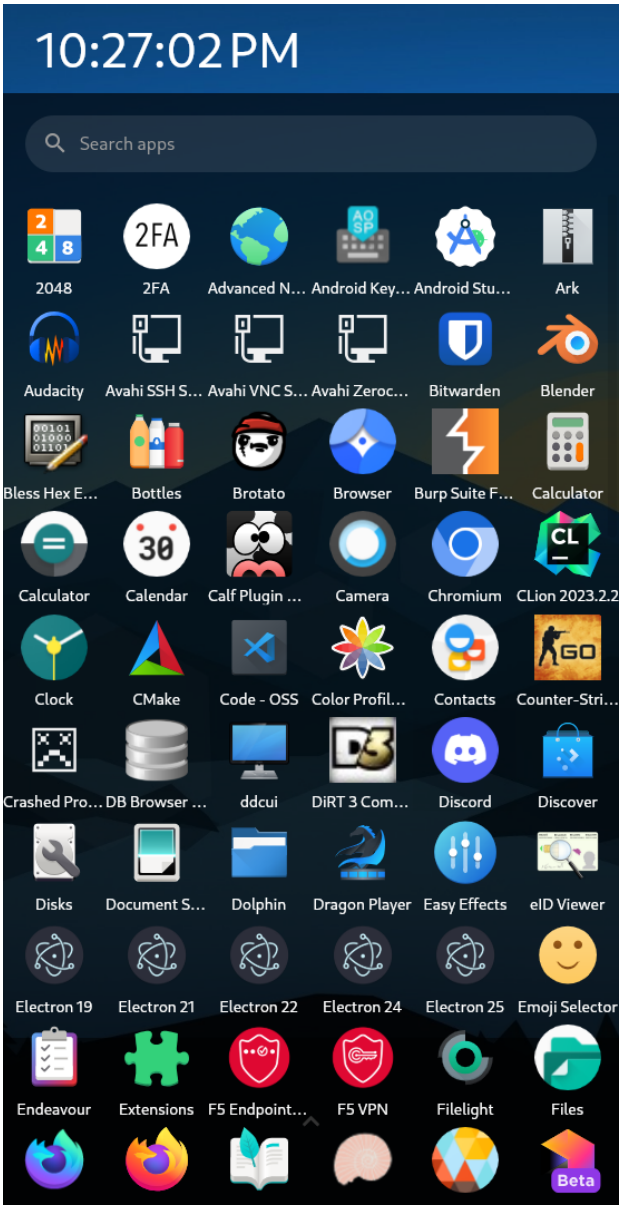

Figure 12: The app drawer of the compositor allows the user to launch applications.

`Stack` was smaller than the widgets that it contained, so most widgets would become untappable even though they were visible. To fix this issue, I decided to apply the scrolling position offset to every child of the `Stack` instead of applying it to the `Stack` itself. Even though this gives exactly the same visual result, the `Stack` always spans the entire screen and every widget inside it would remain tappable.

Figure 13 shows the task switcher in action:

Starting the compositor at boot

When the computer starts up, the kernel is loaded and spawns one single process, typically the init system. The init system, usually systemd on modern Linux distributions, has the job of starting all services and processes that are required for regular usage. These are not started randomly. Every service on the system that is started at boot has a configuration file that specifies what program to run, what other services it depends on, what user to run the process under, etc. The init system reads all these configuration files and creates a dependency graph. It then starts the services in the order specified by the graph, making sure that all dependencies are started before the service that depends on them. Not only services can have configuration files, but even short-lived programs. For this reason, systemd calls these configuration files unit files.

If we want to start our compositor at boot, we need to create a unit file for it and put it somewhere on the disk where the init system will read it. The appropriate place would be `/usr/lib/systemd/system`, where Debian packages usually install their unit files.

The unit file is a text file with a specific format. It contains a `[Unit]` section describing dependencies on other unit files, a `[Service]` section with environment variables and the command to run, and an `[Install]` section where you can specify an aliasing unit file. It's a convention in systemd to have a unit file called `display-manager.service` which is responsible for managing the display and the user sessions. If other unit files want to have a dependency on the display manager, they can refer to it by this generic name instead of using the name of a particular one. By specifying an alias to `display-manager.service` in the unit file, systemd will automatically symlink `display-manager.service` to our own unit file when it's installed in the system.

Since the compositor is using DBus for sending the power-off command, we have to write the following two lines:

```
Wants=dbus.socket
After=dbus.socket
```

`Wants` tells systemd that we have a dependency on DBus. `After` is to make the compositor start only after DBus has started. If `After` is not specified, systemd might start the compositor and DBus at the

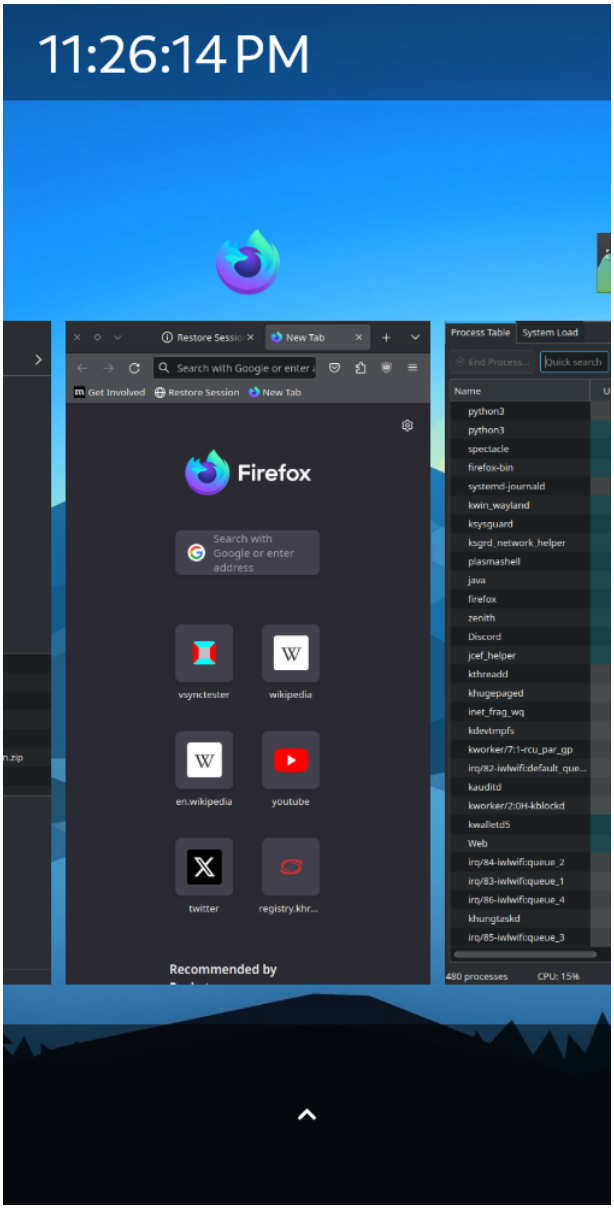

Figure 13: The task switcher of the compositor lets the user switch between running applications.

same time, which could create some race conditions.

Another interesting line is `TTYPath=/dev/tty3` which tells systemd to start the compositor on the third TTY.

The unit file is found in `dpkg/usr/lib/systemd/system/zenith.service` in the repo. I didn't write it from scratch. I've inspired myself from another compositor called phosh[33] and modified the file to better suit mine.

Powering down the device

The init system, systemd, registers a manager object on DBus and exposes the interface described here at this URL[8]. This interface has a method called `PowerOff`. Calling this method through DBus will power off the computer, so I just created a button in Flutter that powers off the computer when pressed.

Another solution would have been to execute the shell command `systemctl poweroff` which does the same thing, but it's the `systemctl` command that does the DBus call. I decided to cut unnecessary steps and call the method directly.

Packaging

I wanted my compositor to be installable like any other package. Since I'm targeting Debian Bookworm as the Linux distribution, the only sensible choice I had was to package it as a deb file. The libraries I'm depending on are the same on Debian Bookworm and on Ubuntu 22.04, so the package could be installed on both distributions.

I've glanced over other distribution methods. Flatpak and Snap are the most popular ones and allow packaging applications for all Linux distributions. Flatpak would probably not be the best choice because it sandboxes applications and is more appropriate for regular GUI apps. I read that for system programs, Snap, developed by Canonical, is better suited. I decided, however, to go with the Debian package because Snap is not preinstalled on any Linux distribution other than Ubuntu, and I wanted something that could easily be installed without additional steps.

Deb files are basically tarballs containing a few metadata files alongside the files to be installed on the system. Among metadata files, inside the archive, there must be a file called `control` located in a directory called `DEBIAN`. This is a text file containing information like the package name, architecture, version, other dependent packages, the maintainer of the package, a description, and others. A few of them are mandatory, but most of them are optional. The full list of fields can be found in the Debian Policy Manual[9].

At the root of the archive, all directories excluding `DEBIAN` will end up copied to the system by the package manager. For example, if the archive contains the directory structure `usr/bin/program`, the file `program` will be copied to `/usr/bin/` on the system. Usually, executable files are placed in `/usr/bin`, shared libraries in `/usr/lib`, and other additional files in `/usr/share`. I, however, am not quite fond of the idea of scattering my compositor's files over multiple directories in the system. I think this approach works great for fundamental programs like `ls` because it's easier to search for executables and libraries if there is a standardized directory for them. The same goes for common libraries used by other applications. All files of the compositor, including the Flutter engine shared library, are not meant to be shared with other programs on the computer. I very much prefer installing everything in one single directory. There is a directory called `/opt` where software can be installed in its own subdirectory, so I decided to install my compositor under `/opt/zenith/`. Usually, proprietary software gets installed in `/opt`, but some open-source software can be also found here.

I created a directory `opt/zenith/` inside the archive and copied all the files there. There is also the unit file that has to be installed on the system, and this one must be installed in `/usr/lib/systemd/system`, so I created the appropriate directory structure inside the archive and placed the file there.

The PAM configuration file must be placed inside `/etc/pam.d/`, so I created the directory structure inside the archive and placed the file there.

I explained the process like I would have done it manually, but I wanted to automate the process. Instead of copying these files manually, I defined a Make target that compiles the compositor, creates a temporary directory meant to store all the files that will end up inside the archive, copies the files inside the directory, then runs `dpgk-deb`, a tool provided by Debian that creates a deb package.

Automating releases

On my repo, every time I commit with a new tag, GitHub Actions will automatically compile the compositor for x86_64 and ARM, create the Debian packages for both architectures, and then upload them as artifacts on the release page.

The GitHub action is defined in `.github/workflows/main.yml`. The process of creating an artifact for x86_64 was quite straightforward. I chose Ubuntu for the container I wanted to use, I installed all the required dependencies with `apt`, cloned the repo, compiled the compositor, created a deb archive, and then published the archive as a new release.

Compiling for arm64 was a little bit more complicated, simply because surprisingly enough, GitHub doesn't have runners for arm64. After searching for solutions, I found a GitHub action called `run-on-arch-action`[18] which allows running a GitHub action on a different architecture than

the one of the runner. This action uses QEMU to emulate the target architecture. The steps are the same as for x86_64, just run in an emulated arm64 environment. `run-on-arch-action` did not provide Debian Bookworm as a distribution for arm64, so I ended up forking the repository[25] and adding it myself by modifying a few lines of configuration files. I opened a pull request[27] to the original repository but unfortunately, the maintainer did not seem to be active, so the pull request remains open to this day. For this reason, I continue using my own fork.

Testing and benchmarking

I've tested my compositor on 3 devices: a laptop with Intel integrated graphics with a touchscreen, a desktop PC with AMD integrated graphics, and my phone, a OnePlus 6. It's important to test on multiple devices with different GPUs because sometimes GPU drivers can behave differently on undefined behavior, especially for low-level operations like texture synchronization. It happened to me that the compositor would work well on my laptop, but I would have visual artifacts on my phone because I forgot to wait for the GPU to finish rendering before showing the frame on the screen.

Regarding the Wayland integration, I've tested multiple applications. This is a non-exhaustive list of applications that I regularly tested during development:

- Firefox
- Google Chrome
- VS Code
- Other Chromium-based applications
- Qt applications
- GTK applications
- Flutter applications

These applications all worked without problems. I'm aware that there are also Wayland clients written specifically to test the completeness of a Wayland implementation including edge cases, but I didn't test those because my goal was to make a prototype and show that you can use real-world applications. If I made sure I implemented 100% of the Wayland protocol correctly, I wouldn't have had the time to implement anything else because some parts of the protocol are complex. There are parts I haven't implemented yet because they were not really necessary to have applications running. They exist to make the experience more pleasant for the user, for example, by letting an application put another window in focus.

Testing touch input was pretty easy because I fortunately have a laptop with a touch screen so I didn't have to exclusively rely on my phone to guess why something was not working. Developing and running the compositor on the same machine was very convenient.

Regarding benchmarking the compositor, I didn't perform any precise benchmarks because I didn't think the numbers would be useful. Any latency that the compositor might have introduced is negligible compared to the time it takes for the monitor to show a new frame. On a 60 Hz screen, the user sees a new frame every 16 ms. Even if my compositor introduced a few milliseconds of latency somewhere between actuating a mouse button and seeing the feedback on the screen, in most cases it would still not matter because the refresh rate is the biggest limiting factor.

Latency is important for video games, especially esports, where minimizing the time between the user's input and the feedback on the screen is crucial because it gives you a real competitive advantage, but for user interfaces, it is a non-issue. Frame pacing and smoothness are much more important than frame latency. I didn't notice any additional latency (input or output) over my day-to-day desktop environment that I use (KDE Plasma) which is good.

I also cannot do direct comparisons in resource utilization with other compositors because I'm not even close to having the number of features that they have. It's not surprising that RAM usage was lower than KDE Plasma. CPU utilization was very similar to KDE Plasma. Compositors are event-driven programs that will idle most of the time, so CPU usage was 0% when idle, and 1-2% when it had to generate frames continuously. CPU usage was a little bit higher on my 5-year-old laptop compared to my desktop PC because the processor is not as powerful, but not by much.

I'm not saying that my compositor is as efficient as it can be, there are lots of improvements that can be made, but it's definitely not the worst.

Implementing a system settings application

As every desktop environment has its own settings application, I wanted to have one too. The reason every desktop environment has its own, it's that some settings are dependent on the compositor, and there there no standard protocol that specifies how a settings application should communicate with the compositor. Changing resolutions is, for example, a compositor-specific that every desktop environment implements differently.

However, there are settings that are not related to the compositor at all. For example, networking is handled entirely by a system service called `NetworkManager` on all modern Linux distributions, and it's very well-documented how to communicate with it through DBus.

I decided to implement a settings application written in Flutter, a normal Flutter application for Linux separate from the compositor. I only had time to implement some basic networking functionality like turning on and off Wi-Fi and connecting to WPA-protected Wi-Fi networks, which is enough for home usage. There are other authentication methods for enterprise networks that I didn't integrate because

it would take me too much time and I had other priorities.

There is a D-Bus library for Dart called `dbus` written by Canonical which I could have used to communicate with `NetworkManager`, but if I want to call some method, I have to look at its documentation for the method signature that I have to manually encode as strings. This library is great for writing wrappers around D-Bus interfaces, but it's not at all ergonomic to use it directly. Luckily for me, Canonical has written another library called `nm`, short for NetworkManager that basically wraps the `dbus` library and provides a much nicer API by exposing D-Bus objects and methods as Dart classes and methods. And since Dart is a statically typed language, you can't pass in the wrong argument types which is great.

The repo is on GitHub at [36]. Figure 14 the list of networks after the scan is complete, and figure 15 shows buttons for forgetting and disconnecting from a network. I basically copied the design of the settings app on Android because I don't think it would have been a good idea to come up with my own design.

Future improvements and ideas

Improving the reliability

I've written the Flutter embedder in C++ because I was embarking on too many new things at once: learning the Wayland protocol, learning wlroots, learning how to write a Flutter embedder, and using the Flutter engine as a library. wlroots is considered the most mature batteries-included Wayland implementation library written in C. Besides, I had to copy some C++ code from the GitHub repository of the Flutter Engine in order to encode and decode platform channel messages. All these reasons made me choose C++ as the language for the embedder because it makes it very easy to integrate with existing C and C++ code and libraries. Now that I have a much better understanding of how everything works, I'm starting to believe that C++ was not the best choice.

Compositors have to be rock solid because if the compositor crashes, it can bring down all the applications with it. Xorg is the most popular implementation of the X11 protocol, and the compositor is another process that sends messages to the Xorg instance. The compositor can crash and restart without bringing all the apps down with it, as long as Xorg, the display server, is still running. Wayland, however, doesn't have a standardized server implementation. Compositors are expected to provide a display server using libraries like wlroots or build on top of the official Wayland C libraries that provide an implementation of the core Wayland protocol, without any popular protocol extensions that every application is using. This has led to a trend of creating a single process for the display server and the compositor. The upside to this approach is that communication between the display server

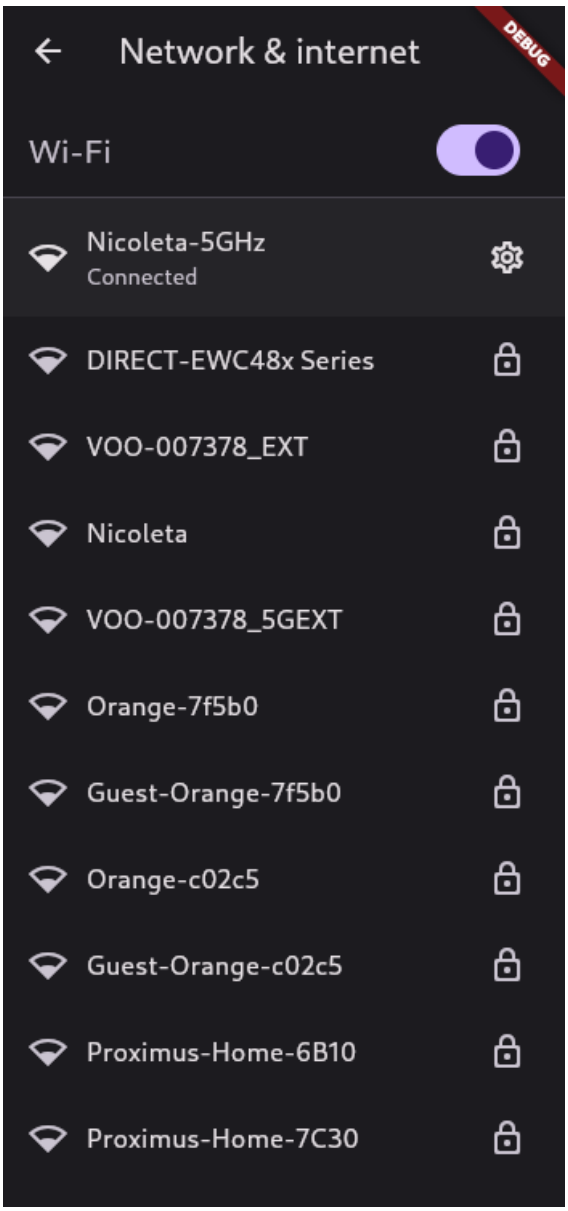

Figure 14: The list of networks after the scan is complete.

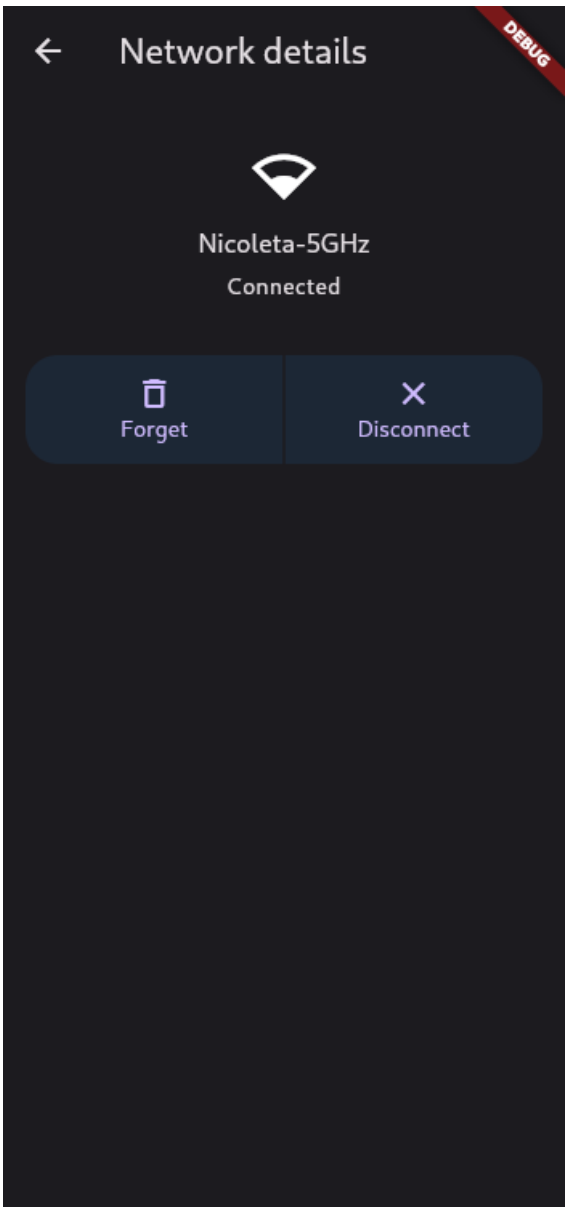

Figure 15: Buttons for forgetting and disconnecting from a network.

and the compositor is very fast because it doesn't involve the kernel, but if the compositor goes down, the display server will obviously go down too, losing all the open applications. I too, have chosen this approach because I didn't want to add even more things to my "to learn" list, and merging both in one process was the simplest way. Splitting the display server and the Flutter embedder into 2 separate processes would be a welcome improvement to improve its reliability.

Another welcome improvement I could make would be to rewrite the Flutter embedder in a memory-safe language. I consider myself quite proficient in C and C++, however, I still make subtle mistakes sometimes that lead to crashes and are hard to debug. I would like a programming language with more compile-time checks without sacrificing performance. Rust would be the best contender because it's a systems programming language that compiles to native code, and it's memory safe, meaning that it should be much more difficult to cause segfault. Of course, it doesn't mean my compositor will not be bug-free if I rewrite it in Rust, but at least it will eliminate whole classes of bugs that most of my crashes come from.

As a matter of fact, I've already started rewriting the embedder in Rust, a work that does not belong to this thesis but I would still like to mention it. I think it's a good idea to invest in Rust now and benefit from memory safety as soon as I can before I invest more time in C++ by growing the compositor with more features. I've spoken with open-source communities that would like to fork my work and use it as a base for their own compositors, and contributors would be much more willing to contribute to a Rust project than a C++ one. For the future of my project and others, I think it's the best move.

Improving the efficiency

Maybe half a year ago, Flutter added the ability to be set up to use the Vulkan graphics API instead of OpenGL ES. Vulkan is a much more modern API that allows more control over the GPU and is more efficient. Using Vulkan wouldn't necessarily make the compositor faster because it produces frames at the refresh rate of the monitor and most of the time it does nothing, but it would make it more power efficient, which is always something to strive for on mobile devices.

The compositor should make use of hardware CRTC planes when possible. A CRTC plane is a hardware component of the monitor that can display a frame buffer without any intervention from the GPU. The compositor could use these planes to display the background image, the cursor, and other UI elements without using the GPU to compose them together in one single frame buffer. Android already uses this technique to draw the application texture on one plane, and the system UI on another plane. CRTC planes are pretty basic in the sense that they can't apply complex transformations on their frame buffers, but the compositor should be able to know when to use them to save on power, and when to use the GPU if the rendering requires more complex operations like scaling, rotation, or blending.

CRTC planes can have random limitations imposed by each manufacturer, but the compositor should still be able to use them when possible.

When running applications in full screen like YouTube videos, the compositor should disable composition using the Flutter renderer and just display the application texture directly on the screen (a technique called direct scan out). This would reduce input and output latency and make the experience more pleasant.

A few additional features and fixes among many others

- Screenshotting and support for screen sharing would be nice to have.
- Notifications are an essential part of the user experience and should be implemented. The `mpris` DBus interface is a standard for controlling media players. The compositor should be aware of it so that it can show what song a media player is playing.
- The compositor should implement more Wayland protocols that applications could use. For example, applications can request the compositor to focus on another window of the same application.
- Authenticating the user with PAM blocks the compositor's main thread for a short time, which causes frame drops. The authentication should be done on a separate thread, and the communication done asynchronously through channels.
- The compositor should be probably packaged as a Snap package in order to make it available on all Linux distributions.
- Multi-monitor support is a big one. Currently, the Flutter engine does not support outputting multiple frame buffers, but Google is slowly working on it because this feature is desired for foldable phones with multiple screens, and multiple windows on the desktop platform. I could probably hack together something that works by compositing the contents of multiple screens on a single frame buffer and then cropping that frame buffer for each screen. However, this would probably present some issues regarding frame pacing if the refresh rates of the monitors are different.

Running Linux apps on Android

Prof. Laurent Mathy proposed the idea that I could run my compositor on Android because mobile hardware is much better supported by Android than by regular Linux distributions. It would be very cool to be able to run any Linux app on my phone and have it behave like a regular Android app. There are applications on the Play Store that let you run a Linux distribution in a container, but none integrate seamlessly with the system because they emulate a virtual screen and use VNC to display it

on the phone. Similar to how WSL works on Windows where you can even run Linux GUI applications, it would be entirely possible to create a proxy layer that translates the Wayland protocol to Android's windowing system.

Social impact

The first mail I received related to my work was from the creator of DahliaOS[2], a Linux distribution that uses Flutter for its user interface. While functional, this user interface provides a launcher where you can launch "applications". I've put the word applications in quotes because every application is actually a Flutter widget part of the user interface, and not a separate process running on the system. No third-party GUI applications can be run on this user interface because it does not host an X11 or Wayland server. The creator of this project was interested in my work because it was the first time he had seen a Wayland compositor that uses Flutter to render the UI and the windows of third-party applications. He told me other people have tried to achieve to same thing but failed, and my project contains the piece of the puzzle that was missing. Today, there is a private branch of DalhiaOS on GitHub (see figure 16) based on my compositor, and the user interface is modified to their own design language.

A second mail I received was from the creator of Material Shell[22], Adrien Peslerbe. Material Shell is an open-source Gnome Shell extension that heavily modifies its user interface into a tiling window manager with some unique ideas that optimize navigability by leveraging spatial awareness.

He wanted to transition the project from a Gnome Shell extension to a standalone Wayland compositor because the Gnome Shell API was too limited, was sometimes buggy, and the API had very poor backward compatibility, forcing him to make significant changes every time a new version of Gnome Shell was released. He was interested in my work because I was the first who had a decently working Wayland compositor using Flutter. He already had the idea of a Flutter compositor a few years ago, but no open-source prototype existed. After discussing with him for a while, he was convinced that my project was a good base for his standalone compositor. I'm excited to collaborate with him because Material Shell already has a community of users around it, and I want to see my work in production used by real people.

On November 4th, Adrien Peslerbe was invited by Canonical to talk about Material Shell at Ubuntu Summit 2023. He talked about Material Shell, and in the second part of the presentation he announced that he wanted to transition the project to a standalone Wayland compositor named Veshell using Rust and Flutter for the reasons I mentioned before. Veshell is under the Free Explorers collective[17]. I also got mentioned in the presentation for my prototype compositor that Veshell is based on.

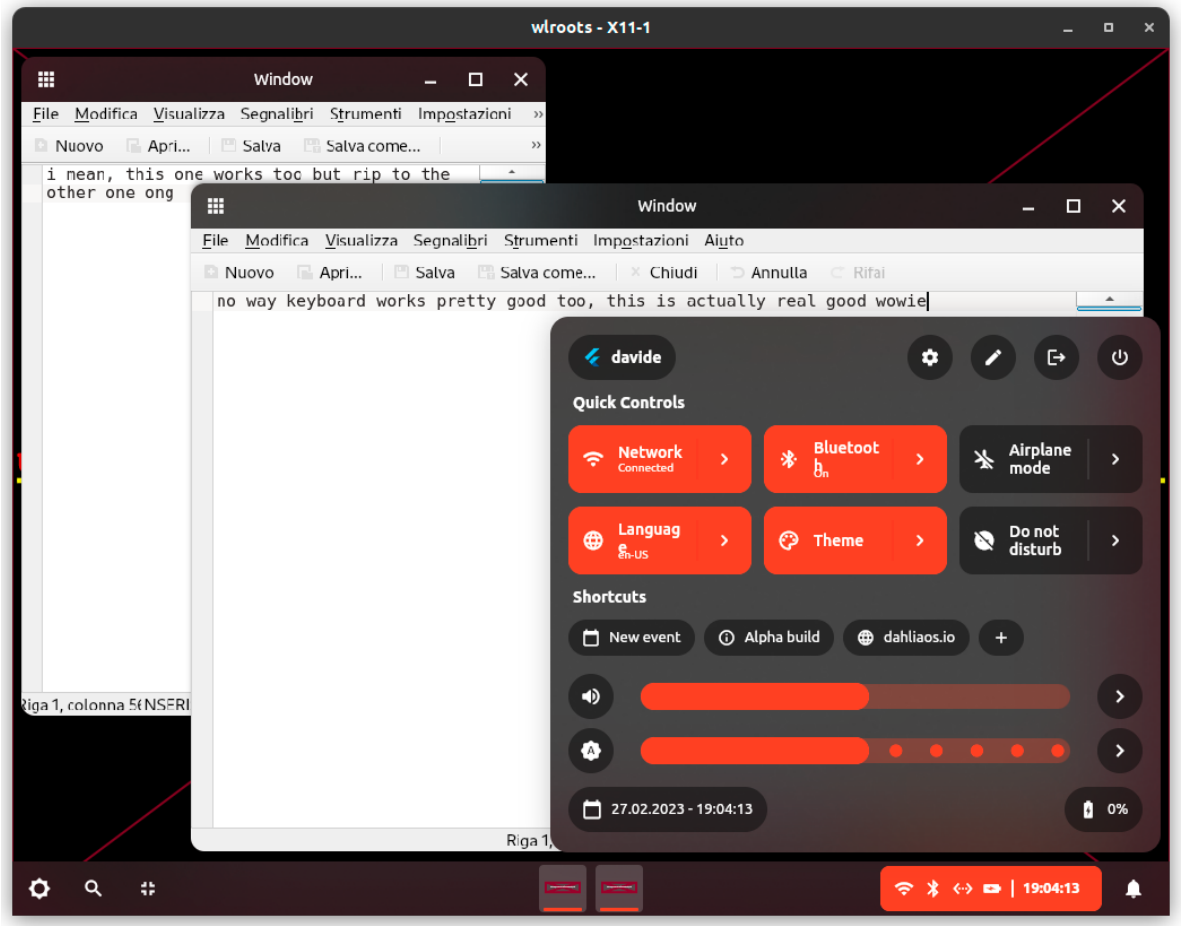

Figure 16: A screenshot of the DahliaOS private branch made by another user, which uses my compositor as a base. .

Conclusion

I'm very happy to see that people are interested in my project, and I'm glad I can contribute to open source by bringing something new to the table, however, my work is far from done. My compositor was all this time a prototype to prove to myself that it is possible to mix these technologies, and I never had a final product in mind. My new goal is to evolve the Flutter embedder further by collaborating with other open-source projects that want to incorporate my work, and once it's ready to be daily-driven on the desktop platform, I will continue with my quest of making a great mobile compositor. I can reuse the Flutter embedder, and just use my user interface written in Flutter adapted to mobile devices.

Aside from this project, I hope that more people will start experimenting with this technology stack and we'll see more compositors based on Flutter in the following years.

References

[1] *A C++ class that models the state of a text field in Flutter.* URL: https://github.com/flutter/engine/blob/fb80aafd259bacd20e9650522dd1cbeff11c8681/shell/platform/common/text_input_model.h.

[2] *An operating system that uses Flutter for its user interface.* URL: https://dahliaos.io/.

[3] *Basic integration of Flutter engine with GLFW.* URL: https://github.com/flutter/engine/blob/476985ae301414f0bf3dbf710f06eef9c26779f1/examples/glfw/FlutterEmbedderGLFW.cc#L85.

[4] *Code generator from Wayland XML protocol descriptions.* URL: https://gitlab.freedesktop.org/wayland/wayland/-/blob/1.16/src/scanner.c.

[5] *Creating a buffer object from a DMA-BUF.* URL: https://wayland.app/protocols/linux-dmabuf-unstable-v1#zwp_linux_buffer_params_v1:request:create.

[6] *Creating a buffer object from shared memory.* URL: https://wayland.app/protocols/wayland#wl_shm_pool:request:create_buffer.

[7] *Custom Flutter Engine Embedders.* URL: https://github.com/flutter/flutter/wiki/Custom-Flutter-Engine-Embedders.

[8] *D-Bus interface for querying and manipulating user sessions.* URL: https://www.freedesktop.org/software/systemd/man/org.freedesktop.login1.html.

[9] *Debian control file format.* URL: https://www.debian.org/doc/debian-policy/ch-controlfields.html.

[10] *EGL extension for exporting sync objects as file descriptors.* URL: https://registry.khronos.org/EGL/extensions/ANDROID/EGL_ANDROID_native_fence_sync.txt.

[11] *Explicit synchronization on Linux, article by Collabora.* URL: https://www.collabora.com/news-and-blog/blog/2022/06/09/bridging-the-synchronization-gap-on-linux/.

[12] *Exporting a sync object from a dma-buf.* URL: https://gitlab.freedesktop.org/mesa/mesa/-/blob/2e9ce1152e7205116050c5d7da58a2a66d0ed909/src/vulkan/wsi/wsi_common_drm.c#L43.

[13] *Exporting a texture as DMA-BUF for other processes.* URL: https://registry.khronos.org/EGL/extensions/MESA/EGL_MESA_image_dma_buf_export.txt.

[14] *Flutter Embedder Linux Artifacts.* URL: https://github.com/sony/flutter-embedded-linux/releases/.

[15] *Flutter predefined system channels for interacting with the embedder.* URL: https://api.flutter.dev/flutter/services/SystemChannels-class.html.

[16] *Flutter widget for displaying an external texture registered by the embedder.* URL: https://api.flutter.dev/flutter/widgets/Texture-class.html.

[17] *Free Explorers, a GitHub organization hosting the veshell compositor.* URL: https://github.com/free-explorers/.

[18] *GitHub Action for emulating jobs on other architectures.* URL: https://github.com/uraimo/run-on-arch-action.

[19] *Importing a DMA-BUF as a texture.* URL: https://registry.khronos.org/EGL/extensions/EXT/EGL_EXT_image_dma_buf_import.txt.

[20] *Input device library for Wayland compositors.* URL: https://wayland.freedesktop.org/libinput/doc/latest/.

[21] *International Components for Unicode (ICU) library.* URL: https://unicode-org.github.io/icu/userguide/icu_data/#overview.

[22] *Material Shell, a GNOME extension.*

[23] *My Dart library for parsing desktop entry files.* URL: https://pub.dev/packages/freedesktop_desktop_entry.

[24] *My Dart library that implements a listener fighting in gesture arenas.* URL: https://pub.dev/packages/arena_listener.

[25] *My fork of run-on-arch-action that adds support for Debian Bookworm ARM64.* URL: https://github.com/roscale/run-on-arch-action.

[26] *My GitHub repository for arena_listener.* URL: https://github.com/roscale/arena_listener.

[27] *My pull request to add Debian Bookworm ARM64 to run-on-arch-action.* URL: https://github.com/uraimo/run-on-arch-action/pull/82.

[28] *OpenGL sync object for synchronizing activity between the GPU and the application.* URL: https://www.khronos.org/opengl/wiki/Sync_Object.

[29] *Plumbing explicit synchronization through the Linux ecosystem, kernel mailing list discussion.* URL: https://lore.kernel.org/all/949b8373908a9895e97981e872d6e35dcaaba632.camel@lynxeye.de/t/.

[30] *Sharing memory with the compositor on Wayland.* URL: https://wayland.app/protocols/wayland#wl_shm.

[31] *Specification of desktop entry files, used to describe application metadata.* URL: https://specifications.freedesktop.org/desktop-entry-spec/desktop-entry-spec-latest.html.

[32] *Specification of icon themes.* URL: https://specifications.freedesktop.org/icon-theme-spec/icon-theme-spec-latest.html.

[33] *systemd unit file for booting the Phosh compositor.* URL: https://github.com/agx/phosh/blob/2c2e6fef64f30076c2da833cff91fad386961f5a/data/phosh.service.

[34] *The Flutter system channel for controlling compositor text fields.* URL: https://api.flutter.dev/flutter/services/SystemChannels/textInput-constant.html.

[35] *The Flutter system channel for notifying about keyboard events.* URL: https://api.flutter.dev/flutter/services/SystemChannels/keyEvent-constant.html.

[36] *The Git repository of the settings application of the Zenith compositor.* URL: https://github.com/roscale/zenith_settings.

[37] *The XML definition of the Wayland protocol.* URL: https://gitlab.freedesktop.org/wayland/wayland/-/blob/1.16/protocol/wayland.xml.

[38] *Wayland Client Library.* URL: https://gitlab.freedesktop.org/wayland/wayland/-/tree/main/src.

[39] *Wayland protocol for virtual keyboards.* URL: https://wayland.app/protocols/text-input-unstable-v3.

[40] *wlroots, a modular Wayland compositor library.* URL: https://gitlab.freedesktop.org/wlroots/wlroots.

[41] *XDG Shell, a Wayland protocol which describes the semantics of windows and popups.* URL: https://wayland.app/protocols/xdg-shell.