The Wayland Protocol

Print this book

Introduction

Wayland is the next-generation display server for Unix-like systems, designed and built by the alumni of the venerable Xorg server, and is the best way to get your application windows onto your user's screens. Readers who have worked with X11 in the past will be pleasantly surprised by Wayland's improvements, and those who are new to graphics on Unix will find it a flexible and powerful system for building graphical applications and desktops.

This book will help you establish a firm understanding of the concepts, design, and implementation of Wayland, and equip you with the tools to build your own Wayland client and server applications. Over the course of your reading, we'll build a mental model of Wayland and establish the rationale that went into its design. Within these pages you should find many "aha!" moments as the intuitive design choices of Wayland become clear, which should help to keep the pages turning. Welcome to the future of open source graphics!

Notice: this is a *draft*. Chapters 1-10 are more or less complete, but may be updated later. Chapters 11 forward in large part remain to be written.

TODO

- Expand on resource lifetimes and avoiding race conditions in chapter 2.4
- Move linux-dmabuf details to the appendix, add note about wl_drm & Mesa
- Rewrite the introduction text
- Add example code for interactive move, to demonstrate the use of serials
- Prepare PDFs and EPUBs

About the book

About the author

In the words of Preston Carpenter, a close collaborator of Drew's:

Drew DeVault got his start in the Wayland world by building sway, a clone of the popular tiling window manager i3. It is now the most popular tiling Wayland compositor by any measure: users, commit count, and influence. Following its success, Drew gave back to the Wayland community by starting wlroots: unopinionated, composable modules for building a Wayland compositor. Today it is the foundation for dozens of independent compositors, and Drew is one of the foremost experts in Wayland.

High-level design

Your computer has *input* and *output* devices, which respectively are responsible for receiving information from you and displaying information to you. These input devices take the form of, for example:

- Keyboards
- Mice
- Touchpads
- Touch screens
- Drawing tablets

Your output devices generally take the form of displays, on your desk or your laptop or mobile device. These resources are shared between all of your applications, and the role of the **Wayland compositor** is to dispatch input events to the appropriate **Wayland client** and to display their windows in their appropriate place on your outputs. The process of bringing together all of your application windows for display on an output is called *compositing* — and thus we call the software which does this the *compositor*.

In practice

There are many distinct software components in desktop ecosystem. There are tools like Mesa for rendering (and each of its drivers), the Linux KMS/DRM subsystem, buffer allocation with GBM, the userspace libdrm library, libinput and evdev, and much more still. Don't worry — expertise with most of these systems is not required for understanding Wayland, and in any case are largely beyond the scope of this book. In fact, the Wayland protocol is quite conservative and abstract, and a Wayland-based desktop could easily be built & run most applications without implicating any of this software. That being said, a surface-level understanding of what these pieces are and how they work is useful. Let's start from the bottom and work our way up.

The hardware

A typical computer is equipped with a few important pieces of hardware. Outside of the box, we have your displays, keyboard, mouse, perhaps some speakers and a cute USB cup warmer. There are several components *inside* the box for interfacing with these devices. Your keyboard and mouse, for example, are probably plugged into USB ports, for which your system has a dedicated USB controller. Your displays are plugged into your GPU.

These systems have their own jobs and state. For example, your GPU has state in the form of memory for storing pixel buffers in, and jobs like *scanning out* these buffers to your displays. Your GPU also provides a processor which is specially tuned to be good at highly parallel jobs (such as calculating the right color for each of the 2,073,600 pixels on a 1080p display), and bad at everything else. The USB controller has the unenviable job of implementing the legendarily dry USB specification for receiving input events from your keyboard, or instructing your coaster to assume a temperature carefully selected to at once avoid lawsuits and frustrate you with cold coffee.

At this level, your hardware has little concept of what applications are running on your system. The hardware provides an interface with which it can be commanded to perform work, and does what it's told — regardless of who tells it so. For this reason, only one component is allowed to talk to it...

The kernel

This responsibility falls onto the kernel. The kernel is a complex beast, so we'll focus on only the parts which are relevant to Wayland. Linux's job is to provide an abstraction over your hardware, so that it can be safely accessed by *userspace* — where our Wayland compositors run. For graphics, this is called the **DRM**, or *direct rendering manager*, which efficiently tasks the GPU with work from userspace. An important subsystem of DRM is **KMS**, or *kernel mode setting*, for enumerating your displays and setting properties such as their selected resolution (also known as their "mode"). Input devices are abstracted through an interface called **evdev**.

Most kernel interfaces are made available to userspace by way of special files in `/dev`. In the case of DRM, these files are in `/dev/dri/`, usually in the form of a primary node (e.g. `card0`) for privileged operations like modesetting, and a render node (e.g. `renderD128`), for unprivileged operations like rendering or video decoding. For evdev, the "device nodes" are `/dev/input/event*`.

Userspace

Now, we enter userspace. Here, applications are isolated from the hardware and must work via the device nodes provided by the kernel.

libdrm

Most Linux interfaces have a userspace counterpart which provides a pleasant(ish) C API for working with these device nodes. One such library is libdrm, which is the userspace portion of the DRM subsystem. libdrm is used by Wayland compositors to do modesetting and other DRM operations, but is generally not used by Wayland clients directly.

Mesa

Mesa is one of the most important parts of the Linux graphics stack. It provides, among other things, vendor-optimized implementations of OpenGL (and Vulkan) for Linux and the **GBM** (Generic Buffer Management) library — an abstraction on top of libdrm for allocating buffers on the GPU. Most Wayland compositors will use both GBM and OpenGL via Mesa, and most Wayland clients will use at least its OpenGL or Vulkan implementations.

libinput

Like libdrm abstracts the DRM subsystem, libinput provides the userspace end of evdev. It's responsible for receiving input events from the kernel from your various input devices, decoding them into a usable form, and passing them on to the Wayland compositor. The Wayland compositor requires special permissions to use the evdev files, forcing Wayland clients to go through the compositor to receive input events — which, for example, prevents keylogging.

(e)udev

Dealing with the appearance of new devices from the kernel, configuring permissions for the resulting device nodes in `/dev`, and sending word of these changes to applications running on your system, is a responsibility that falls onto userspace. Most systems use udev (or eudev, a fork) for this purpose. Your Wayland compositor uses udev to enumerate input devices and GPUs, and to receive notifications when new ones appear or old ones are unplugged.

xkbcommon

XKB, short for X keyboard, is the original keyboard handling subsystem of the Xorg server. Several years ago, it was extracted from the Xorg tree and made into an independent library for keyboard handling, and it no longer has any practical relationship with X. Libinput (along with the Wayland compositor) delivers keyboard events in the form of scancodes, whose precise meaning varies from keyboard to keyboard. It's the responsibility of xkbcommon to translate these scan codes into meaningful and generic key "symbols" — for example, converting `65` to `XKB_KEY_Space`. It also contains a state machine which knows that pressing "1" while shift is held emits "!".

pixman

A simple library used by clients and compositors alike for efficiently manipulating pixel buffers, doing math with intersecting rectangles, and performing other similar **pix** el **man** ipulation tasks.

libwayland

libwayland the most commonly used implementation of the Wayland protocol, is written in C, and handles much of the low-level wire protocol. It also provides a tool which generates high-level code from Wayland protocol definitions (which are XML files). We will be discussing libwayland in detail in chapter 1.3, and throughout this book.

...and all the rest.

Each of the pieces mentioned so far are consistently found throughout the Linux desktop ecosystem. Beyond this, more components exist. Many graphical applications don't know about Wayland at all, choosing instead to allow libraries like GTK+, Qt, SDL, and GLFW — among many others — to deal with it. Many compositors choose software like wlroots to abstract more of their responsibilities, while others implement everything in-house.

Goals & target audience

Our goal is for you to come away from this book with an understanding of the Wayland protocol and its high-level usage. You should have a solid understanding of everything in the core Wayland protocol, as well as the knowledge necessary to evaluate and implement the various protocol extensions necessary for its productive use. Primarily, this book uses the concerns of a Wayland client to frame its presentation of Wayland. However, it should provide some utility for those working on Wayland compositors as well.

The free desktop ecosystem is complex and built from many discrete parts. We are going to discuss these pieces very little — you won't find information here about leveraging libdrm in your Wayland compositor, or using libinput to process evdev events. Thus, this book is not a comprehensive guide for building Wayland compositors. We're also not going to talk about drawing technologies which are useful for Wayland clients, such as Cairo, Pango, GTK+, and so on, and thus nor is this a robust guide for the practical Wayland client implementation. Instead, we focus only on the particulars of Wayland.

This book only covers the protocol and libwayland. If you are writing a client and are already familiar with your favorite user interface rendering library, bring your own pixels and we'll help you display them on Wayland. If you already have an understanding of the technologies required to operate displays and input devices for your compositor, this book will help you learn how to talk to clients.

What's in the Wayland package

When you install "wayland" in your Linux distribution, you'll most likely end up with the freedesktop.org distribution of libwayland-client, libwayland-server, wayland-scanner, and wayland.xml. Respectively, these will probably be in /usr/lib & /usr/include, /usr/bin, and /usr/share/wayland/. This package represents the most popular *implementation* of the Wayland protocol, but it is not the only one. Chapter 3 covers this implementation of Wayland in detail; the rest of the book is equally applicable to any implementation.

wayland.xml

Wayland protocols are defined by XML files. If you locate and open "wayland.xml" in your favorite text editor, you will find the XML specification for the "core" Wayland protocol. This is a high-level protocol — built on top of the wire protocol that we'll discuss in the next chapter. Most of this book is devoted to explaining this file.

wayland-scanner

The "wayland-scanner" tool is used to process these XML files and generate code from them. The most common implementation is the one you're examining now, and it can be used to generate C headers & glue code from XML files like wayland.xml. Other scanners exist for other programming languages, notably wayland-rs (Rust), waymonadscanner (Haskell), and more.

libwayland

The two libraries, libwayland-client and libwayland-server, include an implementation of the wire protocol for each end of the connection. Some common utilities are offered for working with Wayland data structures, a simple event loop, and so on. Additionally, these libraries contain a pre-compiled copy of the core Wayland protocol as generated by `wayland-scanner`.

Protocol design

The Wayland protocol is built from several layers of abstraction. It starts with a basic wire protocol format, which is a stream of messages decodable with interfaces agreed upon in advance. Then we have higher level procedures for enumerating interfaces, creating resources which conform to these interfaces, and exchanging messages about them — the Wayland protocol and its extensions. On top of this we have some broader patterns which are frequently used in Wayland protocol design. We'll cover all of these in this chapter.

Let's work our way from the bottom-up.

Wire protocol basics

Note: If you're just going to use libwayland, this chapter is optional - feel free to skip to chapter 2.2.

The wire protocol is a stream of 32-bit values, encoded with the host's byte order (e.g. little-endian on x86 family CPUs). These values represent the following primitive types:

int, uint: 32-bit signed or unsigned integer.

fixed: 24.8 bit signed fixed-point numbers.

object: 32-bit object ID.

new_id: 32-bit object ID which allocates that object when received.

In addition to these primitives, the following other types are used:

string: A string, prefixed with a 32-bit integer specifying its length (in bytes), followed by the string contents and a NUL terminator, padded to 32 bits with undefined data. The encoding is not specified, but in practice UTF-8 is used.

array: A blob of arbitrary data, prefixed with a 32-bit integer specifying its length (in bytes), then the verbatim contents of the array, padded to 32 bits with undefined data.

fd: 0-bit value on the primary transport, but transfers a file descriptor to the other end using the ancillary data in the Unix domain socket message (msg_control).

enum: A single value (or bitmap) from an enumeration of known constants, encoded into a 32-bit integer.

Messages

The wire protocol is a stream of messages built with these primitives. Every message is an event (in the case of server to client messages) or request (client to server) which acts upon an *object*.

The message header is two words. The first word is the affected object ID. The second is two 16-bit values; the upper 16 bits are the size of the message (including the header itself) and the lower 16 bits are the event or request opcode. The message arguments follow, based on a message signature agreed upon in advance by both parties. The recipient looks up the object ID's interface and the event or request defined by its opcode to determine the signature and nature of the message.

To understand a message, the client and server have to establish the objects in the first place. Object ID 1 is pre-allocated as the Wayland display singleton, and can be used to bootstrap other objects. We'll discuss this in chapter 4. The next chapter goes over what an interface is, and how requests and events work, assuming you've already negotiated an object ID. Speaking of which...

Object IDs

When a message comes in with a `new_id` argument, the sender allocates an object ID for it — the interface used for this object is established through additional arguments, or agreed upon in advance for that request/event. This object ID can be used in future messages, either as the first word of the header, or as an `object_id` argument. The client allocates IDs in the range of `[1, 0xFEFFFFFF]`, and the server allocates IDs in the range of `[0xFF000000, 0xFFFFFFFF]`. IDs begin at the lower end of this bound and increment with each new object allocation.

An object ID of 0 represents a null object; that is, a non-existent object or the explicit lack of an object.

Transports

To date all known Wayland implementations work over a Unix domain socket. This is used for one reason in particular: file descriptor messages. Unix sockets are the most practical transport capable of transferring file descriptors between processes, and this is necessary for large data transfers (keymaps, pixel buffers, and clipboard contents being the main use-cases). In theory, a different transport (e.g. TCP) is possible, but someone would have to figure out an alternative way of transferring bulk data.

To find the Unix socket to connect to, most implementations just do what libwayland does:

1. If `WAYLAND_SOCKET` is set, interpret it as a file descriptor number on which the connection is already established, assuming that the parent process configured the connection for us.
2. If `WAYLAND_DISPLAY` is set, concat with `XDG_RUNTIME_DIR` to form the path to the Unix socket.
3. Assume the socket name is `wayland-0` and concat with `XDG_RUNTIME_DIR` to form the path to the Unix socket.
4. Give up.

Interfaces, requests, and events

The Wayland protocol works by issuing *requests* and *events* that act on *objects*. Each object has an *interface* which defines what requests and events are possible, and the *signature* of each. Let's consider an example interface: `wl_surface`.

Requests

A surface is a box of pixels that can be displayed on-screen. It's one of the primitives we build things like application windows out of. One of its *requests*, sent from the client to the server, is "damage", which the client uses to indicate that some part of the surface has changed and needs to be redrawn. Here's an annotated example of a "damage" message on the wire (in hexadecimal):

```
0000000A    Object ID (10)
00180002    Message length (24) and request opcode (2)
00000000    X coordinate (int): 0
00000000    Y coordinate (int): 0
00000100    Width        (int): 256
00000100    Height       (int): 256
```

This is a snippet of a session — the surface was allocated earlier and assigned an ID of 10. When the server receives this message, it looks up the object with ID 10 and finds that it's a `wl_surface` instance. Knowing this, it looks up the signature for the request with opcode 2. It then knows to expect four integers as the arguments, and it can decode the message and dispatch it for processing internally.

Events

The server can also send messages back to the client — events. One event that the server can send regarding a `wl_surface` is "enter", which it sends when that surface is being displayed on a specific output (the client might respond to this, for example, by adjusting its scale factor for a HiDPI display). Here's an example of such a message:

```
0000000A    Object ID (10)
000C0000    Message length (12) and event opcode (0)
00000005    Output (object ID): 5
```

This message references another object, by its ID: the `wl_output` object which the surface is being shown on. The client receives this and dances to a similar tune as the server did. It looks up object 10, associates it with the `wl_surface` interface, and looks up the signature of the event corresponding to opcode 0. It decodes the rest of the message accordingly, looking up the `wl_output` with ID 5 as well, then dispatches it for processing internally.

Interfaces

The interfaces which define the list of requests and events, the opcodes associated with each, and the signatures with which you can decode the messages, are agreed upon in advance. I'm sure you're dying to know how — simply turn the page to end the suspense.

The high-level protocol

In chapter 1.3, I mentioned that `wayland.xml` is probably installed with the Wayland package on your system. Find and pull up that file now in your favorite text editor. It's through this file, and others like it, that we define the interfaces supported by a Wayland client or server.

Each interface is defined in this file, along with its requests and events, and their respective signatures. We use XML, everyone's favorite file format, for this purpose. Let's look at the examples we discussed in the previous chapter for `wl_surface`. Here's a sample:

```
<interface name="wl_surface" version="4">
  <request name="damage">
    <arg name="x" type="int" />
    <arg name="y" type="int" />
    <arg name="width" type="int" />
    <arg name="height" type="int" />
  </request>

  <event name="enter">
    <arg name="output" type="object" interface="wl_output" />
  </event>
</interface>
```

Note: I've trimmed this snippet for brevity, but if you have the `wayland.xml` file in front of you, seek out this interface and examine it yourself — included is additional documentation explaining the purpose and precise semantics of each request and event.

When processing this XML file, we assign each request and event an opcode in the order that they appear, numbered from zero and incrementing independently. Combined with the list of arguments, you can decode the request or event when it comes in over the wire, and based on the documentation shipped in the XML file you can decide how to program your software to behave accordingly. This usually comes in the form of code generation — we'll talk about how libwayland does this in chapter 3.

Starting from chapter 4, most of the remainder of this book is devoted to explaining this file, as well as some supplementary protocol extensions.

Protocol design patterns

There are some key concepts which have been applied in the design of most Wayland protocols that we should briefly cover here. These patterns are found throughout the high-level Wayland protocol and protocol extensions (well, in the Wayland protocol, at least[1]). If you find yourself writing your own protocol extensions, you'd be wise to apply these patterns yourself.

Atomicity

The most important of the Wayland protocol design patterns is *atomicity*. A stated goal of Wayland is "every frame is perfect". To this end, most interfaces allow you to update them transactionally, using several requests to build up a new representation of its state, then committing them all at once. For example, there are several properties that may be configured on a `wl_surface`:

- An attached pixel buffer
- A damaged area that needs to be redrawn
- A region defined as opaque, for optimization purposes
- A region where input events are acceptable
- A transformation, like rotating 90 degrees
- A buffer scale, used for HiDPI

The interface includes separate requests for configuring each of these, but these are applied to a *pending* state. Only when the **commit** request is sent does the pending state get merged into the *current* state, allowing you to atomically update all of these properties within a single frame. Combined with a few other key design decisions, this allows

Wayland compositors to render everything perfectly in every frame — no tearing or partially updated windows, just every pixel in its place and every place in its pixel.

Resource lifetimes

Another important design pattern is avoiding a situation where the server or client is sending events or requests that pertain to an invalid object. For this reason, interfaces which define resources that have finite lifetimes will often include requests and events through which the client or server can state their intention to no longer send requests or events for that object. Only once both sides have agreed to this — asynchronously — do they destroy the resources they allocated for that object.

Wayland is a fully asynchronous protocol. Messages are guaranteed to arrive in the order they were sent, but only with respect to one sender. For example, the server may have several input events queued up when the client decides to destroy its keyboard device. The client must correctly deal with events for an object it no longer needs until the server catches up. Likewise, had the client queued up some requests for an object before destroying it, it would have had to send these requests in the correct order so that the object is no longer used after the client agreed it had been destroyed.

Versioning

There are two versioning models in use in Wayland protocols: unstable and stable. Both models only allow for backwards-compatible changes, but when a protocol transitions from unstable to stable, one last breaking change is permitted. This gives protocols an incubation period during which we can test them in practice, then apply our insights in one last big set of breaking changes to make a protocol that should stand the test of time[2].

To make a backwards-compatible change, you may only add new events or requests to the end of an interface, or new members to the end of an enum. Additionally, each implementation must limit itself to using only the messages supported by the other end of the connection. We'll discuss in chapter 5 how we establish which versions of each interface are in use by each party.

1

Except for that one interface. Look, at least we tried, right?

2

Note that many useful protocols are still unstable at the time of writing. They may be a little kludgy, but they still see widespread use, which is why backwards compatibility is important. When promoting a protocol from unstable to stable, it's done in a way that allows software to support both the unstable and stable protocols simultaneously, allowing for a smoother transition.

The libwayland implementation

We spoke briefly about libwayland in chapter 1.3 — the most popular Wayland implementation. Much of this book is applicable to any implementation, but we're going to spend the next two chapters familiarizing you with this one.

The Wayland package includes pkg-config specs for wayland-client and wayland-server — consult your build system's documentation for instructions on linking with them. Naturally, most applications will only link to one or the other. The library includes a few simple primitives (such as a linked list) and a pre-compiled version of `wayland.xml` — the core Wayland protocol.

We'll start by introducing the primitives.

wayland-util primitives

Common to both the client and server libraries is `wayland-util.h`, which defines a number of structs, utility functions, and macros that establish a handful of primitives for use in Wayland applications. Among these are:

- Structures for marshalling & unmarshalling Wayland protocol messages in generated code
- A linked list (`wl_list`) implementation
- An array (`wl_array`) implementation (rigged up to the corresponding Wayland primitive)
- Utilities for conversion between Wayland scalars (such as fixed-point numbers) and C types
- Debug logging facilities to bubble up information from libwayland internals

The header itself contains many comments with quite good documentation — you should read through them yourself. We'll go into detail on how to apply these primitives in the next several pages.

wayland-scanner

The Wayland package comes with one binary: `wayland-scanner`. This tool is used to generate C headers & glue code from the Wayland protocol XML files discussed in chapter 2.3. This tool is used in the "wayland" package's build process to pre-generate headers & glue code for the core protocol, `wayland.xml`. The headers become `wayland-client-protocol.h` and `wayland-server-protocol.h` — though you normally include `wayland-client.h` and `wayland-server.h` instead of using these directly.

The usage of this tool is fairly simple (and summarized by `wayland-scanner -h`), but can be summed up as follows. To generate a client header:

```
wayland-scanner client-header < protocol.xml > protocol.h
```

To generate a server header:

```
wayland-scanner server-header < protocol.xml > protocol.h
```

And to generate the glue code:

```
wayland-scanner private-code < protocol.xml > protocol.c
```

Different build systems will have different approaches to configuring custom commands — consult your build system's docs. Generally speaking, you'll want to run `wayland-scanner` at build time, then compile and link your application to the glue code.

Go ahead and do this with any Wayland protocol now, if you have one handy (`wayland.xml` is probably available in `/usr/share/wayland`, for example). Open up the glue code & header and consult it as you read the following chapters, to understand how the primitives offered by libwayland are applied in practice in the generated code.

Proxies & resources

An *object* is an entity known to both the client and server that has some state, changes to which are negotiated over the wire. On the client side, libwayland refers to these objects through the `wl_proxy` interface. These are a concrete C-friendly "proxy" for the abstract object, and provides functions which are indirectly used by the client to marshall requests into the wire format. If you review the wayland-client-core.h file, you'll find a few low-level functions for this purpose. Generally, you don't use these directly.

On the server, objects are abstracted through wl_resource, which is fairly similar, but have an extra degree of complexity — the server has to track which object belongs to which client. Each wl_resource is owned by a single client. Aside from this, the interface is much the same, and provides low-level abstraction for marshalling events to send to the associated client. You will use wl_resource directly on a server more often than you'll use directly interface with wl_proxy on a client. One example of such a use is to obtain a reference to the wl_client which owns a resource that you're manipulating out-of-context, or send a protocol error when the client attempts an invalid operation.

Another level up is another set of higher-level interfaces, which most Wayland clients & servers interact with to accomplish a majority of their tasks. We will look at them in the next section.

Interfaces & listeners

Finally, we reach the summit of libwayland's abstractions: interfaces and listeners. The ideas discussed in previous chapters — wl_proxy and wl_resource, and the primitives — are singular implementations which live in libwayland, and they exist to provide support to this layer. When you run an XML file through wayland-scanner, it generates *interfaces* and *listeners*, as well as glue code between them and the low-level wire protocol interfaces, all specific to each interface in the high-level protocols.

Recall that each actor on a Wayland connection can both receive and send messages. A client is listening for events and sending requests, and a server listens for requests and sends events. Each side listens for the messages of the other using an aptly-named wl_listener. Here's an example of this interface:

```
struct wl_surface_listener {
    /** surface enters an output */
    void (*enter)(void *data,
              struct wl_surface *wl_surface,
              struct wl_output *output);

    /** surface leaves an output */
    void (*leave)(void *data,
              struct wl_surface *wl_surface,
              struct wl_output *output);
};
```

This is a client-side listener for a wl_surface. The XML that wayland-scanner uses to generate this is:

```
<interface name="wl_surface" version="4">
  <event name="enter">
    <arg name="output"
      type="object"
      interface="wl_output"/>
  </event>

  <event name="leave">
    <arg name="output"
      type="object"
      interface="wl_output"/>
  </event>
  <!-- additional details omitted for brevity -->
</interface>
```

It should be fairly clear how these events become a listener interface. Each function pointer takes some arbitrary user data, a reference to the resource which the event pertains to, and the arguments to that event. We can bind a listener to a `wl_surface` like so:

```
static void wl_surface_enter(void *data,
        struct wl_surface *wl_surface, struct wl_output *output) {
    // ...
}

static void wl_surface_leave(void *data,
        struct wl_surface *wl_surface, struct wl_output *output) {
    // ...
}

static const struct wl_surface_listener surface_listener = {
    .enter = wl_surface_enter,
    .leave = wl_surface_leave,
};

// ...

struct wl_surface *surf;
wl_surface_add_listener(surf, &surface_listener, NULL);
```

The `wl_surface` interface also defines some requests that the client can make for that surface:

```
<interface name="wl_surface" version="4">
  <request name="attach">
    <arg name="buffer"
      type="object"
      interface="wl_buffer"
      allow-null="true"/>
    <arg name="x" type="int"/>
    <arg name="y" type="int"/>
  </request>
  <!-- additional details omitted for brevity -->
</interface>
```

wayland-scanner generates the following prototype, as well as glue code which marshalls this message.

```
void wl_surface_attach(struct wl_surface *wl_surface,
    struct wl_buffer *buffer, int32_t x, int32_t y);
```

The server-side code for interfaces and listeners is identical, but reversed — it generates listeners for requests and glue code for events. When libwayland receives a message, it looks up the object ID, and its interface, then uses that to decode the rest of the message. Then it looks for listeners on this object and invokes your functions with the arguments to the message.

That's all there is to it! It took us a couple of layers of abstraction to get here, but you should now understand how an event starts in your server code, becomes a message on the wire, is understood by the client, and dispatched to your client code. There remains one unanswered question, however. All of this presupposes that you already have references to Wayland objects. How do you get those?

The Wayland display

Up to this point, we've left a crucial detail out of our explanation of how the Wayland protocol manages joint ownership over objects between the client and server: how those objects are created in the first place. The Wayland display, or `wl_display`, implicitly exists on every Wayland connection. It has the following interface:

```
<interface name="wl_display" version="1">
  <request name="sync">
    <arg name="callback" type="new_id" interface="wl_callback"
       summary="callback object for the sync request"/>
  </request>

  <request name="get_registry">
    <arg name="registry" type="new_id" interface="wl_registry"
      summary="global registry object"/>
  </request>

  <event name="error">
    <arg name="object_id" type="object" summary="object where the error occurred"/>
    <arg name="code" type="uint" summary="error code"/>
    <arg name="message" type="string" summary="error description"/>
  </event>

  <enum name="error">
    <entry name="invalid_object" value="0" />
    <entry name="invalid_method" value="1" />
    <entry name="no_memory" value="2" />
    <entry name="implementation" value="3" />
  </enum>

  <event name="delete_id">
    <arg name="id" type="uint" summary="deleted object ID"/>
  </event>
</interface>
```

The most interesting of these for the average Wayland user is `get_registry`, which we'll talk about in detail in the following chapter. In short, the registry is used to allocate other objects. The rest of the interface is used for housekeeping on the connection, and are generally not important unless you're writing your own libwayland replacement.

Instead, this chapter will focus on a number of functions that libwayland associates with the `wl_display` object, for establishing and maintaining your Wayland connection. These are used to manipulate libwayland's internal state, rather than being directly related to wire protocol requests and events.

We'll start with the most important of these functions: establishing the display. For clients, this will cover the actual process of connecting to the server, and for servers, the process of configuring a display for clients to connect to.

Creating a display

Fire up your text editor — it's time to write our first lines of code.

For Wayland clients

Connecting to a Wayland server and creating a `wl_display` to manage the connection's state is quite easy:

```
#include <stdio.h>
#include <wayland-client.h>

int
main(int argc, char *argv[])
{
    struct wl_display *display = wl_display_connect(NULL);
    if (!display) {
        fprintf(stderr, "Failed to connect to Wayland display.\n");
        return 1;
    }
    fprintf(stderr, "Connection established!\n");

    wl_display_disconnect(display);
    return 0;
}
```

Let's compile and run this program. Assuming you're using a Wayland compositor as you read this, the result should look like this:

```
$ cc -o client client.c -lwayland-client
$ ./client
Connection established!
```

`wl_display_connect` is the most common way for clients to establish a Wayland connection. The signature is:

```
struct wl_display *wl_display_connect(const char *name);
```

The "name" argument is the name of the Wayland display, which is typically `"wayland-0"`. You can swap the `NULL` for this in our test client and try for yourself — it's likely to work. This corresponds to the name of a Unix socket in `$XDG_RUNTIME_DIR`. `NULL` is preferred, however, in which case libwayland will:

1. If `$WAYLAND_DISPLAY` is set, attempt to connect to `$XDG_RUNTIME_DIR/$WAYLAND_DISPLAY`
2. Otherwise, attempt to connect to `$XDG_RUNTIME_DIR/wayland-0`
3. Otherwise, fail :(

This allows users to specify the Wayland display they want to run their clients on by setting `$WAYLAND_DISPLAY` to the desired display. If you have more complex requirements, you can also establish the connection yourself and create a Wayland display from a file descriptor:

```
struct wl_display *wl_display_connect_to_fd(int fd);
```

You can also obtain the file descriptor that the `wl_display` is using via `wl_display_get_fd`, regardless of how you created the display.

```
int wl_display_get_fd(struct wl_display *display);
```

For Wayland servers

The process is fairly simple for servers as well. The creation of the display and binding to a socket are separate, to give you time to configure the display before any clients are able to connect to it. Here's another minimal example program:

```
#include <stdio.h>
#include <wayland-server.h>

int
main(int argc, char *argv[])
{
    struct wl_display *display = wl_display_create();
    if (!display) {
        fprintf(stderr, "Unable to create Wayland display.\n");
        return 1;
    }

    const char *socket = wl_display_add_socket_auto(display);
    if (!socket) {
        fprintf(stderr, "Unable to add socket to Wayland display.\n");
        return 1;
    }

    fprintf(stderr, "Running Wayland display on %s\n", socket);
    wl_display_run(display);

    wl_display_destroy(display);
    return 0;
}
```

Let's compile and run this, too:

```
$ cc -o server server.c -lwayland-server
$ ./server &
Running Wayland display on wayland-1
$ WAYLAND_DISPLAY=wayland-1 ./client
Connection established!
```

Using `wl_display_add_socket_auto` will allow libwayland to decide the name for the display automatically, which defaults to `wayland-0`, or `wayland-$n`, depending on whether any other Wayland compositors have sockets in `$XDG_RUNTIME_DIR`. However, as with the client, you have some other options for configuring the display:

```
int wl_display_add_socket(struct wl_display *display, const char *name);

int wl_display_add_socket_fd(struct wl_display *display, int sock_fd);
```

After adding the socket, calling `wl_display_run` will run libwayland's internal event loop and block until `wl_display_terminate` is called. What's this event loop? Turn the page and find out!

Incorporating an event loop

libwayland provides its own event loop implementation for Wayland servers to take advantage of, but the maintainers have acknowledged this as a design overstep. For clients, there is no such equivalent. Regardless, the Wayland server event loop is useful enough, even if it's out-of-scope.

Wayland server event loop

Each `wl_display` created by libwayland-server has a corresponding `wl_event_loop`, which you may obtain a reference to with `wl_display_get_event_loop`. If you're writing a new Wayland compositor, you will likely want to use this as your only event loop. You can add file descriptors to it with `wl_event_loop_add_fd`, and timers with `wl_event_loop_add_timer`. It also handles signals via `wl_event_loop_add_signal`, which can be pretty convenient.

With the event loop configured to your liking to monitor all of the events your compositor has to respond to, you can process events and dispatch Wayland clients all at once by calling `wl_display_run`, which will process the event loop and block until the display terminates (via `wl_display_terminate`). Most Wayland compositors which were built from the ground-up with Wayland in mind (as opposed to being ported from X11) use this approach.

However, it's also possible to take the wheel and incorporate the Wayland display into your own event loop. `wl_display` uses the event loop internally for processing clients, and you can choose to either monitor the Wayland event loop on your own, dispatching it as necessary, or you can disregard it entirely and manually process client updates. If you wish to allow the Wayland event loop to look after itself and treat it as subservient to your own event loop, you can use `wl_event_loop_get_fd` to obtain a poll-able file descriptor, then call `wl_event_loop_dispatch` to process events when activity occurs on that file descriptor. You will also need to call `wl_display_flush_clients` when you have data which needs writing to clients.

Wayland client event loop

libwayland-client, on the other hand, does not have its own event loop. However, since there is only generally one file descriptor, it's easier to manage without. If Wayland events are the only sort which your program expects, then this simple loop will suffice:

```
while (wl_display_dispatch(display) != -1) {
    /* This space deliberately left blank */
}
```

However, if you have a more sophisticated application, you can build your own event loop in any manner you please, and obtain the Wayland display's file descriptor with `wl_display_get_fd`. Upon `POLLIN` events, call `wl_display_dispatch` to process incoming events. To flush outgoing requests, call `wl_display_flush`.

Almost there!

At this point you have all of the context you need to set up a Wayland display and process events and requests. The only remaining step is to allocate objects to chat about with the other side of your connection. For this, we use the registry. At the end of the next chapter, we will have our first useful Wayland client!

Globals & the registry

If you'll recall from chapter 2.1, each request and event is associated with an object ID, but thus far we haven't discussed how objects are created. When we receive a Wayland message, we must know what interface the object ID represents to decode it. We must also somehow negotiate available objects, the creation of new ones, and the assigning of IDs to them, in some manner. In Wayland we solve both of these problems at once — when we *bind* an object ID, we agree on the interface used for it in all future messages, and stash a mapping of object IDs to interfaces in our local state.

In order to bootstrap these, the server offers a list of *global* objects. These globals often provide information and functionality on their own merits, but most often they're used to broker additional objects to fulfill various purposes, such as the creation of application windows. These globals themselves also have their own object IDs and interfaces, which we have to somehow assign and agree upon.

With questions of chickens and eggs no doubt coming to mind by now, I'll reveal the secret trick: object ID 1 is already implicitly assigned to the `wl_display` interface when you make the connection. As you recall the interface, take note of the `wl_display::get_registry` request:

```
<interface name="wl_display" version="1">
  <request name="sync">
    <arg name="callback" type="new_id" interface="wl_callback" />
  </request>

  <request name="get_registry">
    <arg name="registry" type="new_id" interface="wl_registry" />
  </request>

  <!-- ... -->
</interface>
```

The `wl_display::get_registry` request can be used to bind an object ID to the `wl_registry` interface, which is the next one found in `wayland.xml`. Given that the `wl_display` always has object ID 1, the following wire message ought to make sense (in big-endian):

```
C->S    00000001 000C0001 00000002            .... .... ....
```

When we break this down, the first number is the object ID. The most significant 16 bits of the second number are the total length of the message in bytes, and the least significant bits are the request opcode. The remaining words (just one) are the arguments. In short, this calls request 1 (0-indexed) on object ID 1 (the `wl_display`), which accepts one argument: a generated ID for a new object. Note in the XML documentation that this new ID is defined ahead of time to be governed by the `wl_registry` interface:

```
<interface name="wl_registry" version="1">
  <request name="bind">
    <arg name="name" type="uint" />
    <arg name="id" type="new_id" />
  </request>

  <event name="global">
    <arg name="name" type="uint" />
    <arg name="interface" type="string" />
    <arg name="version" type="uint" />
  </event>

  <event name="global_remove">
    <arg name="name" type="uint" />
  </event>
</interface>
```

It is this interface which we'll discuss in the following chapters.

Binding to globals

Upon creating a registry object, the server will emit the `global` event for each global available on the server. You can then bind to the globals you require.

Binding is the process of taking a known object and assigning it an ID. Once the client binds to the registry like this, the server emits the `global` event several times to advertise which interfaces it supports. Each of these globals is assigned a unique `name`, as an unsigned integer. The `interface` string maps to the name of the interface found in the protocol: `wl_display` from the XML above is an example of such a name. The version number is also defined here — for more information about interface versioning, see appendix C.

To bind to any of these interfaces, we use the bind request, which works similarly to the magical process by which we bound to the `wl_registry`. For example, consider this wire protocol exchange:

```
C->S    00000001 000C0001 00000002            .... .... ....

S->C    00000002 001C0000 00000001 00000007   .... .... .... ....
        776C5f73 686d0000 00000001            wl_s hm.. ....
        [...]

C->S    00000002 00100000 00000001 00000003   .... .... .... ....
```

The first message is identical to the one we've already dissected. The second one is an event from the server: object 2 (which the client assigned the `wl_registry` to in the first message) opcode 0 ("global"), with arguments 1, "wl_shm", and 1 — respectively the name, interface, and version of this global. The client responds by calling opcode 0 on object ID 2 (`wl_registry::bind`) and assigns object ID `3` to global name `1` — *binding* to the `wl_shm` global. Future events and requests for this object are defined by the `wl_shm` protocol, which you can find in `wayland.xml`.

Once you've created this object, you can utilize its interface to accomplish various tasks — in the case of `wl_shm`, managing shared memory between the client and server. Most of the remainder of this book is devoted to explaining the usage of each of these globals.

Armed with this information, we can write our first useful Wayland client: one which simply prints all of the globals available on the server.

```
#include <stdint.h>
#include <stdio.h>
#include <wayland-client.h>

static void
registry_handle_global(void *data, struct wl_registry *registry,
        uint32_t name, const char *interface, uint32_t version)
```

```
{
    printf("interface: '%s', version: %d, name: %d\n",
            interface, version, name);
}

static void
registry_handle_global_remove(void *data, struct wl_registry *registry,
        uint32_t name)
{
    // This space deliberately left blank
}

static const struct wl_registry_listener
registry_listener = {
    .global = registry_handle_global,
    .global_remove = registry_handle_global_remove,
};

int
main(int argc, char *argv[])
{
    struct wl_display *display = wl_display_connect(NULL);
    struct wl_registry *registry = wl_display_get_registry(display);
    wl_registry_add_listener(registry, &registry_listener, NULL);
    wl_display_roundtrip(display);
    return 0;
}
```

Feel free to reference previous chapters to interpret this program. We connect to the display (chapter 4.1), obtain the registry (this chapter), add a listener to it (chapter 3.4), then round-trip, handling the global event by printing the globals available on this compositor. Try it for yourself:

```
$ cc -o globals -lwayland-client globals.c
```

Note: this chapter the last time we're going to show wire protocol dumps in hexadecimal, and probably the last time you'll ever see them in general. A better way to trace your Wayland client or server is to set the `WAYLAND_DEBUG` variable in your environment to `1` before running your program. Try it now with the "globals" program!

Registering globals

Registering globals with libwayland-server is done somewhat differently. When you generate "server-code" with wayland-scanner, it creates interfaces (which are analogous to listeners) and glue code for sending events. The first task is to register the global, with a function to rig up a *resource* 1 when the global is bound. In terms of code, the result looks something like this:

```
static void
wl_output_handle_bind(struct wl_client *client, void *data,
    uint32_t version, uint32_t id)
{
    struct my_state *state = data;
    // TODO
}

int
main(int argc, char *argv[])
{
    struct wl_display *display = wl_display_create();
    struct my_state state = { ... };
    // ...
    wl_global_create(wl_display, &wl_output_interface,
        1, &state, wl_output_handle_bind);
    // ...
}
```

If you take this code and, for example, patch it into the server example chapter 4.1, you'll make a `wl_output` global visible to the "globals" program we wrote last time2. However, any attempts to bind to this global will run into our TODO. To fill this in, we need to provide an *implementation* of the `wl_output` interface as well.

```
static void
wl_output_handle_resource_destroy(struct wl_resource *resource)
{
    struct my_output *client_output = wl_resource_get_user_data(resource);

    // TODO: Clean up resource

    remove_to_list(client_output->state->client_outputs, client_output);
}

static void
wl_output_handle_release(struct wl_client *client, struct wl_resource *resource)
{
    wl_resource_destroy(resource);
}

static const struct wl_output_interface
wl_output_implementation = {
    .release = wl_output_handle_release,
};

static void
wl_output_handle_bind(struct wl_client *client, void *data,
    uint32_t version, uint32_t id)
{
    struct my_state *state = data;

    struct my_output *client_output = calloc(1, sizeof(struct client_output));

    struct wl_resource *resource = wl_resource_create(
        client, &wl_output_interface, wl_output_interface.version, id);

    wl_resource_set_implementation(resource, &wl_output_implementation,
        client_output, wl_output_handle_resource_destroy);
    client_output->resource = resource;
    client_output->state = state;

    // TODO: Send geometry event, et al

    add_to_list(state->client_outputs, client_output);
}
```

This is a lot to take in, so let's explain it one piece at a time. At the bottom, we've extended our "bind" handler to create a `wl_resource` to track the server-side state for this object (using the ID that the client allocated). As we do this, we provide `wl_resource_create` with a pointer to our implementation of the interface — `wl_output_implementation`, a constant static struct in this file. The type (`struct wl_output_interface`) is generated by `wayland-scanner` and contains one function pointer for each request supported by this interface. We also take the opportunity to allocate a small container for storing any additional state we need that libwayland doesn't handle for us, the specific nature of which varies from protocol to protocol.

Note: there are two distinct things here which share the same name: `struct wl_output_interface` is an instance of an interface, where `wl_output_interface` is a global constant variable generated by `wayland-scanner` which contains metadata related to the implementation (such as version, used in the example above).

Our `wl_output_handle_release` function is called when the client sends the `release` request, indicating that they no longer need this resource — so we destroy it. This in turn triggers the `wl_output_handle_resource_destroy` function, which later we'll extend to free any of the state we allocated for it earlier. This function is also passed into `wl_resource_create` as the destructor, and will be called if the client terminates without explicitly sending the `release` request.

The other remaining "TODO" in our code is to send the "name" event, as well as a few others. If we review `wayland.xml`, we see this event on the interface:

```
<event name="geometry">
  <description summary="properties of the output">
The geometry event describes geometric properties of the output.
The event is sent when binding to the output object and whenever
any of the properties change.

The physical size can be set to zero if it doesn't make sense for this
output (e.g. for projectors or virtual outputs).
  </description>
  <arg name="x" type="int" />
  <arg name="y" type="int" />
  <arg name="physical_width" type="int" />
  <arg name="physical_height" type="int" />
  <arg name="subpixel" type="int" enum="subpixel" />
  <arg name="make" type="string" />
  <arg name="model" type="string" />
  <arg name="transform" type="int" enum="transform" />
</event>
```

It seems to be our responsibility to send this event when the output is bound. This is easy enough to add:

```
static void
wl_output_handle_bind(struct wl_client *client, void *data,
    uint32_t version, uint32_t id)
{
    struct my_state *state = data;

    struct my_output *client_output = calloc(1, sizeof(struct client_output));

    struct wl_resource *resource = wl_resource_create(
        client, &wl_output_implementation, wl_output_interface.version, id);

    wl_resource_set_implementation(resource, wl_output_implementation,
        client_output, wl_output_handle_resource_destroy);

    client_output->resource = resource;
    client_output->state = state;

    wl_output_send_geometry(resource, 0, 0, 1920, 1080,
        WL_OUTPUT_SUBPIXEL_UNKNOWN, "Foobar, Inc",
        "Fancy Monitor 9001 4K HD 120 FPS Noscope",
        WL_OUTPUT_TRANSFORM_NORMAL);

    add_to_list(state->client_outputs, client_output);
}
```

Note: `wl_output::geometry` is shown here for illustrative purposes, but in practice there are some special considerations for its use. Review the protocol XML before implementing this event in your client or server.

1

Resources represent the server-side state of each client's instance(s) of an object.

2

A slightly more sophisticated version of our "globals" program called `weston-info` is available from the Weston project, if you're interested in something more robust.

Buffers & surfaces

Apparently, the whole point of this system is to display information to users and receive their feedback for additional processing. In this chapter, we'll explore the first of these tasks: showing pixels on the screen.

There are two primitives which are used for this purpose: buffers and surfaces, governed respectively by the `wl_buffer` and `wl_surface` interfaces. Buffers act as an opaque container for some underlying pixel storage, and are supplied by clients with a number of methods — shared memory buffers and GPU handles being the most common.

Using wl_compositor

They say naming things is one of the most difficult problems in computer science, and here we are, with evidence in hand. The `wl_compositor` global is the Wayland compositor's, er, compositor. Through this interface, you may send the server your windows for presentation, to be *composited* with the other windows being shown alongside it. The compositor has two jobs: the creation of surfaces and regions.

To quote the spec, a Wayland *surface* has a rectangular area which may be displayed on zero or more outputs, present buffers, receive user input, and define a local coordinate system. We'll take all of these apart in detail later, but let's start with the basics: obtaining a surface and attaching a buffer to it. To obtain a surface, we first bind to the `wl_compositor` global. By extending the example from chapter 5.1 we get the following:

```
struct our_state {
    // ...
    struct wl_compositor *compositor;
    // ...
};

static void
registry_handle_global(void *data, struct wl_registry *wl_registry,
        uint32_t name, const char *interface, uint32_t version)
{
    struct our_state *state = data;
    if (strcmp(interface, wl_compositor_interface.name) == 0) {
        state->compositor = wl_registry_bind(
            wl_registry, name, &wl_compositor_interface, 4);
    }
}

int
main(int argc, char *argv[])
{
    struct our_state state = { 0 };
    // ...
    wl_registry_add_listener(registry, &registry_listener, &state);
    // ...
}
```

Note that we've specified version 4 when calling `wl_registry_bind`, which is the latest version at the time of writing. With this reference secured, we can create a `wl_surface`:

```
struct wl_surface *surface = wl_compositor_create_surface(state.compositor);
```

Before we can present it, we must first attach a source of pixels to it: a `wl_buffer`.

Shared memory buffers

The simplest means of getting pixels from client to compositor, and the only one enshrined in `wayland.xml`, is `wl_shm` — shared memory. Simply put, it allows you to transfer a file descriptor for the compositor to mmap with `MAP_SHARED`, then share pixel buffers out of this pool. Add some simple synchronization primitives to keep everyone from fighting over each buffer, and you have a workable — and portable — solution.

Binding to wl_shm

The registry global listener explained in chapter 5.1 will advertise the `wl_shm` global when it's available. Binding to it is fairly straightforward. Extending the example given in chapter 5.1, we get the following:

```
struct our_state {
    // ...
    struct wl_shm *shm;
    // ...
};

static void
registry_handle_global(void *data, struct wl_registry *registry,
        uint32_t name, const char *interface, uint32_t version)
{
    struct our_state *state = data;
    if (strcmp(interface, wl_shm_interface.name) == 0) {
        state->shm = wl_registry_bind(
            wl_registry, name, &wl_shm_interface, 1);
    }
}

int
main(int argc, char *argv[])
{
    struct our_state state = { 0 };
    // ...
    wl_registry_add_listener(registry, &registry_listener, &state);
    // ...
}
```

Once bound, we can optionally add a listener via `wl_shm_add_listener`. The compositor will advertise its supported pixel formats via this listener. The full list of possible pixel formats is given in `wayland.xml`. Two formats are required to be supported: `ARGB8888`, and `XRGB8888`, which are 24-bit color, with and without an alpha channel respectively.

Allocating a shared memory pool

A combination of POSIX `shm_open` and random file names can be utilized to create a file suitable for this purpose, and `ftruncate` can be utilized to bring it up to the appropriate size. The following boilerplate may be freely used under public domain or CC0:

```
#define _POSIX_C_SOURCE 200112L
#include <errno.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <time.h>
#include <unistd.h>

static void
randname(char *buf)
{
    struct timespec ts;
    clock_gettime(CLOCK_REALTIME, &ts);
    long r = ts.tv_nsec;
    for (int i = 0; i < 6; ++i) {
        buf[i] = 'A'+(r&15)+(r&16)*2;
        r >>= 5;
    }
}

static int
create_shm_file(void)
{
    int retries = 100;
    do {
        char name[] = "/wl_shm-XXXXXX";
        randname(name + sizeof(name) - 7);
        --retries;
        int fd = shm_open(name, O_RDWR | O_CREAT | O_EXCL, 0600);
        if (fd >= 0) {
            shm_unlink(name);
            return fd;
        }
    } while (retries > 0 && errno == EEXIST);
    return -1;
}

int
allocate_shm_file(size_t size)
{
    int fd = create_shm_file();
    if (fd < 0)
        return -1;
    int ret;
    do {
        ret = ftruncate(fd, size);
    } while (ret < 0 && errno == EINTR);
    if (ret < 0) {
        close(fd);
        return -1;
    }
    return fd;
}
```

Hopefully the code is fairly self-explanatory (famous last words). Armed with this, the client can create a shared memory pool fairly easily. Let's say, for example, that we want to show a 1920x1080 window. We'll need two buffers for double-buffering, so that'll be 4,147,200 pixels. Assuming the pixel format is `WL_SHM_FORMAT_XRGB8888`, that'll be 4 bytes to the pixel, for a total pool size of 16,588,800 bytes. Bind to the `wl_shm` global from the registry as explained in chapter 5.1, then use it like so to create an shm pool which can hold these buffers:

```
const int width = 1920, height = 1080;
const int stride = width * 4;
const int shm_pool_size = height * stride * 2;

int fd = allocate_shm_file(shm_pool_size);
uint8_t *pool_data = mmap(NULL, shm_pool_size,
    PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

struct wl_shm *shm = ...; // Bound from registry
struct wl_shm_pool *pool = wl_shm_create_pool(shm, fd, shm_pool_size);
```

Creating buffers from a pool

Once word of this gets to the compositor, it will `mmap` this file descriptor as well. Wayland is asynchronous, though, so we can start allocating buffers from this pool right away. Since we allocated space for two buffers, we can assign each an index and convert that index into a byte offset in the pool. Equipped with this information, we can create a `wl_buffer`:

```
int index = 0;
int offset = height * stride * index;
struct wl_buffer *buffer = wl_shm_pool_create_buffer(pool, offset,
    width, height, stride, WL_SHM_FORMAT_XRGB8888);
```

We can write an image to this buffer now as well. For example, to set it to solid white:

```
uint32_t *pixels = (uint32_t *)&pool_data[offset];
memset(pixels, 0, width * height * 4);
```

Or, for something more interesting, here's a checkerboard pattern:

```
uint32_t *pixels = (uint32_t *)&pool_data[offset];
for (int y = 0; y < height; ++y) {
  for (int x = 0; x < width; ++x) {
    if ((x + y / 8 * 8) % 16 < 8) {
      pixels[y * width + x] = 0xFF666666;
    } else {
      pixels[y * width + x] = 0xFFEEEEEE;
    }
  }
}
```

With the stage set, we'll attach our buffer to our surface, mark the whole surface as damaged[1], and commit it:

```
wl_surface_attach(surface, buffer, 0, 0);
wl_surface_damage(surface, 0, 0, UINT32_MAX, UINT32_MAX);
wl_surface_commit(surface);
```

If you were to apply all of this newfound knowledge to writing a Wayland client yourself, you may arrive at this point confused when your buffer is not shown on-screen. We're missing a critical final step — assigning your surface a role.

1

"Damaged" meaning "this area needs to be redrawn"

wl_shm on the server

Before we get there, however, the server-side part of this deserves note. libwayland provides some helpers to make using `wl_shm` easier. To configure it for your display, it only requires the following:

```
int
wl_display_init_shm(struct wl_display *display);

uint32_t *
wl_display_add_shm_format(struct wl_display *display, uint32_t format);
```

The former creates the global and rigs up the internal implementation, and the latter adds a supported pixel format (remember to at least add ARGB8888 and XRGB8888). Once a client attaches a buffer to one of its surfaces, you can pass the buffer resource into `wl_shm_buffer_get` to obtain a `wl_shm_buffer` reference, and utilize it like so:

```
void
wl_shm_buffer_begin_access(struct wl_shm_buffer *buffer);

void
wl_shm_buffer_end_access(struct wl_shm_buffer *buffer);

void *
wl_shm_buffer_get_data(struct wl_shm_buffer *buffer);

int32_t
wl_shm_buffer_get_stride(struct wl_shm_buffer *buffer);

uint32_t
wl_shm_buffer_get_format(struct wl_shm_buffer *buffer);

int32_t
wl_shm_buffer_get_width(struct wl_shm_buffer *buffer);

int32_t
wl_shm_buffer_get_height(struct wl_shm_buffer *buffer);
```

If you guard your accesses to the buffer data with `begin_access` and `end_access`, libwayland will take care of locking for you.

Linux dmabuf

Most Wayland compositors do their rendering on the GPU, and many Wayland clients do their rendering on the GPU as well. With the shared memory approach, sending buffers from the client to the compositor in such cases is very inefficient, as the client has to read their data from the GPU to the CPU, then the compositor has to read it from the CPU back to the GPU to be rendered.

The Linux DRM (Direct Rendering Manager) interface (which is also implemented on some BSDs) provides a means for us to export handles to GPU resources. Mesa, the predominant implementation of userspace Linux graphics drivers, implements a protocol that allows EGL users to transfer handles to their GPU buffers from the client to the compositor for rendering, without ever copying data to the CPU.

The internals of how this protocol works are out of scope for this book and would be more appropriate for resources which focus on Mesa or Linux DRM in particular. However, we can provide a short summary of its use.

1. Use `eglGetPlatformDisplayEXT` in concert with `EGL_PLATFORM_WAYLAND_KHR` to create an EGL display.
2. Configure the display normally, choosing a config appropriate to your circumstances with `EGL_SURFACE_TYPE` set to `EGL_WINDOW_BIT`.
3. Use `wl_egl_window_create` to create a `wl_egl_window` for a given `wl_surface`.

4. Use `eglCreatePlatformWindowSurfaceEXT` to create an `EGLSurface` for a `wl_egl_window`.
5. Proceed using EGL normally, e.g. `eglMakeCurrent` to make current the EGL context for your surface and `eglSwapBuffers` to send an up-to-date buffer to the compositor and commit the surface.

Should you need to change the size of the `wl_egl_window` later, use `wl_egl_window_resize`.

But I really want to know about the internals

Some Wayland programmers who don't use libwayland complain that this approach ties Mesa and libwayland tightly together, which is true. However, untangling them is not impossible — it just requires a lot of work for you in the form of implementing `linux-dmabuf` yourself. Consult the Wayland extension XML for details on the protocol, and Mesa's implementation at `src/egl/drivers/dri2/platform_wayland.c` (at the time of writing). Good luck and godspeed.

For the server

Unfortunately, the details for the compositor are both complicated and out-of-scope for this book. I can point you in the right direction, however: the wlroots implementation (found at `types/wlr_linux_dmabuf_v1.c` at the time of writing) is straightforward and should set you on the right path.

Surface roles

We have created a pixel buffer, sent it to the server, and attached it to a surface through which we can allegedly present it to the user. However, one crucial piece is missing to imbue the surface with meaning: its role.

There are a lot of different situations where a pixel buffer might be presented to the user, and each calls for different semantics. Some examples include application windows, sure, but others include a cursor image or your desktop wallpaper. To contrast the semantics of application windows with cursors, consider that your cursor cannot be minimized, and application windows should not be stuck to the mouse. For this reason, *roles* provide another layer of abstraction which allows you to assign the appropriate semantics to the surface.

The role you're probably dying to assign to it, 6 chapters in, is an application window. The following chapter introduces the mechanism by which this is achieved: XDG shell.

XDG shell basics

The XDG (cross-desktop group) shell is a standard protocol extension for Wayland which describes the semantics for application windows. It defines two `wl_surface` roles: "toplevel", for your top-level application windows, and "popup", for things like context menus, dropdown menus, tooltips, and so on - which are children of top-level windows. With these together, you can form a tree of surfaces, with a toplevel at the top and popups or additional toplevels at the leaves. The protocol also defines a *positioner* interface, which is used for help positioning popups with limited information about the things around your window.

xdg-shell, as a protocol *extension*, is not defined in `wayland.xml`. Instead you'll find it in the `wayland-protocols` package. It's probably installed at a path somewhat like `/usr/share/wayland-protocols/stable/xdg-shell/xdg-shell.xml` on your system.

xdg_wm_base

`xdg_wm_base` is the only global defined by the specification, and it provides requests for creating each of the other objects you need. The most basic implementation starts by handling the “ping” event — when the compositor sends it, you should respond with a “pong” request in a timely manner to hint that you haven’t become deadlocked. Another request deals with the creation of positioners, the objects mentioned earlier, and we’ll save the details on these for chapter 10. The request we want to look into first is `get_xdg_surface`.

XDG surfaces

Surfaces in the domain of xdg-shell are referred to as `xdg_surfaces`, and this interface brings with it a small amount of functionality common to both kinds of XDG surfaces — toplevels and popups. The semantics for each kind of XDG surface are different enough still that they must be specified explicitly through an additional role.

The `xdg_surface` interface provides additional requests for assigning the more specific roles of popup and toplevel. Once we’ve bound an object to the `xdg_wm_base` global, we can use the `get_xdg_surface` request to obtain one for a `wl_surface`.

```
<request name="get_xdg_surface">
  <arg name="id" type="new_id" interface="xdg_surface"/>
  <arg name="surface" type="object" interface="wl_surface"/>
</request>
```

The `xdg_surface` interface, in addition to requests for assigning a more specific role of toplevel or popup to your surface, also includes some important functionality common to both roles. Let’s review these before we move on to the toplevel and popup-specific semantics.

```
<event name="configure">
  <arg name="serial" type="uint" summary="serial of the configure event"/>
</event>

<request name="ack_configure">
  <arg name="serial" type="uint" summary="the serial from the configure event"/>
</request>
```

The most important API of xdg-surface is this pair: `configure` and `ack_configure`. You may recall that a goal of Wayland is to make every frame perfect. That means no frames are shown with a half-applied state change, and to accomplish this we have to synchronize these changes between the client and server. For XDG surfaces, this pair of messages is the mechanism which supports this.

We’re only covering the basics for now, so we’ll summarize the importance of these two events as such: as events from the server inform your configuration (or reconfiguration) of a surface, apply them to a pending state. When a `configure` event arrives, apply the pending changes, use `ack_configure` to acknowledge you’ve done so, and render and commit a new frame. We’ll show this in practice in the next chapter, and explain it in detail in chapter 8.1.

```
<request name="set_window_geometry">
  <arg name="x" type="int"/>
  <arg name="y" type="int"/>
  <arg name="width" type="int"/>
  <arg name="height" type="int"/>
</request>
```

The `set_window_geometry` request is used primarily for applications using client-side decorations, to distinguish the parts of their surface which are considered a part of the window, and the parts which are not. Most commonly, this is used to exclude client-side drop-shadows rendered behind the window from being considered a part of it. The compositor may apply this information to govern its own behaviors for arranging and interacting with the window.

Application windows

We have shaved many yaks to get here, but it's time: XDG toplevel is the interface which we will finally use to display an application window. The XDG toplevel interface has many requests and events for managing application windows, including dealing with minimized and maximized states, setting window titles, and so on. We'll be discussing each part of it in detail in future chapters, so let's just concern ourselves with the basics now.

Based on our knowledge from the last chapter, we know that we can obtain an `xdg_surface` from a `wl_surface`, but that it only constitutes the first step: bringing a surface into the fold of XDG shell. The next step is to turn that XDG surface into an XDG toplevel — a "top-level" application window, so named for its top-level position in the hierarchy of windows and popup menus we will eventually create with XDG shell. To create one of these, we can use the appropriate request from the `xdg_surface` interface:

```
<request name="get_toplevel">
  <arg name="id" type="new_id" interface="xdg_toplevel"/>
</request>
```

This new `xdg_toplevel` interface puts many requests and events at our disposal for managing the lifecycle of application windows. Chapter 10 explores these in depth, but I know you're itching to get something on-screen. If you follow these steps, handling the `configure` and `ack_configure` riggings for XDG surface discussed in the previous chapter, and attach and commit a `wl_buffer` to our `wl_surface`, an application window will appear and present your buffer's contents to the user. Example code which does just this is provided in the next chapter. It also leverages one additional XDG toplevel request which we haven't covered yet:

```
<request name="set_title">
  <arg name="title" type="string"/>
</request>
```

This should be fairly self-explanatory. There's a similar one that we don't use in the example code, but which may be appropriate for your application:

```
<request name="set_app_id">
  <arg name="app_id" type="string"/>
</request>
```

The title is often shown in window decorations, taskbars, etc, whereas the app ID is used to identify your application or group your windows together. You might utilize these by setting your window title to "Application windows — The Wayland Protocol — Firefox", and your app ID to "firefox".

In summary, the following steps will take you from zero to a window on-screen:

1. Bind to `wl_compositor` and use it to create a `wl_surface`.
2. Bind to `xdg_wm_base` and use it to create an `xdg_surface` with your `wl_surface`.
3. Create an `xdg_toplevel` from the `xdg_surface` with `xdg_surface.get_toplevel`.
4. Configure a listener for the `xdg_surface` and await the `configure` event.
5. Bind to the buffer allocation mechanism of your choosing (such as `wl_shm`) and allocate a shared buffer, then render your content to it.
6. Use `wl_surface.attach` to attach the `wl_buffer` to the `wl_surface`.
7. Use `xdg_surface.ack_configure`, passing it the serial from `configure`, acknowledging that you have prepared a suitable frame.
8. Send a `wl_surface.commit` request.

Turn the page to see these steps in action.

Extended example code

Using the sum of what we've learned so far, we can now write a Wayland client which displays something on the screen. The following code is a complete Wayland application which opens an XDG toplevel window and shows a 640x480 grid of squares on it. Compile it like so:

```
wayland-scanner private-code \
  < /usr/share/wayland-protocols/stable/xdg-shell/xdg-shell.xml \
  > xdg-shell-protocol.c
wayland-scanner client-header \
  < /usr/share/wayland-protocols/stable/xdg-shell/xdg-shell.xml \
  > xdg-shell-client-protocol.h
cc -o client client.c xdg-shell-protocol.c -lwayland-client -lrt
```

Then run `./client` to see it in action, or `WAYLAND_DEBUG=1 ./client` to include a bunch of useful debugging information. Tada! In future chapters we will be building upon this client, so stow this code away somewhere safe.

```
#define _POSIX_C_SOURCE 200112L
#include <errno.h>
#include <fcntl.h>
#include <limits.h>
#include <stdbool.h>
#include <string.h>
#include <sys/mman.h>
#include <time.h>
#include <unistd.h>
#include <wayland-client.h>
#include "xdg-shell-client-protocol.h"

/* Shared memory support code */
static void
randname(char *buf)
{
    struct timespec ts;
    clock_gettime(CLOCK_REALTIME, &ts);
    long r = ts.tv_nsec;
    for (int i = 0; i < 6; ++i) {
        buf[i] = 'A'+(r&15)+(r&16)*2;
        r >>= 5;
    }
}

static int
create_shm_file(void)
{
    int retries = 100;
    do {
        char name[] = "/wl_shm-XXXXXX";
        randname(name + sizeof(name) - 7);
        --retries;
        int fd = shm_open(name, O_RDWR | O_CREAT | O_EXCL, 0600);
        if (fd >= 0) {
            shm_unlink(name);
            return fd;
        }
    } while (retries > 0 && errno == EEXIST);
    return -1;
}

static int
allocate_shm_file(size_t size)
{
    int fd = create_shm_file();
    if (fd < 0)
        return -1;
    int ret;
    do {
        ret = ftruncate(fd, size);
    } while (ret < 0 && errno == EINTR);
    if (ret < 0) {
        close(fd);
        return -1;
    }
    return fd;
}

/* Wayland code */
struct client_state {
    /* Globals */
    struct wl_display *wl_display;
    struct wl_registry *wl_registry;
    struct wl_shm *wl_shm;
    struct wl_compositor *wl_compositor;
    struct xdg_wm_base *xdg_wm_base;
    /* Objects */
    struct wl_surface *wl_surface;
    struct xdg_surface *xdg_surface;
    struct xdg_toplevel *xdg_toplevel;
};

static void
wl_buffer_release(void *data, struct wl_buffer *wl_buffer)
{
    /* Sent by the compositor when it's no longer using this buffer */
    wl_buffer_destroy(wl_buffer);
}

static const struct wl_buffer_listener wl_buffer_listener = {
    .release = wl_buffer_release,
};

static struct wl_buffer *
draw_frame(struct client_state *state)
{
    const int width = 640, height = 480;
    int stride = width * 4;
    int size = stride * height;

    int fd = allocate_shm_file(size);
    if (fd == -1) {
        return NULL;
    }

    uint32_t *data = mmap(NULL, size,
            PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (data == MAP_FAILED) {
        close(fd);
        return NULL;
    }

    struct wl_shm_pool *pool = wl_shm_create_pool(state->wl_shm, fd, size);
    struct wl_buffer *buffer = wl_shm_pool_create_buffer(pool, 0,
            width, height, stride, WL_SHM_FORMAT_XRGB8888);
    wl_shm_pool_destroy(pool);
    close(fd);

    /* Draw checkerboxed background */
    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            if ((x + y / 8 * 8) % 16 < 8)
                data[y * width + x] = 0xFF666666;
            else
                data[y * width + x] = 0xFFEEEEEE;
        }
    }
```

```c
    munmap(data, size);
    wl_buffer_add_listener(buffer, &wl_buffer_listener, NULL);
    return buffer;
}

static void
xdg_surface_configure(void *data,
        struct xdg_surface *xdg_surface, uint32_t serial)
{
    struct client_state *state = data;
    xdg_surface_ack_configure(xdg_surface, serial);

    struct wl_buffer *buffer = draw_frame(state);
    wl_surface_attach(state->wl_surface, buffer, 0, 0);
    wl_surface_commit(state->wl_surface);
}

static const struct xdg_surface_listener xdg_surface_listener = {
    .configure = xdg_surface_configure,
};

static void
xdg_wm_base_ping(void *data, struct xdg_wm_base *xdg_wm_base, uint32_t serial)
{
    xdg_wm_base_pong(xdg_wm_base, serial);
}

static const struct xdg_wm_base_listener xdg_wm_base_listener = {
    .ping = xdg_wm_base_ping,
};

static void
registry_global(void *data, struct wl_registry *wl_registry,
        uint32_t name, const char *interface, uint32_t version)
{
    struct client_state *state = data;
    if (strcmp(interface, wl_shm_interface.name) == 0) {
        state->wl_shm = wl_registry_bind(
                wl_registry, name, &wl_shm_interface, 1);
    } else if (strcmp(interface, wl_compositor_interface.name) == 0) {
        state->wl_compositor = wl_registry_bind(
                wl_registry, name, &wl_compositor_interface, 4);
    } else if (strcmp(interface, xdg_wm_base_interface.name) == 0) {
        state->xdg_wm_base = wl_registry_bind(
                wl_registry, name, &xdg_wm_base_interface, 1);
        xdg_wm_base_add_listener(state->xdg_wm_base,
                &xdg_wm_base_listener, state);
    }
}

static void
```

```
registry_global_remove(void *data,
        struct wl_registry *wl_registry, uint32_t name)
{
    /* This space deliberately left blank */
}

static const struct wl_registry_listener wl_registry_listener = {
    .global = registry_global,
    .global_remove = registry_global_remove,
};

int
main(int argc, char *argv[])
{
    struct client_state state = { 0 };
    state.wl_display = wl_display_connect(NULL);
    state.wl_registry = wl_display_get_registry(state.wl_display);
    wl_registry_add_listener(state.wl_registry, &wl_registry_listener, &state);
    wl_display_roundtrip(state.wl_display);

    state.wl_surface = wl_compositor_create_surface(state.wl_compositor);
    state.xdg_surface = xdg_wm_base_get_xdg_surface(
            state.xdg_wm_base, state.wl_surface);
    xdg_surface_add_listener(state.xdg_surface, &xdg_surface_listener, &state);
    state.xdg_toplevel = xdg_surface_get_toplevel(state.xdg_surface);
    xdg_toplevel_set_title(state.xdg_toplevel, "Example client");
    wl_surface_commit(state.wl_surface);

    while (wl_display_dispatch(state.wl_display)) {
        /* This space deliberately left blank */
    }

    return 0;
}
```

Surfaces in depth

The basic areas of the surface interface that we've shown until now are sufficient to present data to the user, but the surface interface offers many additional requests and events for more efficient use. Many — if not most — applications do not need to redraw the entire surface each frame. Even deciding *when* to draw the next frame is best done with the assistance of the compositor. In this chapter, we'll explore the features of `wl_surface` in depth.

Surface lifecycle

We mentioned earlier that Wayland is designed to update everything atomically, such that no frame is ever presented in an invalid or intermediate state. Of the many attributes that can be configured for an application window and other surfaces, the driving mechanism behind that atomicity is the `wl_surface` itself.

Every surface has a *pending* state and an *applied* state, and no state at all when it's first created. The *pending* state is negotiated over the course of any number of requests from clients and events from the server, and when both sides agree that it represents a consistent surface state, the surface is *committed* — and the pending state is *applied* to the current state of the surface. Until this time, the compositor will continue to render the last consistent state; once committed, will use the new state from the next frame forward.

Among the state which is updated atomically are:

- The attached `wl_buffer`, or the pixels making up the content of the surface
- The region which was "damaged" since the last frame, and needs to be redrawn
- The region which accepts input events
- The region considered opaque1
- Transformations on the attached `wl_buffer`, to rotate or present a subset of the buffer
- The scale factor of the buffer, used for HiDPI displays

In addition to these features of the surface, the surface's *role* may have additional double-buffered state like this. All of this state, along with any state associated with the role, is applied when `wl_surface.commit` is sent. You can send these requests several times if you change your mind, and only the most recent value for any of these properties will be considered when the surface is eventually committed.

When you first create your surface, the initial state is invalid. To make it valid (or to *map* the surface), you need to provide the necessary information to build the first consistent state for that surface. This includes giving it a role (like `xdg_toplevel`), allocating and attaching a buffer, and configuring any role-specific state for that surface. When you send a `wl_surface.commit` with this state correctly configured, the surface becomes valid (or *mapped*) and will be presented by the compositor.

The next question is: when should I prepare a new frame?

1

This is used by the compositor for optimization purposes.

Frame callbacks

The simplest way to update your surface is to simply render and attach new frames when it needs to change. This approach works well, for example, with event-driven applications. The user presses a key and the textbox needs to be re-rendered, so you can just re-render it immediately, damage the appropriate area, and attach a new buffer to be presented on the next frame.

However, some applications may want to render frames continuously. You might be rendering frames of a video game, playing back a video, or rendering an animation. Your display has an inherent *refresh rate*, or the fastest rate at which it's able to display updates (generally this is a number like 60 Hz, 144 Hz, etc). It doesn't make sense to render frames any faster than this, and doing so would be a waste of resources — CPU, GPU, even the user's battery. If you send several frames between each display refresh, all but the last will be discarded and have been rendered for naught.

Additionally, the compositor might not even want to show new frames for you. Your application might be off-screen, minimized, or hidden behind other windows; or only a small thumbnail of your application is being shown, so they might want to render you at a slower framerate to conserve resources. For this reason, the best way to continuously render frames in a Wayland client is to let the compositor tell you when it's ready for a new frame: using *frame callbacks*.

```
<interface name="wl_surface" version="4">
  <!-- ... -->

  <request name="frame">
    <arg name="callback" type="new_id" interface="wl_callback" />
  </request>

  <!-- ... -->
</interface>
```

This request will allocate a `wl_callback` object, which has a pretty simple interface:

```
<interface name="wl_callback" version="1">
  <event name="done">
    <arg name="callback_data" type="uint" />
  </event>
</interface>
```

When you request a frame callback on a surface, the compositor will send a `done` event to the callback object once it's ready for a new frame for this surface. In the case of `frame` events, the `callback_data` is set to the current time in millisecond, from an unspecified epoch. You can compare this with your last frame to calculate the progress of an animation or to scale input events.[1]

With frame callbacks in our toolbelt, why don't we update our application from chapter 7.3 so it scrolls a bit each frame? Let's start by adding a little bit of state to our `client_state` struct:

```
--- a/client.c
+++ b/client.c
@@ -71,6 +71,8 @@ struct client_state {
    struct xdg_surface *xdg_surface;
    struct xdg_toplevel *xdg_toplevel;
+   /* State */
+   float offset;
+   uint32_t last_frame;
 };

 static void wl_buffer_release(void *data, struct wl_buffer *wl_buffer) {
```

Then we'll update our `draw_frame` function to take the offset into account:

```
@@ -107,9 +109,10 @@ draw_frame(struct client_state *state)
    close(fd);

    /* Draw checkerboxed background */
+   int offset = (int)state->offset % 8;
    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
-           if ((x + y / 8 * 8) % 16 < 8)
+           if (((x + offset) + (y + offset) / 8 * 8) % 16 < 8)
                data[y * width + x] = 0xFF666666;
            else
                data[y * width + x] = 0xFFEEEEEE;
```

In the `main` function, let's register a callback for our first new frame:

```
@@ -195,6 +230,9 @@ main(int argc, char *argv[])
     xdg_toplevel_set_title(state.xdg_toplevel, "Example client");
     wl_surface_commit(state.wl_surface);

+    struct wl_callback *cb = wl_surface_frame(state.wl_surface);
+    wl_callback_add_listener(cb, &wl_surface_frame_listener, &state);
+
     while (wl_display_dispatch(state.wl_display)) {
         /* This space deliberately left blank */
     }
```

Then implement it like so:

```
@@ -147,6 +150,38 @@ static const struct xdg_wm_base_listener xdg_wm_base_listener =
     .ping = xdg_wm_base_ping,
 };

+static const struct wl_callback_listener wl_surface_frame_listener;
+
+static void
+wl_surface_frame_done(void *data, struct wl_callback *cb, uint32_t time)
+{
+    /* Destroy this callback */
+    wl_callback_destroy(cb);
+
+    /* Request another frame */
+    struct client_state *state = data;
+    cb = wl_surface_frame(state->wl_surface);
+    wl_callback_add_listener(cb, &wl_surface_frame_listener, state);
+
+    /* Update scroll amount at 24 pixels per second */
+    if (state->last_frame != 0) {
+        int elapsed = time - state->last_frame;
+        state->offset += elapsed / 1000.0 * 24;
+    }
+
+    /* Submit a frame for this event */
+    struct wl_buffer *buffer = draw_frame(state);
+    wl_surface_attach(state->wl_surface, buffer, 0, 0);
+    wl_surface_damage_buffer(state->wl_surface, 0, 0, INT32_MAX, INT32_MAX);
+    wl_surface_commit(state->wl_surface);
+
+    state->last_frame = time;
+}
+
+static const struct wl_callback_listener wl_surface_frame_listener = {
+    .done = wl_surface_frame_done,
+};
+
 static void
 registry_global(void *data, struct wl_registry *wl_registry,
        uint32_t name, const char *interface, uint32_t version)
```

Now, with each frame, we'll

1. Destroy the now-used frame callback.
2. Request a new callback for the next frame.
3. Render and submit the new frame.

The third step, broken down, is:

1. Update our state with a new offset, using the time since the last frame to scroll at a consistent rate.
2. Prepare a new `wl_buffer` and render a frame for it.
3. Attach the new `wl_buffer` to our surface.
4. Damage the entire surface.
5. Commit the surface.

Steps 3 and 4 update the *pending* state for the surface, giving it a new buffer and indicating the entire surface has changed. Step 5 commits this pending state, applying it to the surface's current state, and using it on the following frame. Applying this new buffer atomically means that we never show half of the last frame, resulting in a nice tear-free experience. Compile and run the updated client to try it out for yourself!

1

Want something more accurate? In chapter 12.1 we talk about a protocol extension which tells you with nanosecond resolution exactly when each frame was presented to the user.

Damaging surfaces

You may have noticed in the last example that we added this line of code when we committed a new frame for the surface:

```
wl_surface_damage_buffer(state->wl_surface, 0, 0, INT32_MAX, INT32_MAX);
```

If so, sharp eye! This code *damages* our surface, indicating to the compositor that it needs to be redrawn. Here we damage the entire surface (and well beyond it), but we could instead only damage part of it.

Let's say, for example, that you've written a GUI toolkit and the user is typing into a textbox. That textbox probably only takes up a small part of the window, and each new character takes up a smaller part still. When the user presses a key, you could render just the new character appended to the text they're writing, then damage only that part of the surface. The compositor can then copy just a fraction of your surface, which can speed things up considerably - especially for embedded devices. As you blink the caret between characters, you'll want to submit damage for its updates, and when the user changes views, you'll likely damage the entire surface. This way, everyone does less work, and the user will thank you for their improved battery life.

Note: The Wayland protocol provides two requests for damaging surfaces: `damage` and `damage_buffer`. The former is effectively deprecated, and you should only use the latter. The difference between them is that `damage` takes into account all of the transforms affecting the surface, such as rotations, scale factor, and buffer position and clipping. The latter instead applies damage relative to the buffer, which is generally easier to reason about.

Surface regions

We've already used the `wl_compositor` interface to create `wl_surfaces` via `wl_compositor.create_surface`. Note, however, that it has a second request: `create_region`.

```
<interface name="wl_compositor" version="4">
  <request name="create_surface">
    <arg name="id" type="new_id" interface="wl_surface" />
  </request>

  <request name="create_region">
    <arg name="id" type="new_id" interface="wl_region" />
  </request>
</interface>
```

The `wl_region` interface defines a group of rectangles, which collectively make up an arbitrarily shaped region of geometry. Its requests allow you to do bitwise operations against the geometry it defines by adding or subtracting rectangles from it.

```
<interface name="wl_region" version="1">
  <request name="destroy" type="destructor" />

  <request name="add">
    <arg name="x" type="int" />
    <arg name="y" type="int" />
    <arg name="width" type="int" />
    <arg name="height" type="int" />
  </request>

  <request name="subtract">
    <arg name="x" type="int" />
    <arg name="y" type="int" />
    <arg name="width" type="int" />
    <arg name="height" type="int" />
  </request>
</interface>
```

To make, for example, a rectangle with a hole in it, you could:

1. Send `wl_compositor.create_region` to allocate a `wl_region` object.
2. Send `wl_region.add(0, 0, 512, 512)` to create a 512x512 rectangle.
3. Send `wl_region.subtract(128, 128, 256, 256)` to remove a 256x256 rectangle from the middle of the region.

These areas can be disjoint as well; it needn't be a single continuous polygon. Once you've created one of these regions, you can pass it into the `wl_surface` interface, namely with the `set_opaque_region` and `set_input_region` requests.

```
<interface name="wl_surface" version="4">
  <request name="set_opaque_region">
    <arg name="region" type="object" interface="wl_region" allow-null="true" />
  </request>

  <request name="set_input_region">
```

```
    <arg name="region" type="object" interface="wl_region" allow-null="true" />
  </request>
</interface>
```

The opaque region is a hint to the compositor as to which parts of your surface are considered opaque. Based on this information, they can optimize their rendering process. For example, if your surface is completely opaque and occludes another window beneath it, then the compositor won't waste any time on redrawing the window beneath yours. By default, this is empty, which assumes that any part of your surface might be transparent. This makes the default case the least efficient but the most correct.

The input region indicates which parts of your surface accept pointer and touch input events. You might, for example, draw a drop-shadow underneath your surface, but input events which happen in this region should be passed to the client beneath you. Or, if your window is an unusual shape, you could create an input region in that shape. For most surface types by default, your entire surface accepts input.

Both of these requests can be used to set an empty region by passing in null instead of a `wl_region` object. They're also both double-buffered — so send a `wl_surface.commit` to make your changes effective. You can destroy the `wl_region` object to free up its resources as soon as you've sent the `set_opaque_region` or `set_input_region` requests with it. Updating the region after you send these requests will not update the state of the surface.

Subsurfaces

There's only one[1] surface role defined in the core Wayland protocol, `wayland.xml`: subsurfaces. They have an X, Y position relative to the parent surface — which needn't be constrained by the bounds of their parent surface — and a Z-order relative to their siblings and parent surface.

Some use-cases for this feature include playing a video surface in its native pixel format with an RGBA user-interface or subtitles shown on top, using an OpenGL surface for your primary application interface and using subsurfaces to render window decorations in software, or moving around parts of the UI without having to redraw on the client. With the assistance of hardware planes, the compositor, too, might not even have to redraw anything for updating your subsurfaces. On embedded systems in particular, this can be especially useful when it fits your use-case. A cleverly designed application can take advantage of subsurfaces to be very efficient.

The interface for managing these is the `wl_subcompositor` interface. The `get_subsurface` request is the main entry-point to the subcompositor:

```
<request name="get_subsurface">
  <arg name="id" type="new_id" interface="wl_subsurface" />
  <arg name="surface" type="object" interface="wl_surface" />
  <arg name="parent" type="object" interface="wl_surface" />
</request>
```

Once you have a `wl_subsurface` object associated with a `wl_surface`, it becomes a child of that surface. Subsurfaces can themselves have subsurfaces, resulting in an ordered tree of surfaces beneath any top-level surface. Manipulating these children is done through the `wl_subsurface` interface:

```
<request name="set_position">
  <arg name="x" type="int" summary="x coordinate in the parent surface"/>
  <arg name="y" type="int" summary="y coordinate in the parent surface"/>
```

```
</request>

<request name="place_above">
  <arg name="sibling" type="object" interface="wl_surface" />
</request>

<request name="place_below">
  <arg name="sibling" type="object" interface="wl_surface" />
</request>

<request name="set_sync" />
<request name="set_desync" />
```

A subsurface's z-order may be changed by placing it above or below any sibling surface that shares the same parent, or the parent surface itself.

The synchronization of the various properties of a `wl_subsurface` requires some explanation. These position and z-order properties are synchronized with the parent surface's lifecycle. When a `wl_surface.commit` request is sent for the main surface, all of its subsurfaces have changes to their position and z-order applied with it.

However, the `wl_surface` state associated with this subsurface, such as the attachment of buffers and accumulation of damage, need not be linked to the parent surface's lifecycle. This is the purpose of the `set_sync` and `set_desync` requests. Subsurfaces synced with their parent surface will commit all of their state when the parent surface is committed. Desynced surfaces will manage their own commit lifecycle like any other.

In short, the sync and desync requests are non-buffered and apply immediately. The position and z-order requests are buffered, and are not affected by the sync/desync property of the surface — they are always committed with the parent surface. The remaining surface state, on the associated `wl_surface`, is committed in accordance with the sync/desync status of the subsurface.

1

Disregarding the deprecated `wl_shell` interface.

High density surfaces (HiDPI)

In the past several years, a huge leap in pixel density in high-end displays has been seen, new displays packing twice as many pixels into the same physical area as we've seen in years past. We call these displays "HiDPI", short for "high dots per inch". However, these displays are so far ahead of their "LoDPI" peers that application-level changes are necessary to utilize them properly. By doubling the screen resolution in the same space, we would halve the size of all of our user interfaces if we lent them no special consideration. For most displays, this would make the text unreadable and the interactive elements uncomfortably small.

In exchange, however, we're offered a much greater amount of graphical fidelity with our vector graphics, most notably with respect to text rendering. Wayland addresses this by adding a "scale factor" to each output, and clients are expected to apply this scale factor to their interfaces. Additionally, clients which are unaware of HiDPI signal this limitation through inaction, allowing the compositor to make up for it by scaling up their buffers. The compositor signals the scale factor for each output via the appropriate event:

```
<interface name="wl_output" version="3">
  <!-- ... -->
  <event name="scale" since="2">
    <arg name="factor" type="int" />
  </event>
</interface>
```

Note that this was added in version 2, so when binding to the `wl_output` global you must set the version to at least 2 to receive these events. This is *not* enough to decide to use HiDPI in your clients, however. In order to make that call, the compositor must also send `enter` events for your `wl_surface` to indicate that it has "entered" (is being shown on) a particular output or outputs:

```
<interface name="wl_surface" version="4">
  <!-- ... -->
  <event name="enter">
    <arg name="output" type="object" interface="wl_output" />
  </event>
</interface>
```

Once you know the collection of outputs a client is shown on, it should take the maximum value of the scale factors, multiply the size (in pixels) of its buffers by this value, then render the UI at 2x or 3x (or Nx) scale. Then, indicate the scale the buffer was prepared at like so:

```
<interface name="wl_surface" version="4">
  <!-- ... -->
  <request name="set_buffer_scale" since="3">
    <arg name="scale" type="int" />
  </request>
</interface>
```

Note: this requires version 3 or newer of `wl_surface`. This is the version number you should pass to the `wl_registry` when you bind to `wl_compositor`.

Upon the next `wl_surface.commit`, your surface will assume this scale factor. If it's greater than the scale factor of an output the surface is shown on, the compositor will scale it down. If it's less than the scale factor of an output, the compositor will scale it up.

Seats: Handling input

Displaying your application to the user is only half of the I/O equation — most applications also need to process input. For this purpose, the seat provides an abstraction over input events on Wayland. In philosophical terms, a Wayland seat refers to one "seat" at which a user sits and operates the computer, and is associated with up to one keyboard and up to one "pointer" device (i.e. a mouse or touchpad). A similar relationship is defined for touchscreens, drawing tablet devices, and so on.

It's important to remember that this is an *abstraction*, and that the seats represented on a Wayland display may not correspond 1:1 with reality. In practice, it's rare for more than a single seat to be available on a Wayland session. If you plug a second keyboard into your computer, it's generally assigned to the same seat as the first, and the keyboard layout and so on are dynamically switched as you start typing on each. These implementation details are left to the Wayland compositor to consider.

From the client's perspective, it's reasonably straightforward. If you bind to the `wl_seat` global, you get access to the following interface:

```
<interface name="wl_seat" version="7">
  <enum name="capability" bitfield="true">
    <entry name="pointer" value="1" />
    <entry name="keyboard" value="2" />
    <entry name="touch" value="4" />
  </enum>

  <event name="capabilities">
    <arg name="capabilities" type="uint" enum="capability" />
  </event>

  <event name="name" since="2">
    <arg name="name" type="string" />
  </event>

  <request name="get_pointer">
    <arg name="id" type="new_id" interface="wl_pointer" />
  </request>

  <request name="get_keyboard">
    <arg name="id" type="new_id" interface="wl_keyboard" />
  </request>

  <request name="get_touch">
    <arg name="id" type="new_id" interface="wl_touch" />
  </request>

  <request name="release" type="destructor" since="5" />
</interface>
```

Note: This interface has been updated many times — take note of the version when you bind to the global. This book assumes you're binding to the latest version, which is version 7 at the time of writing.

This interface is relatively straightforward. The server will send the client a `capabilities` event to signal what kinds of input devices are supported by this seat — represented by a bitfield of `capability` values — and the client can bind to the input devices it wishes to use accordingly. For example, if the server sends `capabilities` where

`(caps & WL_SEAT_CAPABILITY_KEYBOARD) > 0` is true, the client may then use the `get_keyboard` request to obtain a `wl_keyboard` object for this seat. The semantics for each particular input device are covered in the remaining chapters.

Before we get to those, let's cover some common semantics.

Event serials

Some actions that a Wayland client may perform require a trivial form of authentication in the form of input event serials. For example, a client which opens a popup (a context menu summoned with a right click is one kind of popup) may want to "grab" all input events server-side from the affected seat until the popup is dismissed. To prevent abuse of this feature, the server can assign serials to each input event it sends, and require the client to include one of these serials in the request.

When the server receives such a request, it looks up the input event associated with the given serial and makes a judgement call. If the event was too long ago, or for the wrong surface, or wasn't the right kind of event — for example, it could reject grabs when you wiggle the mouse, but allow them when you click — it can reject the request.

From the server's perspective, they can simply send a incrementing integer with each input event, and record the serials which are considered valid for a particular use-case for later validation. The client receives these serials from their input event handlers, and can simply pass them back right away to perform the desired action.

We'll discuss these in more detail in later chapters, when we start covering the specific requests which require input event serials to validate them.

Input frames

A single input event from an input device may be broken up into several Wayland events for practical reasons. For example, a `wl_pointer` will emit an `axis` event as you use the scroll wheel, but it will separately emit an event telling you what *kind* of axis it was: scroll wheel, a finger on a touchpad, tilting the scroll wheel to the side, etc. The same input event from the input source may have also included some motion of the mouse, or the click of a button, if the user did all of these things quickly enough.

The semantic grouping of these related events differs slightly from input type to input type, but the `frame` event is generally common between them. In short, if you buffer up all of the input events you receive from a device, then wait for the `frame` event to signal that you've received all events for a single input "frame", you can interpret the buffered up *Wayland* events as a single *input* event, then reset the buffer and start collecting events for the next frame.

If this sounds too complicated, don't sweat it. Many applications don't have to worry about input frames. It's only when you start doing more complex input event handling that you'll want to concern yourself with this.

Releasing devices

When you're done using a device, each interface has a `release` request you can use to clean it up. It'll look something like this:

```
<request name="release" type="destructor" />
```

Easy enough.

Pointer input

Using the `wl_seat.get_pointer` request, clients may obtain a `wl_pointer` object. The server will send events to it whenever the user moves their pointer, presses mouse buttons, uses the scroll wheel, etc — whenever the pointer is over one of your surfaces. We can determine if this condition is met with the `wl_pointer.enter` event:

```
<event name="enter">
  <arg name="serial" type="uint" />
  <arg name="surface" type="object" interface="wl_surface" />
  <arg name="surface_x" type="fixed" />
  <arg name="surface_y" type="fixed" />
</event>
```

The server sends this event when the pointer moves over one of our surfaces, and specifies both the surface that was "entered", as well as the surface-local coordinates (from the top-left corner) that the pointer is positioned over. Coordinates here are specified with the "fixed" type, which you may remember from chapter 2.1 represents a 24.8-bit fixed-precision number (`wl_fixed_to_double` will convert this to C's `double` type).

When the pointer is moved away from your surface, the corresponding event is more brief:

```
<event name="leave">
  <arg name="serial" type="uint" />
  <arg name="surface" type="object" interface="wl_surface" />
</event>
```

Once a pointer has entered your surface, you'll start receiving additional events for it, which we'll discuss shortly. The first thing you will likely want to do, however, is provide a cursor image. The process is as such:

1. Create a new `wl_surface` with the `wl_compositor`.
2. Use `wl_pointer.set_cursor` to attach that surface to the pointer.
3. Attach a cursor image `wl_buffer` to the surface and commit it.

The only new API introduced here is `wl_pointer.set_cursor`:

```
<request name="set_cursor">
  <arg name="serial" type="uint" />
  <arg name="surface" type="object" interface="wl_surface" allow-null="true" />
  <arg name="hotspot_x" type="int" />
  <arg name="hotspot_y" type="int" />
</request>
```

The `serial` here has to come from the `enter` event. The `hotspot_x` and `hotspot_y` arguments specify the cursor-surface-local coordinates of the "hotspot", or the effective position of the pointer within the cursor image (e.g. at the tip of an arrow). Note also that the surface can be null — use this to hide the cursor entirely.

If you're looking for a good source of cursor images, libwayland ships with a separate `wayland-cursor` library, which can load X cursor themes from disk and create `wl_buffers` for them. See `wayland-cursor.h` for details, or the updates to our example client in chapter 9.5.

Note: wayland-cursor includes code for dealing with animated cursors, which weren't even cool in 1998. If I were you, I wouldn't bother with that. No one has ever complained that my Wayland clients don't support animated cursors.

After the cursor has entered your surface and you have attached an appropriate cursor, you're ready to start processing input events. There are motion, button, and axis events.

Pointer frames

A single frame of input processing on the server could carry information about lots of changes — for example, polling the mouse once could return, in a single packet, an updated position and the release of a button. The server sends these changes as separate *Wayland* events, and uses the "frame" event to group them together.

```
<event name="frame"></event>
```

Clients should accumulate all `wl_pointer` events as they're received, then process pending inputs as a single pointer event once the "frame" event is received.

Motion events

Motion events are specified in the same coordinate space as the `enter` event uses, and are straightforward enough:

```
<event name="motion">
  <arg name="time" type="uint" />
  <arg name="surface_x" type="fixed" />
  <arg name="surface_y" type="fixed" />
</event>
```

Like all input events which include a timestamp, the `time` value is a monotonically increasing millisecond-precision timestamp associated with this input event.

Button events

Button events are mostly self-explanatory:

```
<enum name="button_state">
  <entry name="released" value="0" />
  <entry name="pressed" value="1" />
</enum>

<event name="button">
  <arg name="serial" type="uint" />
  <arg name="time" type="uint" />
  <arg name="button" type="uint" />
  <arg name="state" type="uint" enum="button_state" />
</event>
```

However, the `button` argument merits some additional explanation. This number is a platform-specific input event, though note that FreeBSD reuses the Linux values. You can find these values for Linux in `linux/input-event-codes.h`, and the most useful ones will probably be represented by the constants `BTN_LEFT`, `BTN_RIGHT`, and `BTN_MIDDLE`. There are more, I'll leave you to peruse the header at your leisure.

Axis events

The axis event is used for scrolling actions, such as rotating your scroll wheel or rocking it from left to right. The most basic form looks like this:

```
<enum name="axis">
  <entry name="vertical_scroll" value="0" />
  <entry name="horizontal_scroll" value="1" />
</enum>

<event name="axis">
  <arg name="time" type="uint" />
  <arg name="axis" type="uint" enum="axis" />
  <arg name="value" type="fixed" />
</event>
```

However, axis events are complex, and this is the part of the `wl_pointer` interface which has received the most attention over the years. Several additional events exist which increase the specificity of the axis event:

```
<enum name="axis_source">
  <entry name="wheel" value="0" />
  <entry name="finger" value="1" />
  <entry name="continuous" value="2" />
  <entry name="wheel_tilt" value="3" />
</enum>

<event name="axis_source" since="5">
  <arg name="axis_source" type="uint" enum="axis_source" />
</event>
```

The axis_source event tells you what kind of axis was actuated — a scroll wheel, or a finger on a touchpad, tilting a rocker to the side, or something more novel. This event is simple, but the remainder are less so:

```
<event name="axis_stop" since="5">
  <arg name="time" type="uint" />
  <arg name="axis" type="uint" enum="axis" />
</event>

<event name="axis_discrete" since="5">
  <arg name="axis" type="uint" enum="axis" />
  <arg name="discrete" type="int" />
</event>
```

The precise semantics of these two events are complex, and if you wish to leverage them I recommend a careful reading of the summaries in `wayland.xml`. In short, the `axis_discrete` event is used to disambiguate axis events on an arbitrary scale from discrete steps of, for example, a scroll wheel where each "click" of the wheel represents a single discrete change in the axis value. The `axis_stop` event signals that a discrete user motion has completed, and is used when accounting for a scrolling event which takes place over several frames. Any future events should be interpreted as a separate motion.

XKB, briefly

The next input device on our list is keyboards, but we need to stop and give you some additional context before we discuss them. Keymaps are an essential detail involved in keyboard input, and XKB is the recommended way of handling them on Wayland.

When you press a key on your keyboard, it sends a *scancode* to the computer, which is simply a number assigned to that physical key. On my keyboard, scancode 1 is the Escape key, the '1' key is scancode 2, 'a' is 30, Shift is 42, and so on. I use a US ANSI keyboard layout, but there are many other layouts, and their scancodes differ. On my friend's German keyboard, scancode 12 produces 'ß', while mine produces '-'.

To solve this problem, we use a library called "xkbcommon", which is named for its role as the common code from XKB (X KeyBoard) extracted into a standalone library. XKB defines a huge number of key *symbols*, such as XKB_KEY_A, and XKB_KEY_ssharp (ß, from German), and XKB_KEY_kana_WO (, from Japanese).

Identifying these keys and correlating them with key symbols like this is only part of the problem, however. 'a' can produce 'A' if the shift key is held down, '' is written as '' in Katakana mode, and while there is strictly speaking an uppercase version of 'ß', it's hardly ever used and certainly never typed. Keys like Shift are called *modifiers*, and groups like Hiragana and Katakana are called *groups*. Some modifiers can *latch*, like Caps

Lock. XKB has primitives for dealing with all of these cases, and maintains a state machine which tracks what your keyboard is doing and figures out exactly which *Unicode codepoints* the user is trying to type.

Using XKB

So how is xkbcommon actually used? Well, the first step is to link to it and grab the header, `xkbcommon/xkbcommon.h`.[1] Most programs which utilize xkbcommon will have to manage three objects:

- xkb_context: a handle used for configuring other XKB resources
- xkb_keymap: a mapping from scancodes to key symbols
- xkb_state: a state machine that turns key symbols into UTF-8 strings

The process for setting this up usually goes as follows:

1. Use `xkb_context_new` to create a new xkb_context, passing it `XKB_CONTEXT_NO_FLAGS` unless you're doing something weird.
2. Obtain a key map as a string.*
3. Use `xkb_keymap_new_from_string` to create an `xkb_keymap` for this key map. There's only one key map format, `XKB_KEYMAP_FORMAT_TEXT_V1`, which you'll pass for the format parameter. Again, unless you're doing something weird, you'll use `XKB_KEYMAP_COMPILE_NO_FLAGS` for the flags.
4. Use `xkb_state_new` to create an xkb_state with your keymap. The state will increment the refcount for the keymap, so use `xkb_keymap_unref` if you're done with it yourself.
5. Obtain scancodes from a keyboard.*
6. Feed the scancodes into `xkb_state_key_get_one_sym` to get keysyms, and into `xkb_state_key_get_utf8` to get UTF-8 strings. Tada!

* *These steps are discussed in the next section.*

In terms of code, the process looks like the following:

```c
#include <xkbcommon/xkbcommon.h> // -lxkbcommon
/* ... */

const char *keymap_str = /* ... */;

/* Create an XKB context */
struct xkb_context *context = xkb_context_new(XKB_CONTEXT_NO_FLAGS);

/* Use it to parse a keymap string */
struct xkb_keymap *keymap = xkb_keymap_new_from_string(
    xkb_context, keymap_str, XKB_KEYMAP_FORMAT_TEXT_V1,
    XKB_KEYMAP_COMPILE_NO_FLAGS);

/* Create an XKB state machine */
struct xkb_state *state = xkb_state_new(keymap);
```

Then, to process scancodes:

```c
int scancode = /* ... */;
```

```
xkb_keysym_t sym = xkb_state_key_get_one_sym(xkb_state, scancode);
if (sym == XKB_KEY_F1) {
    /* Do the thing you do when the user presses F1 */
}

char buf[128];
xkb_state_key_get_utf8(xkb_state, scancode, buf, sizeof(buf));
printf("UTF-8 input: %s\n", buf);
```

Equipped with these details, we're ready to tackle processing keyboard input.

1

xkbcommon ships with a pc file: use `pkgconf --cflags xkbcommon` and `pkgconf --libs xkbcommon`, or your build system's preferred way of consuming pc files.

Keyboard input

Equipped with an understanding of how to use XKB, let's extend our Wayland code to provide us with key events to feed into it. Similarly to how we obtained a `wl_pointer` resource, we can use the `wl_seat.get_keyboard` request to create a `wl_keyboard` for a seat whose capabilities include `WL_SEAT_CAPABILITY_KEYBOARD`. When you're done with it, you should send the "release" request:

```
<request name="release" type="destructor" since="3">
</request>
```

This will allow the server to clean up the resources associated with this keyboard.

But how do you actually use it? Let's start with the basics.

Key maps

When you bind to `wl_keyboard`, the first event that the server is likely to send is `keymap`.

```
<enum name="keymap_format">
  <entry name="no_keymap" value="0" />
  <entry name="xkb_v1" value="1" />
</enum>

<event name="keymap">
  <arg name="format" type="uint" enum="keymap_format" />
  <arg name="fd" type="fd" />
  <arg name="size" type="uint" />
</event>
```

The `keymap_format` enum is provided in the event that we come up with a new format for keymaps, but at the time of writing, XKB keymaps are the only format which the server is likely to send.

Bulk data like this is transferred over file descriptors. We could simply read from the file descriptor, but in general it's recommended to mmap it instead. In C, this could look similar to the following code:

```
#include <sys/mman.h>
// ...

static void wl_keyboard_keymap(void *data, struct wl_keyboard *wl_keyboard,
        uint32_t format, int32_t fd, uint32_t size) {
    assert(format == WL_KEYBOARD_KEYMAP_FORMAT_XKB_V1);
    struct my_state *state = (struct my_state *)data;

    char *map_shm = mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);
    assert(map_shm != MAP_FAILED);

    struct xkb_keymap *keymap = xkb_keymap_new_from_string(
        state->xkb_context, map_shm, XKB_KEYMAP_FORMAT_TEXT_V1,
        XKB_KEYMAP_COMPILE_NO_FLAGS);
    munmap(map_shm, size);
    close(fd);

    // ...do something with keymap...
}
```

Once we have a keymap, we can interpret future keypress events for this `wl_keyboard`. Note that the server can send a new keymap at any time, and all future key events should be interpreted in that light.

Keyboard focus

```
<event name="enter">
  <arg name="serial" type="uint" />
  <arg name="surface" type="object" interface="wl_surface" />
  <arg name="keys" type="array" />
</event>

<event name="leave">
  <arg name="serial" type="uint" />
  <arg name="surface" type="object" interface="wl_surface" />
</event>
```

Like `wl_pointer`'s "enter" and "leave" events are issued when a pointer is moved over your surface, the server sends `wl_keyboard.enter` when a surface receives keyboard focus, and `wl_keyboard.leave` when it's lost. Many applications will change their appearance under these conditions — for example, to start drawing a blinking caret.

The "enter" event also includes an array of currently pressed keys. This is an array of 32-bit unsigned integers, each representing the scancode of a pressed key.

Input events

Once the keyboard has entered your surface, you can expect to start receiving input events.

```
<enum name="key_state">
  <entry name="released" value="0" />
  <entry name="pressed" value="1" />
</enum>

<event name="key">
  <arg name="serial" type="uint" />
  <arg name="time" type="uint" />
  <arg name="key" type="uint" />
  <arg name="state" type="uint" enum="key_state" />
</event>

<event name="modifiers">
  <arg name="serial" type="uint" />
  <arg name="mods_depressed" type="uint" />
  <arg name="mods_latched" type="uint" />
  <arg name="mods_locked" type="uint" />
  <arg name="group" type="uint" />
</event>
```

The “key” event is sent when the user presses or releases a key. Like many input events, a serial is included which you can use to associate future requests with this input event. The “key” is the scancode of the key which was pressed or released, and the “state” is the pressed or released state of that key.

Important: the scancode from this event is the Linux evdev scancode. To translate this to an XKB scancode, you must add 8 to the evdev scancode.

The modifiers event includes a similar serial, as well as masks of the depressed, latched, and locked modifiers, and the index of the input group currently in use. A modifier is depressed, for example, while you hold down Shift. A modifier can latch, such as pressing Shift with sticky keys enabled - it’ll stop taking effect after the next non-modifier key is pressed. And a modifier can be locked, such as when caps lock is toggled on or off. Input groups are used to switch between various keyboard layouts, such as toggling between ISO and ANSI layouts, or for more language-specific features.

The interpretation of modifiers is keymap-specific. You should forward them both to XKB to deal with. Most implementations of the “modifiers” event are straightforward:

```
static void wl_keyboard_modifiers(void *data, struct wl_keyboard *wl_keyboard,
        uint32_t serial, uint32_t depressed, uint32_t latched,
        uint32_t locked, uint32_t group) {
    struct my_state *state = (struct my_state *)data;
    xkb_state_update_mask(state->xkb_state,
        depressed, latched, locked, 0, 0, group);
}
```

Key repeat

The last event to consider is the “repeat_info” event:

```
<event name="repeat_info" since="4">
  <arg name="rate" type="int" />
  <arg name="delay" type="int" />
</event>
```

In Wayland, the client is responsible for implementing “key repeat” — the feature which continues to type characters as long as you’ve got the key held doooooown. This event is sent to inform the client of the user’s preferences for key repeat settings. The “delay” is the number of milliseconds a key should be held down for before key repeat kicks in, and the “rate” is the number of characters per second to repeat until the key is released.

Touch input

On the surface, touchscreen input is fairly simple, and your implementation can be simple as well. However, the protocol offers you a lot of depth, which applications may take advantage of to provide more nuanced touch-driven gestures and feedback.

Most touch-screen devices support *multitouch*: they can track multiple locations where the screen has been touched. Each of these "touch points" is assigned an ID which is unique among all currently active points where the screen is being touched, but might be reused if you lift your finger and press again.[1]

Similarly to other input devices, you may obtain a `wl_touch` resource with `wl_seat.get_touch`, and you should send a "release" request when you're finished with it.

Touch frames

Like pointers, a single frame of touch processing on the server might carry information about many changes, but the server sends these as discrete Wayland events. The `wl_touch.frame` event is used to group these together.

```
<event name="frame"></event>
```

Clients should accumulate all `wl_touch` events as they're received, then process pending inputs as a single touch event when the "frame" event is received.

Touch and release

The first events we'll look at are "down" and "up", which are respectively raised when you press your finger against the device, and remove your finger from the device.

```
<event name="down">
  <arg name="serial" type="uint" />
  <arg name="time" type="uint" />
  <arg name="surface" type="object" interface="wl_surface" />
  <arg name="id" type="int" />
  <arg name="x" type="fixed" />
  <arg name="y" type="fixed" />
</event>

<event name="up">
  <arg name="serial" type="uint" />
  <arg name="time" type="uint" />
  <arg name="id" type="int" />
</event>
```

The "x" and "y" coordinates are fixed-point coordinates in the coordinate space of the surface which was touched — which is given in the "surface" argument. The time is a monotonically increasing timestamp with an arbitrary epoch, in milliseconds.[2] Note also the inclusion of a serial, which can be included in future requests to associate them with this input event.

Motion

After you receive a "down" event with a specific touch ID, you will begin to receive motion events which describe the movement of that touch point across the device.

```
<event name="motion">
  <arg name="time" type="uint" />
  <arg name="id" type="int" />
  <arg name="x" type="fixed" />
  <arg name="y" type="fixed" />
</event>
```

The "x" and "y" coordinates here are in the relative coordinate space of the surface which the "enter" event was sent for.

Gesture cancellation

Touch events often have to meet some threshold before they're recognized as a gesture. For example, swiping across the screen from left to right could be used by the Wayland compositor to switch between applications. However, it's not until some threshold has been crossed — say, reaching the midpoint of the screen in a certain amount of time — that the compositor recognizes this behavior as a gesture.

Until this threshold is reached, the compositor will be sending normal touch events for the surface that is being touched. Once the gesture is identified, the compositor will send a "cancel" event to let you know that the compositor is taking over.

```
<event name="cancel"></event>
```

When you receive this event, all active touch points are cancelled.

Shape and orientation

Some high-end touch hardware is capable of determining more information about the way the user is interacting with it. For users of suitable hardware and applications wishing to employ more advanced interactions or touch feedback, the "shape" and "orientation" events are provided.

```
<event name="shape" since="6">
  <arg name="id" type="int" />
  <arg name="major" type="fixed" />
  <arg name="minor" type="fixed" />
</event>

<event name="orientation" since="6">
  <arg name="id" type="int" />
  <arg name="orientation" type="fixed" />
</event>
```

The "shape" event defines an elliptical approximation of the shape of the object which is touching the screen, with a major and minor axis represented in units in the coordinate space of the touched surface. The orientation event rotates this ellipse by specifying the angle between the major axis and the Y-axis of the touched surface, in degrees.

Touch is the last of the input devices supported by the Wayland protocol. With this knowledge in hand, let's update our example code.

1

Emphasis on "might" — don't make any assumptions based on the repeated use of a touch point ID.

2

This means that separate timestamps can be compared to each other to obtain the time between events, but are not comparable to wall-clock time.

Expanding our example code

In previous chapters, we built a simple client which can present its surfaces on the display. Let's expand this code a bit to build a client which can receive input events. For the sake of simplicity, we're just going to be logging input events to stderr.

This is going to require a lot more code than we've worked with so far, so get strapped in. The first thing we need to do is set up the seat.

Setting up the seat

The first thing we'll need is a reference to a seat. We'll add it to our `client_state` struct, and add keyboard, pointer, and touch objects for later use as well:

```
        struct wl_shm *wl_shm;
        struct wl_compositor *wl_compositor;
        struct xdg_wm_base *xdg_wm_base;
+       struct wl_seat *wl_seat;
        /* Objects */
        struct wl_surface *wl_surface;
        struct xdg_surface *xdg_surface;
+       struct wl_keyboard *wl_keyboard;
+       struct wl_pointer *wl_pointer;
+       struct wl_touch *wl_touch;
        /* State */
        float offset;
        uint32_t last_frame;
        int width, height;
```

We'll also need to update `registry_global` to register a listener for that seat.

```
                                wl_registry, name, &xdg_wm_base_interface, 1);
                xdg_wm_base_add_listener(state->xdg_wm_base,
                                &xdg_wm_base_listener, state);
+       } else if (strcmp(interface, wl_seat_interface.name) == 0) {
+               state->wl_seat = wl_registry_bind(
+                               wl_registry, name, &wl_seat_interface, 7);
+               wl_seat_add_listener(state->wl_seat,
+                               &wl_seat_listener, state);
        }
 }
```

Note that we bind to the latest version of the seat interface, version 7. Let's also rig up that listener:

```
+static void
+wl_seat_capabilities(void *data, struct wl_seat *wl_seat, uint32_t capabilities)
+{
+       struct client_state *state = data;
+       /* TODO */
+}
+
+static void
+wl_seat_name(void *data, struct wl_seat *wl_seat, const char *name)
+{
+       fprintf(stderr, "seat name: %s\n", name);
+}
+
+static const struct wl_seat_listener wl_seat_listener = {
+       .capabilities = wl_seat_capabilities,
+       .name = wl_seat_name,
+};
```

If you compile (`cc -o client client.c xdg-shell-protocol.c`) and run this now, you should seat the name of the seat printed to stderr.

Rigging up pointer events

Let's get to pointer events. If you recall, pointer events from the Wayland server are to be accumulated into a single logical event. For this reason, we'll need to define a struct to store them in.

```
+enum pointer_event_mask {
+       POINTER_EVENT_ENTER = 1 << 0,
+       POINTER_EVENT_LEAVE = 1 << 1,
+       POINTER_EVENT_MOTION = 1 << 2,
+       POINTER_EVENT_BUTTON = 1 << 3,
+       POINTER_EVENT_AXIS = 1 << 4,
+       POINTER_EVENT_AXIS_SOURCE = 1 << 5,
+       POINTER_EVENT_AXIS_STOP = 1 << 6,
+       POINTER_EVENT_AXIS_DISCRETE = 1 << 7,
+};
+
+struct pointer_event {
+       uint32_t event_mask;
+       wl_fixed_t surface_x, surface_y;
+       uint32_t button, state;
+       uint32_t time;
+       uint32_t serial;
+       struct {
+               bool valid;
+               wl_fixed_t value;
+               int32_t discrete;
+       } axes[2];
+       uint32_t axis_source;
+};
```

We'll be using a bitmask here to identify which events we've received for a single pointer frame, and storing the relevant information from each event in their respective fields. Let's add this to our state struct as well:

```
        /* State */
        float offset;
        uint32_t last_frame;
        int width, height;
        bool closed;
+       struct pointer_event pointer_event;
 };
```

Then we'll need to update our `wl_seat_capabilities` to set up the pointer object for seats which are capable of pointer input.

```
 static void
 wl_seat_capabilities(void *data, struct wl_seat *wl_seat, uint32_t capabilities)
 {
         struct client_state *state = data;
-        /* TODO */
+
+        bool have_pointer = capabilities & WL_SEAT_CAPABILITY_POINTER;
+
+        if (have_pointer && state->wl_pointer == NULL) {
+                state->wl_pointer = wl_seat_get_pointer(state->wl_seat);
+                wl_pointer_add_listener(state->wl_pointer,
+                                &wl_pointer_listener, state);
+        } else if (!have_pointer && state->wl_pointer != NULL) {
+                wl_pointer_release(state->wl_pointer);
+                state->wl_pointer = NULL;
+        }
 }
```

This merits some explanation. Recall that `capabilities` is a bitmask of the kinds of devices supported by this seat — a bitwise AND (&) with a capability will produce a non-zero value if supported. Then, if we have a pointer and have *not* already configured it, we take the first branch, using `wl_seat_get_pointer` to obtain a pointer reference and storing it in our state. If the seat does *not* support pointers, but we already have one configured, we use `wl_pointer_release` to get rid of it. Remember that the capabilities of a seat can change at runtime, for example when the user un-plugs and re-plugs their mouse.

We also configured a listener for the pointer. Let's add the struct for that, too:

```
+static const struct wl_pointer_listener wl_pointer_listener = {
+        .enter = wl_pointer_enter,
+        .leave = wl_pointer_leave,
+        .motion = wl_pointer_motion,
+        .button = wl_pointer_button,
+        .axis = wl_pointer_axis,
+        .frame = wl_pointer_frame,
+        .axis_source = wl_pointer_axis_source,
+        .axis_stop = wl_pointer_axis_stop,
+        .axis_discrete = wl_pointer_axis_discrete,
+};
```

Pointers have a lot of events. Let's have a look at them.

```
+static void
+wl_pointer_enter(void *data, struct wl_pointer *wl_pointer,
+                uint32_t serial, struct wl_surface *surface,
+                wl_fixed_t surface_x, wl_fixed_t surface_y)
+{
+        struct client_state *client_state = data;
+        client_state->pointer_event.event_mask |= POINTER_EVENT_ENTER;
+        client_state->pointer_event.serial = serial;
+        client_state->pointer_event.surface_x = surface_x,
+                client_state->pointer_event.surface_y = surface_y;
+}
```

```
+
+static void
+wl_pointer_leave(void *data, struct wl_pointer *wl_pointer,
+               uint32_t serial, struct wl_surface *surface)
+{
+       struct client_state *client_state = data;
+       client_state->pointer_event.serial = serial;
+       client_state->pointer_event.event_mask |= POINTER_EVENT_LEAVE;
+}
```

The "enter" and "leave" events are fairly straightforward, and they set the stage for the rest of the implementation. We update the event mask to include the appropriate event, then populate it with the data we were provided. The "motion" and "button" events are rather similar:

```
+static void
+wl_pointer_motion(void *data, struct wl_pointer *wl_pointer, uint32_t time,
+               wl_fixed_t surface_x, wl_fixed_t surface_y)
+{
+       struct client_state *client_state = data;
+       client_state->pointer_event.event_mask |= POINTER_EVENT_MOTION;
+       client_state->pointer_event.time = time;
+       client_state->pointer_event.surface_x = surface_x,
+               client_state->pointer_event.surface_y = surface_y;
+}
+
+static void
+wl_pointer_button(void *data, struct wl_pointer *wl_pointer, uint32_t serial,
+               uint32_t time, uint32_t button, uint32_t state)
+{
+       struct client_state *client_state = data;
+       client_state->pointer_event.event_mask |= POINTER_EVENT_BUTTON;
+       client_state->pointer_event.time = time;
+       client_state->pointer_event.serial = serial;
+       client_state->pointer_event.button = button,
+               client_state->pointer_event.state = state;
+}
```

Axis events are somewhat more complex, because there are two axes: horizontal and vertical. Thus, our `pointer_event` struct contains an array with two groups of axis events. Our code to handle these ends up something like this:

```
+static void
+wl_pointer_axis(void *data, struct wl_pointer *wl_pointer, uint32_t time,
+               uint32_t axis, wl_fixed_t value)
+{
+       struct client_state *client_state = data;
+       client_state->pointer_event.event_mask |= POINTER_EVENT_AXIS;
+       client_state->pointer_event.time = time;
+       client_state->pointer_event.axes[axis].valid = true;
+       client_state->pointer_event.axes[axis].value = value;
+}
+
+static void
```

```
+wl_pointer_axis_source(void *data, struct wl_pointer *wl_pointer,
+               uint32_t axis_source)
+{
+       struct client_state *client_state = data;
+       client_state->pointer_event.event_mask |= POINTER_EVENT_AXIS_SOURCE;
+       client_state->pointer_event.axis_source = axis_source;
+}
+
+static void
+wl_pointer_axis_stop(void *data, struct wl_pointer *wl_pointer,
+               uint32_t time, uint32_t axis)
+{
+       struct client_state *client_state = data;
+       client_state->pointer_event.time = time;
+       client_state->pointer_event.event_mask |= POINTER_EVENT_AXIS_STOP;
+       client_state->pointer_event.axes[axis].valid = true;
+}
+
+static void
+wl_pointer_axis_discrete(void *data, struct wl_pointer *wl_pointer,
+               uint32_t axis, int32_t discrete)
+{
+       struct client_state *client_state = data;
+       client_state->pointer_event.event_mask |= POINTER_EVENT_AXIS_DISCRETE;
+       client_state->pointer_event.axes[axis].valid = true;
+       client_state->pointer_event.axes[axis].discrete = discrete;
+}
```

Similarly straightforward, aside from the main change of updating whichever axis was affected. Note the use of the “valid” boolean as well: it’s possible that we’ll receive a pointer frame which updates one axis, but not another, so we use this “valid” value to determine which axes were updated in the frame event.

Speaking of which, it’s time for the main attraction: our “frame” handler.

```
+static void
+wl_pointer_frame(void *data, struct wl_pointer *wl_pointer)
+{
+       struct client_state *client_state = data;
+       struct pointer_event *event = &client_state->pointer_event;
+       fprintf(stderr, "pointer frame @ %d: ", event->time);
+
+       if (event->event_mask & POINTER_EVENT_ENTER) {
+               fprintf(stderr, "entered %f, %f ",
+                               wl_fixed_to_double(event->surface_x),
+                               wl_fixed_to_double(event->surface_y));
+       }
+
+       if (event->event_mask & POINTER_EVENT_LEAVE) {
+               fprintf(stderr, "leave");
+       }
+
+       if (event->event_mask & POINTER_EVENT_MOTION) {
```

```
+                fprintf(stderr, "motion %f, %f ",
+                                wl_fixed_to_double(event->surface_x),
+                                wl_fixed_to_double(event->surface_y));
+        }
+
+        if (event->event_mask & POINTER_EVENT_BUTTON) {
+                char *state = event->state == WL_POINTER_BUTTON_STATE_RELEASED ?
+                        "released" : "pressed";
+                fprintf(stderr, "button %d %s ", event->button, state);
+        }
+
+        uint32_t axis_events = POINTER_EVENT_AXIS
+                | POINTER_EVENT_AXIS_SOURCE
+                | POINTER_EVENT_AXIS_STOP
+                | POINTER_EVENT_AXIS_DISCRETE;
+        char *axis_name[2] = {
+                [WL_POINTER_AXIS_VERTICAL_SCROLL] = "vertical",
+                [WL_POINTER_AXIS_HORIZONTAL_SCROLL] = "horizontal",
+        };
+        char *axis_source[4] = {
+                [WL_POINTER_AXIS_SOURCE_WHEEL] = "wheel",
+                [WL_POINTER_AXIS_SOURCE_FINGER] = "finger",
+                [WL_POINTER_AXIS_SOURCE_CONTINUOUS] = "continuous",
+                [WL_POINTER_AXIS_SOURCE_WHEEL_TILT] = "wheel tilt",
+        };
+        if (event->event_mask & axis_events) {
+                for (size_t i = 0; i < 2; ++i) {
+                        if (!event->axes[i].valid) {
+                                continue;
+                        }
+                        fprintf(stderr, "%s axis ", axis_name[i]);
+                        if (event->event_mask & POINTER_EVENT_AXIS) {
+                                fprintf(stderr, "value %f ", wl_fixed_to_double(
+                                                        event->axes[i].value));
+                        }
+                        if (event->event_mask & POINTER_EVENT_AXIS_DISCRETE) {
+                                fprintf(stderr, "discrete %d ",
+                                                event->axes[i].discrete);
+                        }
+                        if (event->event_mask & POINTER_EVENT_AXIS_SOURCE) {
+                                fprintf(stderr, "via %s ",
+                                                axis_source[event->axis_source]);
+                        }
+                        if (event->event_mask & POINTER_EVENT_AXIS_STOP) {
+                                fprintf(stderr, "(stopped) ");
+                        }
+                }
+        }
+
+        fprintf(stderr, "\n");
+        memset(event, 0, sizeof(*event));
```

```
+}
```

It certainly is the longest of the bunch, isn't it? Hopefully it isn't too confusing, though. All we're doing here is pretty-printing the accumulated state for this frame to stderr. If you compile and run this again now, you should be able to wiggle your mouse over the window and see input events printed out!

Rigging up keyboard events

Let's update our `client_state` struct with some fields to store XKB state.

```
@@ -105,6 +107,9 @@ struct client_state {
        int width, height;
        bool closed;
        struct pointer_event pointer_event;
+       struct xkb_state *xkb_state;
+       struct xkb_context *xkb_context;
+       struct xkb_keymap *xkb_keymap;
};
```

We need the xkbcommon headers to define these. While we're at it, I'm going to pull in `assert.h` as well:

```
@@ -1,4 +1,5 @@
 #define _POSIX_C_SOURCE 200112L
+#include <assert.h>
 #include <errno.h>
 #include <fcntl.h>
 #include <limits.h>
@@ -9,6 +10,7 @@
 #include <time.h>
 #include <unistd.h>
 #include <wayland-client.h>
+#include <xkbcommon/xkbcommon.h>
 #include "xdg-shell-client-protocol.h"
```

We'll also need to initialize the xkb_context in our main function:

```
@@ -603,6 +649,7 @@ main(int argc, char *argv[])
        state.height = 480;
        state.wl_display = wl_display_connect(NULL);
        state.wl_registry = wl_display_get_registry(state.wl_display);
+       state.xkb_context = xkb_context_new(XKB_CONTEXT_NO_FLAGS);
        wl_registry_add_listener(state.wl_registry, &wl_registry_listener, &state);
        wl_display_roundtrip(state.wl_display);
```

Next, let's update our seat capabilities function to rig up our keyboard listener, too.

```
        } else if (!have_pointer && state->wl_pointer != NULL) {
                wl_pointer_release(state->wl_pointer);
                state->wl_pointer = NULL;
        }
+
+       bool have_keyboard = capabilities & WL_SEAT_CAPABILITY_KEYBOARD;
+
+       if (have_keyboard && state->wl_keyboard == NULL) {
+               state->wl_keyboard = wl_seat_get_keyboard(state->wl_seat);
+                wl_keyboard_add_listener(state->wl_keyboard,
+                                &wl_keyboard_listener, state);
+        } else if (!have_keyboard && state->wl_keyboard != NULL) {
+                wl_keyboard_release(state->wl_keyboard);
+                state->wl_keyboard = NULL;
+        }
 }
```

We'll have to define the `wl_keyboard_listener` we use here, too.

```
+static const struct wl_keyboard_listener wl_keyboard_listener = {
+        .keymap = wl_keyboard_keymap,
+        .enter = wl_keyboard_enter,
+        .leave = wl_keyboard_leave,
+        .key = wl_keyboard_key,
+        .modifiers = wl_keyboard_modifiers,
+        .repeat_info = wl_keyboard_repeat_info,
+};
```

And now, the meat of the changes. Let's start with the keymap:

```
+static void
+wl_keyboard_keymap(void *data, struct wl_keyboard *wl_keyboard,
+                uint32_t format, int32_t fd, uint32_t size)
+{
+        struct client_state *client_state = data;
+        assert(format == WL_KEYBOARD_KEYMAP_FORMAT_XKB_V1);
+
+        char *map_shm = mmap(NULL, size, PROT_READ, MAP_SHARED, fd, 0);
+        assert(map_shm != MAP_FAILED);
+
+        struct xkb_keymap *xkb_keymap = xkb_keymap_new_from_string(
+                        client_state->xkb_context, map_shm,
+                        XKB_KEYMAP_FORMAT_TEXT_V1, XKB_KEYMAP_COMPILE_NO_FLAGS);
+        munmap(map_shm, size);
+        close(fd);
+
+        struct xkb_state *xkb_state = xkb_state_new(xkb_keymap);
+        xkb_keymap_unref(client_state->xkb_keymap);
+        xkb_state_unref(client_state->xkb_state);
+        client_state->xkb_keymap = xkb_keymap;
+        client_state->xkb_state = xkb_state;
+}
```

Now we can see why we added `assert.h` — we're using it here to make sure that the keymap format is the one we expect. Then, we use mmap to map the file descriptor the compositor sent us to a `char *` pointer we can pass into `xkb_keymap_new_from_string`. Don't forget to munmap and close that fd afterwards — then we set up our XKB state. Note as well that we have also unrefed any previous XKB keymap or state that we had set up in a prior call to this function, in case the compositor changes the keymap at runtime.[1]

```
+static void
+wl_keyboard_enter(void *data, struct wl_keyboard *wl_keyboard,
+                uint32_t serial, struct wl_surface *surface,
+                struct wl_array *keys)
+{
+       struct client_state *client_state = data;
+       fprintf(stderr, "keyboard enter; keys pressed are:\n");
+       uint32_t *key;
+       wl_array_for_each(key, keys) {
+               char buf[128];
+               xkb_keysym_t sym = xkb_state_key_get_one_sym(
+                               client_state->xkb_state, *key + 8);
+               xkb_keysym_get_name(sym, buf, sizeof(buf));
+               fprintf(stderr, "sym: %-12s (%d), ", buf, sym);
+               xkb_state_key_get_utf8(client_state->xkb_state,
+                               *key + 8, buf, sizeof(buf));
+               fprintf(stderr, "utf8: '%s'\n", buf);
+       }
+}
```

When the keyboard "enters" our surface, we have received keyboard focus. The compositor forwards a list of keys which were already pressed at that time, and here we just enumerate them and log their keysym names and UTF-8 equivalent. We'll do something similar when keys are pressed:

```
+static void
+wl_keyboard_key(void *data, struct wl_keyboard *wl_keyboard,
+               uint32_t serial, uint32_t time, uint32_t key, uint32_t state)
+{
+       struct client_state *client_state = data;
+       char buf[128];
+       uint32_t keycode = key + 8;
+       xkb_keysym_t sym = xkb_state_key_get_one_sym(
+                       client_state->xkb_state, keycode);
+       xkb_keysym_get_name(sym, buf, sizeof(buf));
+       const char *action =
+               state == WL_KEYBOARD_KEY_STATE_PRESSED ? "press" : "release";
+       fprintf(stderr, "key %s: sym: %-12s (%d), ", action, buf, sym);
+       xkb_state_key_get_utf8(client_state->xkb_state, keycode,
+                       buf, sizeof(buf));
+       fprintf(stderr, "utf8: '%s'\n", buf);
+}
```

And finally, we add small implementations of the three remaining events:

```
+static void
+wl_keyboard_leave(void *data, struct wl_keyboard *wl_keyboard,
+               uint32_t serial, struct wl_surface *surface)
+{
+       fprintf(stderr, "keyboard leave\n");
+}
+
+static void
+wl_keyboard_modifiers(void *data, struct wl_keyboard *wl_keyboard,
+               uint32_t serial, uint32_t mods_depressed,
+               uint32_t mods_latched, uint32_t mods_locked,
+               uint32_t group)
```

```
+{
+       struct client_state *client_state = data;
+       xkb_state_update_mask(client_state->xkb_state,
+               mods_depressed, mods_latched, mods_locked, 0, 0, group);
+}
+
+static void
+wl_keyboard_repeat_info(void *data, struct wl_keyboard *wl_keyboard,
+               int32_t rate, int32_t delay)
+{
+       /* Left as an exercise for the reader */
+}
```

For modifiers, we could decode these further, but most applications won't need to. We just update the XKB state here. As for handling key repeat — this has a lot of constraints particular to your application. Do you want to repeat text input? Do you want to repeat keyboard shortcuts? How does the timing of these interact with your event loop? The answers to these questions is left for you to decide.

If you compile this again, you should be able to start typing into the window and see your input printed into the log. Huzzah!

Rigging up touch events

Finally, we'll add support for touch-capable devices. Like pointers, a "frame" event exists for touch devices. However, they're further complicated by the possibility that multiple touch points may be updated within a single frame. We'll add some more structures and enums to represent the accumulated state:

```
+enum touch_event_mask {
+       TOUCH_EVENT_DOWN = 1 << 0,
+       TOUCH_EVENT_UP = 1 << 1,
+       TOUCH_EVENT_MOTION = 1 << 2,
+       TOUCH_EVENT_CANCEL = 1 << 3,
+       TOUCH_EVENT_SHAPE = 1 << 4,
+       TOUCH_EVENT_ORIENTATION = 1 << 5,
+};
+
+struct touch_point {
+       bool valid;
+       int32_t id;
+       uint32_t event_mask;
+       wl_fixed_t surface_x, surface_y;
+       wl_fixed_t major, minor;
+       wl_fixed_t orientation;
+};
+
+struct touch_event {
+       uint32_t event_mask;
+       uint32_t time;
+       uint32_t serial;
+       struct touch_point points[10];
+};
```

Note that I've arbitrarily chosen 10 touchpoints here, with the assumption that most users will only ever use that many fingers. For larger, multi-user touch screens, you may need a higher limit. Additionally, some touch hardware supports fewer than 10 touch points concurrently — 8 is also common, and hardware which supports fewer still is common among older devices.

We'll add this struct to `client_state`:

```
@@ -110,6 +135,7 @@ struct client_state {
        struct xkb_state *xkb_state;
        struct xkb_context *xkb_context;
        struct xkb_keymap *xkb_keymap;
+       struct touch_event touch_event;
 };
```

And we'll update the seat capabilities handler to rig up a listener when touch support is available.

```
        } else if (!have_keyboard && state->wl_keyboard != NULL) {
                wl_keyboard_release(state->wl_keyboard);
                state->wl_keyboard = NULL;
        }
+
+       bool have_touch = capabilities & WL_SEAT_CAPABILITY_TOUCH;
+
+       if (have_touch && state->wl_touch == NULL) {
+               state->wl_touch = wl_seat_get_touch(state->wl_seat);
+               wl_touch_add_listener(state->wl_touch,
+                               &wl_touch_listener, state);
+       } else if (!have_touch && state->wl_touch != NULL) {
+               wl_touch_release(state->wl_touch);
+               state->wl_touch = NULL;
+       }
 }
```

We've repeated again the pattern of handling both the appearance and disappearance of touch capabilities on the seat, so we're robust to devices appearing and disappearing at runtime. It's less common for touch devices to be hotplugged, though.

Here's the listener itself:

```
+static const struct wl_touch_listener wl_touch_listener = {
+       .down = wl_touch_down,
+       .up = wl_touch_up,
+       .motion = wl_touch_motion,
+       .frame = wl_touch_frame,
+       .cancel = wl_touch_cancel,
+       .shape = wl_touch_shape,
+       .orientation = wl_touch_orientation,
+};
```

To deal with multiple touch points, we'll need to write a small helper function:

```
+static struct touch_point *
+get_touch_point(struct client_state *client_state, int32_t id)
+{
+       struct touch_event *touch = &client_state->touch_event;
+       const size_t nmemb = sizeof(touch->points) / sizeof(struct touch_point);
+       int invalid = -1;
+       for (size_t i = 0; i < nmemb; ++i) {
+               if (touch->points[i].id == id) {
+                       return &touch->points[i];
+               }
+               if (invalid == -1 && !touch->points[i].valid) {
+                       invalid = i;
+               }
+       }
+       if (invalid == -1) {
+               return NULL;
+       }
+       touch->points[invalid].valid = true;
+       touch->points[invalid].id = id;
+       return &touch->points[invalid];
+}
```

The basic purpose of this function is to pick a `touch_point` from the array we added to the `touch_event` struct, based on the touch ID we're receiving events for. If we find an existing `touch_point` for that ID, we return it. If not, we return the first available touch point. In case we run out, we return NULL.

Now we can take advantage of this to implement our first function: touch up.

```
+static void
+wl_touch_down(void *data, struct wl_touch *wl_touch, uint32_t serial,
+               uint32_t time, struct wl_surface *surface, int32_t id,
+               wl_fixed_t x, wl_fixed_t y)
+{
+       struct client_state *client_state = data;
+       struct touch_point *point = get_touch_point(client_state, id);
+       if (point == NULL) {
+               return;
+       }
+       point->event_mask |= TOUCH_EVENT_UP;
+       point->surface_x = wl_fixed_to_double(x),
+               point->surface_y = wl_fixed_to_double(y);
+       client_state->touch_event.time = time;
+       client_state->touch_event.serial = serial;
+}
```

Like the pointer events, we're also simply accumulating this state for later use. We don't yet know if this event represents a complete touch frame. Let's add something similar for touch up:

```
+static void
+wl_touch_up(void *data, struct wl_touch *wl_touch, uint32_t serial,
+               uint32_t time, int32_t id)
+{
+       struct client_state *client_state = data;
+       struct touch_point *point = get_touch_point(client_state, id);
+       if (point == NULL) {
+               return;
+        }
+        point->event_mask |= TOUCH_EVENT_UP;
+}
```

And for motion:

```
+static void
+wl_touch_motion(void *data, struct wl_touch *wl_touch, uint32_t time,
+                int32_t id, wl_fixed_t x, wl_fixed_t y)
+{
+        struct client_state *client_state = data;
+        struct touch_point *point = get_touch_point(client_state, id);
+        if (point == NULL) {
+                return;
+        }
+        point->event_mask |= TOUCH_EVENT_MOTION;
+        point->surface_x = x, point->surface_y = y;
+        client_state->touch_event.time = time;
+}
```

The touch cancel event is somewhat different, as it "cancels" all active touch points at once. We'll just store this in the `touch_event`'s top-level event mask.

```
+static void
+wl_touch_cancel(void *data, struct wl_touch *wl_touch)
+{
+        struct client_state *client_state = data;
+        client_state->touch_event.event_mask |= TOUCH_EVENT_CANCEL;
+}
```

The shape and orientation events are similar to up, down, and move, however, in that they inform us about the dimensions of a specific touch point.

```
+static void
+wl_touch_shape(void *data, struct wl_touch *wl_touch,
+                int32_t id, wl_fixed_t major, wl_fixed_t minor)
+{
+        struct client_state *client_state = data;
+        struct touch_point *point = get_touch_point(client_state, id);
+        if (point == NULL) {
+                return;
+        }
+        point->event_mask |= TOUCH_EVENT_SHAPE;
+        point->major = major, point->minor = minor;
+}
+
+static void
+wl_touch_orientation(void *data, struct wl_touch *wl_touch,
+                int32_t id, wl_fixed_t orientation)
+{
+        struct client_state *client_state = data;
+        struct touch_point *point = get_touch_point(client_state, id);
+        if (point == NULL) {
+                return;
+        }
+       point->event_mask |= TOUCH_EVENT_ORIENTATION;
+       point->orientation = orientation;
+}
```

And finally, upon receiving a frame event, we can interpret all of this accumulated state as a single input event, much like our pointer code.

```
+static void
+wl_touch_frame(void *data, struct wl_touch *wl_touch)
+{
+       struct client_state *client_state = data;
+       struct touch_event *touch = &client_state->touch_event;
+       const size_t nmemb = sizeof(touch->points) / sizeof(struct touch_point);
+       fprintf(stderr, "touch event @ %d:\n", touch->time);
+
+       for (size_t i = 0; i < nmemb; ++i) {
+               struct touch_point *point = &touch->points[i];
+               if (!point->valid) {
+                       continue;
+               }
+               fprintf(stderr, "point %d: ", touch->points[i].id);
+
+               if (point->event_mask & TOUCH_EVENT_DOWN) {
+                       fprintf(stderr, "down %f,%f ",
+                                       wl_fixed_to_double(point->surface_x),
+                                       wl_fixed_to_double(point->surface_y));
+               }
+
+               if (point->event_mask & TOUCH_EVENT_UP) {
+                       fprintf(stderr, "up ");
+               }
+
+               if (point->event_mask & TOUCH_EVENT_MOTION) {
+                       fprintf(stderr, "motion %f,%f ",
+                                       wl_fixed_to_double(point->surface_x),
+                                       wl_fixed_to_double(point->surface_y));
+               }
+
+               if (point->event_mask & TOUCH_EVENT_SHAPE) {
+                       fprintf(stderr, "shape %fx%f ",
+                                       wl_fixed_to_double(point->major),
+                                       wl_fixed_to_double(point->minor));
+               }
+
+               if (point->event_mask & TOUCH_EVENT_ORIENTATION) {
+                       fprintf(stderr, "orientation %f ",
+                                       wl_fixed_to_double(point->orientation));
+               }
+
+               point->valid = false;
+               fprintf(stderr, "\n");
+       }
+}
```

Compile and run this again, and you'll be able to see touch events printed to stderr as you interact with your touch device (assuming you have such a device to test with). And now our client supports input!

What's next?

There are a lot of different kinds of input devices, so extending our code to support them was a fair bit of work — our code has grown by 2.5× in this chapter alone. The rewards should feel pretty great, though, as you are now familiar with enough Wayland concepts (and code) that you can implement a lot of clients.

There's still a little bit more to learn — in the last few chapters, we'll cover popup windows, context menus, interactive window moving and resizing, clipboard and drag & drop support, and, later, a handful of interesting protocol extensions which support more niche use-cases. I definitely recommend reading at least chapter 10.1 before you start building your own client, as it covers things like having the window resized at the compositor's request.

1

This actually does happen in practice!

XDG shell in depth

So far we've managed to display something on-screen in a top-level application window, but there's more to XDG shell that we haven't fully appreciated yet. Even the simplest application would be well-served to implement the configuration lifecycle correctly, and xdg-shell offers useful features to more complex application as well. The full breadth of xdg-shell's feature set includes client/server negotiation on window size, multi-window hierarchies, client-side decorations, and semantic positioning for windows like context menus.

Configuration & lifecycle

Previously, we created a window at a fixed size of our choosing: 640x480. However, the compositor will often have an opinion about what size our window should assume, and we may want to communicate our preferences as well. Failure to do so will often lead to undesirable behavior, like parts of your window being cut off by a compositor who's trying to tell you to make your surface smaller.

The compositor can offer additional clues to the application about the context in which it's being shown. It can let you know if your application is maximized or fullscreen, tiled on one or more sides against other windows or the edge of the display, focused or idle, and so on. As `wl_surface` is used to atomically communicate surface changes from client to server, the `xdg_surface` interface provides the following two messages for the compositor to suggest changes and the client to acknowledge them:

```xml
<request name="ack_configure">
  <arg name="serial" type="uint" />
</request>

<event name="configure">
  <arg name="serial" type="uint" />
</event>
```

On their own, these messages carry little meaning. However, each subclass of `xdg_surface` (`xdg_toplevel` and `xdg_popup`) have additional events that the server can send ahead of "configure", to make each of the suggestions we've mentioned so far. The server will send all of this state; maximized, focused, a suggested size; then a `configure` event with a serial. When the client has assumed a state consistent with these suggestions, it sends an `ack_configure` request with the same serial to indicate this. Upon the next commit to the associated `wl_surface`, the compositor will consider the state consistent.

XDG top-level lifecycle

Our example code from chapter 7 works, but it's not the best citizen of the desktop right now. It does not assume the compositor's recommended size, and if the user tries to close the window, it won't go away. Responding to these compositor-supplied events implicates two Wayland events: `configure` and `close`.

```
<event name="configure">
  <arg name="width" type="int"/>
  <arg name="height" type="int"/>
  <arg name="states" type="array"/>
</event>

<event name="close" />
```

The width and height are the compositor's preferred size for the window1, and states is an array of the following values:

```
<enum name="state">
  <entry name="maximized" />
  <entry name="fullscreen" />
  <entry name="resizing" />
  <entry name="activated" />
  <entry name="tiled_left" />
  <entry name="tiled_right" />
  <entry name="tiled_top" />
  <entry name="tiled_bottom" />
</enum>
```

The close event can be ignored, a typical reason being to show the user a confirmation to save their unsaved work. Our example code from chapter 7 can be updated fairly easily to support these events:

```
diff --git a/client.c b/client.c
--- a/client.c
+++ b/client.c
@@ -70,9 +70,10 @@ struct client_state {
    struct xdg_surface *xdg_surface;
    struct xdg_toplevel *xdg_toplevel;
    /* State */
-   bool closed;
    float offset;
    uint32_t last_frame;
+   int width, height;
+   bool closed;
 };

 static void wl_buffer_release(void *data, struct wl_buffer *wl_buffer) {
```

```
@@ -86,7 +87,7 @@ static const struct wl_buffer_listener wl_buffer_listener = {
 static struct wl_buffer *
 draw_frame(struct client_state *state)
 {
-    const int width = 640, height = 480;
+    int width = state->width, height = state->height;
     int stride = width * 4;
     int size = stride * height;

@@ -124,6 +125,32 @@ draw_frame(struct client_state *state)
     return buffer;
 }

+static void
+xdg_toplevel_configure(void *data,
+        struct xdg_toplevel *xdg_toplevel, int32_t width, int32_t height,
+        struct wl_array *states)
+{
+    struct client_state *state = data;
+    if (width == 0 || height == 0) {
+        /* Compositor is deferring to us */
+        return;
+    }
+    state->width = width;
+    state->height = height;
+}
+
+static void
+xdg_toplevel_close(void *data, struct xdg_toplevel *toplevel)
+{
+    struct client_state *state = data;
+    state->closed = true;
+}
+
+static const struct xdg_toplevel_listener xdg_toplevel_listener = {
+    .configure = xdg_toplevel_configure,
+    .close = xdg_toplevel_close,
+};
+
 static void
 xdg_surface_configure(void *data,
         struct xdg_surface *xdg_surface, uint32_t serial)
@@ -163,7 +190,7 @@ wl_surface_frame_done(void *data, struct wl_callback *cb, uint32
     cb = wl_surface_frame(state->wl_surface);
     wl_callback_add_listener(cb, &wl_surface_frame_listener, state);

-    /* Update scroll amount at 8 pixels per second */
+    /* Update scroll amount at 24 pixels per second */
     if (state->last_frame != 0) {
         int elapsed = time - state->last_frame;
         state->offset += elapsed / 1000.0 * 24;
```

```
@@ -217,6 +244,8 @@ int
 main(int argc, char *argv[])
 {
    struct client_state state = { 0 };
+   state.width = 640;
+   state.height = 480;
    state.wl_display = wl_display_connect(NULL);
    state.wl_registry = wl_display_get_registry(state.wl_display);
    wl_registry_add_listener(state.wl_registry, &wl_registry_listener, &state);
@@ -227,6 +256,8 @@ main(int argc, char *argv[])
            state.xdg_wm_base, state.wl_surface);
    xdg_surface_add_listener(state.xdg_surface, &xdg_surface_listener, &state);
    state.xdg_toplevel = xdg_surface_get_toplevel(state.xdg_surface);
+   xdg_toplevel_add_listener(state.xdg_toplevel,
+           &xdg_toplevel_listener, &state);
    xdg_toplevel_set_title(state.xdg_toplevel, "Example client");
    wl_surface_commit(state.wl_surface);
```

If you compile and run this client again, you'll notice that it's a lot more well-behaved than before.

Requesting state changes

The client can also request that the compositor put the client into one of these states, or place constraints on the size of the window.

```
<request name="set_max_size">
  <arg name="width" type="int"/>
  <arg name="height" type="int"/>
</request>

<request name="set_min_size">
  <arg name="width" type="int"/>
  <arg name="height" type="int"/>
</request>

<request name="set_maximized" />

<request name="unset_maximized" />

<request name="set_fullscreen" />
  <arg name="output"
    type="object"
    interface="wl_output"
    allow-null="true"/>
</request>

<request name="unset_fullscreen" />

<request name="set_minimized" />
```

The compositor indicates its acknowledgement of these requests by sending a corresponding configure event.

1

This takes into account the window geometry sent by the `set_window_geometry` request from the client. The suggested size only includes the space represented by the window geometry.

Popups

When designing software which utilizes application windows, there are many cases where smaller secondary surfaces are used for various purposes. Some examples include context menus which appear on right click, dropdown boxes to select a value from several options, contextual hints which are shown when you hover the mouse over a UI element, or menus and toolbars along the top and bottom of a window. Often these will be nested, for example, by following a path like "File → Recent Documents → Example.odt".

For Wayland, the XDG shell provides facilities for managing these windows: popups. We looked at `xdg_surface`'s " `get_toplevel`" request for creating top-level application windows earlier. In the case of popups, the " `get_popup`" request is used instead.

```
<request name="get_popup">
  <arg name="id" type="new_id" interface="xdg_popup"/>
  <arg name="parent" type="object" interface="xdg_surface" allow-null="true"/>
  <arg name="positioner" type="object" interface="xdg_positioner"/>
</request>
```

The first and second arguments are reasonably self-explanatory, but the third one introduces a new concept: positioners. The purpose of the positioner is, as the name might suggest, to *position* the new popup. This is used to allow the compositor to participate in the positioning of popups using its privileged information, for example to avoid having the popup extend past the edge of the display. We'll discuss positioners in chapter 10.4, for now you can simply create one and pass it in without further configuration to achieve reasonably sane default behavior, utilizing the appropriate `xdg_wm_base` request:

```
<request name="create_positioner">
  <arg name="id" type="new_id" interface="xdg_positioner"/>
</request>
```

So, in short, we can:

1. Create a new `wl_surface`
2. Obtain an `xdg_surface` for it
3. Create a new `xdg_positioner`, saving its configuration for chapter 10.4
4. Create an `xdg_popup` from our XDG surface and XDG positioner, assigning its parent to the `xdg_toplevel` we created earlier.

Then we can render and attach buffers to our popup surface with the same lifecyle discussed earlier. We also have access to a few other popup-specific features.

Configuration

Like the XDG toplevel configure event, the compositor has an event which it may use to suggest the size for your popup to assume. Unlike toplevels, however, this also includes a positioning event, which informs the client as to the position of the popup relative to its parent surface.

```
<event name="configure">
  <arg name="x" type="int"
 summary="x position relative to parent surface window geometry"/>
  <arg name="y" type="int"
 summary="y position relative to parent surface window geometry"/>
  <arg name="width" type="int" summary="window geometry width"/>
  <arg name="height" type="int" summary="window geometry height"/>
</event>
```

The client can influence these values with the XDG positioner, to be discussed in chapter 10.4.

Popup grabs

Popup surfaces will often want to "grab" all input, for example to allow the user to use the arrow keys to select different menu items. This is facilitated through the grab request:

```
<request name="grab">
  <arg name="seat" type="object" interface="wl_seat" />
  <arg name="serial" type="uint" />
</request>
```

A prerequisite of this request is having received a qualifying input event, such as a right click. The serial from this input event should be used in this request. These semantics are covered in detail in chapter 9. The compositor can cancel this grab later, for example if the user presses escape or clicks outside of your popup.

Dismissal

In these cases where the compositor dismisses your popup, such as when the escape key is pressed, the following event is sent:

```
<event name="popup_done" />
```

To avoid race conditions, the compositor keeps the popup structures in memory and services requests for them even after their dismissal. For more detail about object lifetimes and race conditions, see chapter 2.4.

Destroying popups

Client-initiated destruction of a popup is fairly straightforward:

```
<request name="destroy" type="destructor" />
```

However, one detail bears mentioning: you must destroy all popups from the top-down. The only popup you can destroy at any given moment is the top-most one. If you don't, you'll be disconnected with a protocol error.

Interactive move and resize

Many application windows have interactive UI elements the user can use to drag around or resize windows. Many Wayland clients, by default, expect to be responsible for their own window decorations to provide these interactive elements. On X11, application windows could position themselves independently anywhere on the screen, and used this to facilitate these interactions.

However, a deliberate design trait of Wayland makes application windows ignorant of their exact placement on screen or relative to other windows. This decision affords Wayland compositors a greater deal of flexibility — windows could be shown in several places at once, arranged in the 3D space of a VR scene, or presented in any other novel way. Wayland is designed to be generic and widely applicable to many devices and form factors.

To balance these two design needs, XDG toplevels offer two requests which can be used to ask the compositor to begin an interactive move or resize operation. The relevant parts of the interface are:

```
<request name="move">
  <arg name="seat" type="object" interface="wl_seat" />
  <arg name="serial" type="uint" />
</request>
```

Like the popup creation request explained in the previous chapter, you have to provide an input event serial to start an interactive operation. For example, when you receive a mouse button down event, you can use that event's serial to begin an interactive move operation. The compositor will take over from here, and begin an interactive operation to your window in its internal coordinate space.

Resizing is a bit more complex, due to the need to specify which edges or corners of the window are implicated in the operation:

```
<enum name="resize_edge">
  <entry name="none" value="0"/>
  <entry name="top" value="1"/>
  <entry name="bottom" value="2"/>
  <entry name="left" value="4"/>
  <entry name="top_left" value="5"/>
  <entry name="bottom_left" value="6"/>
  <entry name="right" value="8"/>
  <entry name="top_right" value="9"/>
  <entry name="bottom_right" value="10"/>
</enum>

<request name="resize">
  <arg name="seat" type="object" interface="wl_seat" />
  <arg name="serial" type="uint" />
  <arg name="edges" type="uint" />
</request>
```

But otherwise, it functions much the same. If the user clicks and drags along the bottom-left corner of your window, you may want to send an interactive resize request with the corresponding seat & serial, and set the edges argument to bottom_left.

There's one additional request necessary for clients to totally implement interactive client-side window decorations:

```
<request name="show_window_menu">
  <arg name="seat" type="object" interface="wl_seat" />
  <arg name="serial" type="uint" />
  <arg name="x" type="int" />
  <arg name="y" type="int" />
</request>
```

A contextual menu offering window operations, such as closing or minimizing the window, is often raised when clicking on window decorations. For clients where window decorations are managed by the client, this serves to link the client-driven interactions with compositor-driven meta operations like minimizing windows. If your client uses client-side decorations, you may use this request for this purpose.

xdg-decoration

The last detail which bears mentioning when discussing the behavior of client-side decorations is the protocol which governs the negotiation of their use in the first place. Different Wayland clients and servers may have different preferences about the use of server-side or client-side window decorations. To express these intentions, a protocol extension is used: `xdg-decoration`. It can be found in wayland-protocols. The protocol provides a global:

```
<interface name="zxdg_decoration_manager_v1" version="1">
  <request name="destroy" type="destructor" />

  <request name="get_toplevel_decoration">
    <arg name="id" type="new_id" interface="zxdg_toplevel_decoration_v1"/>
    <arg name="toplevel" type="object" interface="xdg_toplevel"/>
  </request>
</interface>
```

You may pass your xdg_toplevel object into the `get_toplevel_decoration` request to obtain an object with the following interface:

```
<interface name="zxdg_toplevel_decoration_v1" version="1">
  <request name="destroy" type="destructor" />

  <enum name="mode">
    <entry name="client_side" value="1" />
    <entry name="server_side" value="2" />
  </enum>

  <request name="set_mode">
    <arg name="mode" type="uint" enum="mode" />
  </request>

  <request name="unset_mode" />

  <event name="configure">
    <arg name="mode" type="uint" enum="mode" />
  </event>
</interface>
```

The `set_mode` request is used to express a preference from the client, and `unset_mode` is used to express no preference. The compositor will then use the `configure` event to tell the client whether or not to use client-side decorations. For more details, consult the full XML.

Positioners

When we introduced pop-ups a few pages ago, we noted that you had to provide a positioner object when creating the pop-up. We asked you not to worry about it and just use the defaults, because it's a complicated interface and was beside the point. Now, we'll explore this complex interface in depth.

When you open a pop-up window, it's shown in a windowing system which has constraints that your client is not privy to. For example, Wayland clients are unaware of where their windows are being shown on-screen. Therefore, if you right click a window, the client does not possess the necessary information to determine that the resulting pop-up might end up running itself off the edge of the screen. The positioner is designed to address these issues, by letting the client specify certain constraints in how the pop-up can be moved or resized, and then the compositor, being in full possession of the facts, can make the final call on how to accommodate.

The Basics

```
<request name="destroy" type="destructor"></request>
```

This destroys the positioner when you're done with it. You can call this after your pop-up has been created.

```
<request name="set_size">
  <arg name="width" type="int" />
  <arg name="height" type="int" />
</request>
```

The set_size request is used to set the size of the pop-up window being created.

All clients which use a positioner will use these two requests. Now, let's get to the interesting ones.

Anchoring

Clipboard access

Data offers

Drag & drop

Popular protocol extensions

Accurate timing

Pointer constraints

Extended clipboard

Desktop shell components

Miscellaneous extensions

Writing protocol extensions

Acknowledgements

TODO (there's a lot)