Learn Wayland by writing a GUI from scratch


Philippe Gaultier

Back to all articles

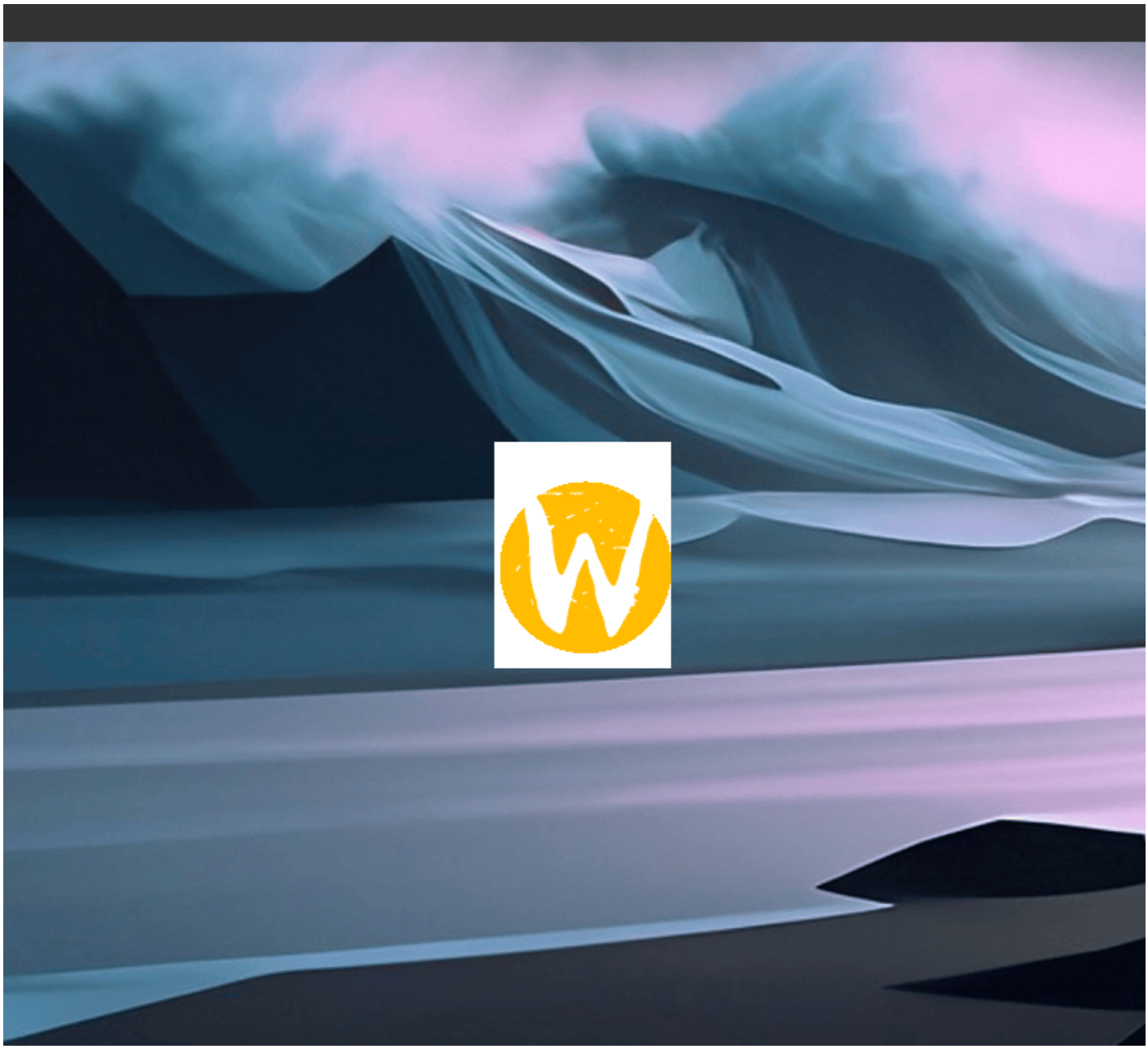
Published on 2023-10-12

Table of contents

Discussions: Hacker News, Lobsters.

Wayland is all the rage those days. Distributions left and right switch to it, many readers of my previous article on writing a X11 GUI from scratch in x86_64 assembly asked for a follow-up article about Wayland, and I now run Wayland on my desktop. So here we go, let's write a (very simple) GUI program with Wayland, without any libraries, this time in C.

Here is what we are working towards:



We display the Wayland logo in its own window (we can see the mountain wallpaper in the background since we use a fixed size buffer). It's not quite Visual Studio yet, I know, but it's a good foundation for more in future articles, perhaps.

Why not in assembly again you ask? Well, the Wayland protocol has some peculiarities that necessitate the use of some C standard library macros to make it work reliably on different platforms (Linux, FreeBSD, etc): namely, sending a file descriptor over a UNIX socket. Maybe it could be done in assembly, but it would be much more tedious. Also, the Wayland protocol is completely asynchronous by nature, whereas the X11 protocol was more of a request-(maybe) response chatter, and as such, we have to keep track of some state in our program, and C makes it easier.

Now, if you want to follow along and translate the C snippets into assembly, go for it, it is doable, just tedious.

> If you spot an error, please open a Github issue!

What do we need?

Not much: We'll use C99 so any C compiler of the last 20 years will do. Having a Wayland desktop to test the application will also greatly help.

Note that I have only run it on Linux; it should work (meaning: compile and run) on other platforms running Wayland such as FreeBSD, it's just that I have not tried.

Note that the code in this article has not been written in the most robust way, it simply exits when things are not how they should be for example. So, not production ready, but still a good learning resource and a good foundation for more.

Wayland basics

Wayland is a protocol specification for GUI applications (and more), in short. We will write the client side, while the server side is a compositor which understands our protocol. If you have a Wayland desktop right now, a Wayland compositor is already running so there is nothing to do.

Much like X11, a client opens a UNIX socket, sends some commands in a specific format (which are different from the X11 ones), to open a window and the server can also send messages to notify the client to resize the window, that there is some keyboard input, etc. It's important to note that contrary to X11, in Wayland, the client only has access to its own window.

It is also interesting to note that Wayland is quite a limited protocol and any GUI will have to use extension protocols.

Most client applications use `libwayland` which is a library composed of C files that are autogenerated from a XML file describing the protocol. The same goes for extension protocols: they simply are one XML file that is turned into C files, which are then compiled and linked to a GUI application.

Now, we will not do any of this: we will instead write our own serialization and deserialization functions, which is really not a lot of work as you will see.

There are many advantages:

- No need to link to external libraries: no build system complexities, no dynamic linking issues, and so on.
- We do not have to use the callback system that `libwayland` requires.
- We can use the I/O mechanism we wish to listen to incoming messages: blocking, `poll`, `select`, `epoll`, `io_uring`, `kqueue` on some systems, etc. Here, we will use blocking calls for simplicity but the world is your oyster.
- Easy troubleshooting: 100% of the code is our own.
- No XML
- The protocols we will use are stable so the numeric values on the wire should not change underneath us, but in the unlikely event they do, we simply have to fix them in our code and compile again.

So at this point you might be thinking: this is going to be so much work! Well, not really. Here are **all** of the Wayland protocol numeric values we will need, including the extension protocols:

C

```
1 static const uint32_t wayland_display_object_id = 1;
2 static const uint16_t wayland_wl_registry_event_global = 0;
3 static const uint16_t wayland_shm_pool_event_format = 0;
4 static const uint16_t wayland_wl_buffer_event_release = 0;
5 static const uint16_t wayland_xdg_wm_base_event_ping = 0;
6 static const uint16_t wayland_xdg_toplevel_event_configure = 0;
7 static const uint16_t wayland_xdg_toplevel_event_close = 1;
8 static const uint16_t wayland_xdg_surface_event_configure = 0;
9 static const uint16_t wayland_wl_display_get_registry_opcode = 1;
10 static const uint16_t wayland_wl_registry_bind_opcode = 0;
11 static const uint16_t wayland_wl_compositor_create_surface_opcode = 0;
12 static const uint16_t wayland_xdg_wm_base_pong_opcode = 3;
13 static const uint16_t wayland_xdg_surface_ack_configure_opcode = 4;
14 static const uint16_t wayland_wl_shm_create_pool_opcode = 0;
15 static const uint16_t wayland_xdg_wm_base_get_xdg_surface_opcode = 2;
16 static const uint16_t wayland_wl_shm_pool_create_buffer_opcode = 0;
17 static const uint16_t wayland_wl_surface_attach_opcode = 1;
18 static const uint16_t wayland_xdg_surface_get_toplevel_opcode = 1;
19 static const uint16_t wayland_wl_surface_commit_opcode = 6;
20 static const uint16_t wayland_wl_display_error_event = 0;
21 static const uint32_t wayland_format_xrgb8888 = 1;
22 static const uint32_t wayland_header_size = 8;
23 static const uint32_t color_channels = 4;
```

So, not that much!

Opening a socket

The first step is opening a UNIX domain socket. Note that this step is exactly the same as for X11, save for the path of the socket. Also, X11 is designed to be used over the network so it does not have to be a UNIX domain socket, on the same machine - but everybody does so on their desktop machine anyway.

To craft the socket path, we follow these simple steps:

- If `$WAYLAND_DISPLAY` is set, attempt to connect to `$XDG_RUNTIME_DIR/$WAYLAND_DISPLAY`
- Otherwise, attempt to connect to `$XDG_RUNTIME_DIR/wayland-0`
- Otherwise, fail

Here goes, along with two utility macros we'll use everywhere:

C

```
1 #define cstring_len(s) (sizeof(s) - 1)
2
3 #define roundup_4(n) (((n) + 3) & -4)
4
5 static int wayland_display_connect() {
6   char *xdg_runtime_dir = getenv("XDG_RUNTIME_DIR");
7   if (xdg_runtime_dir == NULL)
8     return EINVAL;
9
10   uint64_t xdg_runtime_dir_len = strlen(xdg_runtime_dir);
11
12   struct sockaddr_un addr = {.sun_family = AF_UNIX};
13   assert(xdg_runtime_dir_len <= cstring_len(addr.sun_path));
14   uint64_t socket_path_len = 0;
15
16   memcpy(addr.sun_path, xdg_runtime_dir, xdg_runtime_dir_len);
17   socket_path_len += xdg_runtime_dir_len;
18
19   addr.sun_path[socket_path_len++] = '/';
20
21   char *wayland_display = getenv("WAYLAND_DISPLAY");
22   if (wayland_display == NULL) {
23     char wayland_display_default[] = "wayland-0";
24     uint64_t wayland_display_default_len =
cstring_len(wayland_display_default);
25
26     memcpy(addr.sun_path + socket_path_len, wayland_display_default,
27            wayland_display_default_len);
28     socket_path_len += wayland_display_default_len;
29   } else {
30     uint64_t wayland_display_len = strlen(wayland_display);
31     memcpy(addr.sun_path + socket_path_len, wayland_display,
32            wayland_display_len);
33     socket_path_len += wayland_display_len;
34   }
35
36   int fd = socket(AF_UNIX, SOCK_STREAM, 0);
37   if (fd == -1)
38     exit(errno);
39
40   if (connect(fd, (struct sockaddr *)&addr, sizeof(addr)) == -1)
41     exit(errno);
42
```

```
43   return fd;
44 }
```

In Wayland, there is no connection setup to do, such as sending some special messages, so there is nothing more to do.

Creating a registry

Now, to do anything useful, we want to create a registry: it is an object that allows us to query at runtime the capabilities of the compositor.

In Wayland, to create an object, we simply send the right message followed by an id of our own. Ids should be unique so we simply increment a number each time we want to create a new resource. After this is done, we will remember this number to be able to refer to it in later messages:

This is coincidentally our first message we send, so let's briefly go over the structure of a Wayland message. It is basically a RPC mechanism. All bytes are in the host endianness so there is nothing special to do about it:

- 4 bytes: The id of the resource ('object') we want to call a method on
- 2 bytes: The opcode of the method we want to call
- 2 bytes: The size of the message
- Depending on the method, arguments in their wire format follow

The object id in this case is `1`, which is the singleton `wl_display` that already exists. The method is: `get_registry(u32 new_id)` whose opcode we listed before. The sole argument takes 4 bytes and is this incremental number we keep track of client-side. It does not necessarily have to be incremental, but that's what `libwayland` does and also it's the easiest.

For convenience and efficiency, we always craft the message on the stack and do not allocate dynamic memory.

We first introduce a few utility functions to read and write parts of messages:

C

```
1 static void buf_write_u32(char *buf, uint64_t *buf_size, uint64_t
buf_cap,
2                           uint32_t x) {
3   assert(*buf_size + sizeof(x) <= buf_cap);
4   assert(((size_t)buf + *buf_size) % sizeof(x) == 0);
5
6   *(uint32_t *)(buf + *buf_size) = x;
7   *buf_size += sizeof(x);
8 }
9
10 static void buf_write_u16(char *buf, uint64_t *buf_size, uint64_t
```

```
buf_cap,
11                            uint16_t x) {
12   assert(*buf_size + sizeof(x) <= buf_cap);
13   assert(((size_t)buf + *buf_size) % sizeof(x) == 0);
14
15   *(uint16_t *)(buf + *buf_size) = x;
16   *buf_size += sizeof(x);
17 }
18
19 static void buf_write_string(char *buf, uint64_t *buf_size, uint64_t
buf_cap,
20                              char *src, uint32_t src_len) {
21   assert(*buf_size + src_len <= buf_cap);
22
23   buf_write_u32(buf, buf_size, buf_cap, src_len);
24   memcpy(buf + *buf_size, src, roundup_4(src_len));
25   *buf_size += roundup_4(src_len);
26 }
27
28 static uint32_t buf_read_u32(char **buf, uint64_t *buf_size) {
29   assert(*buf_size >= sizeof(uint32_t));
30   assert((size_t)*buf % sizeof(uint32_t) == 0);
31
32   uint32_t res = *(uint32_t *)(*buf);
33   *buf += sizeof(res);
34   *buf_size -= sizeof(res);
35
36   return res;
37 }
38
39 static uint16_t buf_read_u16(char **buf, uint64_t *buf_size) {
40   assert(*buf_size >= sizeof(uint16_t));
41   assert((size_t)*buf % sizeof(uint16_t) == 0);
42
43   uint16_t res = *(uint16_t *)(*buf);
44   *buf += sizeof(res);
45   *buf_size -= sizeof(res);
46
47   return res;
48 }
49
50 static void buf_read_n(char **buf, uint64_t *buf_size, char *dst,
uint64_t n) {
51   assert(*buf_size >= n);
```

```
52
53   memcpy(dst, *buf, n);
54
55   *buf += n;
56   *buf_size -= n;
57 }
```

And we finally can send our first message:

C

```
1 static uint32_t wayland_wl_display_get_registry(int fd) {
2   uint64_t msg_size = 0;
3   char msg[128] = "";
4   buf_write_u32(msg, &msg_size, sizeof(msg), wayland_display_object_id);
5
6   buf_write_u16(msg, &msg_size, sizeof(msg),
7                 wayland_wl_display_get_registry_opcode);
8
9   uint16_t msg_announced_size =
10       wayland_header_size + sizeof(wayland_current_id);
11   assert(roundup_4(msg_announced_size) == msg_announced_size);
12   buf_write_u16(msg, &msg_size, sizeof(msg), msg_announced_size);
13
14   wayland_current_id++;
15   buf_write_u32(msg, &msg_size, sizeof(msg), wayland_current_id);
16
17   if ((int64_t)msg_size != send(fd, msg, msg_size, MSG_DONTWAIT))
18     exit(errno);
19
20   printf("-> wl_display@%u.get_registry: wl_registry=%u\n",
21          wayland_display_object_id, wayland_current_id);
22
23   return wayland_current_id;
24 }
```

And by calling it, we have created our very first Wayland resource!

From this point on, the utility functions to send Wayland messages (`wayland_*`) will not be included in the code snippets for brevity (but you will find all of the code at the end!), just because they all are similar to the one above.

Shared memory: the frame buffer

To avoid drawing a frame in our application, and having to send all of the bytes over the socket to the compositor, there is a smarter approach: the buffer should be shared between the two processes, so that no copying is required.

We need to synchronize the access between the two so that presenting the frame does not happen while we are still drawing it, and Wayland has us covered here.

First, we need to create this buffer. We are going to make it easier for us by using a fixed size. Wayland is going to send us 'resize' events, whenever the window size changes, which we will acknowledge and ignore. This is done here just to simplify a bit the article, obviously in a real application, you would resize the buffer.

First, we introduce a struct that will hold all of the client-side state so that we remember which resources we have created so far. We also need a super simple state machine for later to track whether the surface (i.e. the 'frame' data) should be drawn to, as mentioned:

C

```c
typedef enum state_state_t state_state_t;
enum state_state_t {
  STATE_NONE,
  STATE_SURFACE_ACKED_CONFIGURE,
  STATE_SURFACE_ATTACHED,
};

typedef struct state_t state_t;
struct state_t {
  uint32_t wl_registry;
  uint32_t wl_shm;
  uint32_t wl_shm_pool;
  uint32_t wl_buffer;
  uint32_t xdg_wm_base;
  uint32_t xdg_surface;
  uint32_t wl_compositor;
  uint32_t wl_surface;
  uint32_t xdg_toplevel;
  uint32_t stride;
  uint32_t w;
  uint32_t h;
  uint32_t shm_pool_size;
  int shm_fd;
  uint8_t *shm_pool_data;

  state_state_t state;
};
```

We use it so in `main()`:

C

```
state_t state = {
    .wl_registry = wayland_wl_display_get_registry(fd),
    .w = 117,
    .h = 150,
    .stride = 117 * color_channels,
};

// Single buffering.
state.shm_pool_size = state.h * state.stride;
```

The window is a rectangle, of width `w` and height `h`. We will use the color format `xrgb8888` which is 4 color channels, each taking one bytes, so 4 bytes per pixel. This is one of the two formats that is guaranteed to be supported by the compositor per the specification. The stride counts how many bytes a horizontal row takes: `w * 4`.

And so, our buffer size for the frame is : `w * h * 4`. We use single buffering again for simplicity and also because we want to display a static image.

We could choose to use double or even triple buffering, thus respectively doubling or tripling the buffer size. The compositor is none the wiser - we would simply keep a counter client-side that increments each time we render a frame (and wraps around back to 0 when reaching the number of buffers), we would draw in the right location of this big buffer (i.e. at an offset), and attach the right part of the buffer to the surface. All the Wayland calls would remain the same.

Alright, time to really create this buffer, and not only keep track of its size:

C

```
static void create_shared_memory_file(uint64_t size, state_t *state) {
  char name[255] = "/";
  for (uint64_t i = 1; i < cstring_len(name); i++) {
    name[i] = ((double)rand()) / (double)RAND_MAX * 26 + 'a';
  }

  int fd = shm_open(name, O_RDWR | O_EXCL | O_CREAT, 0600);
  if (fd == -1)
    exit(errno);

   assert(shm_unlink(name) != -1);

   if (ftruncate(fd, size) == -1)
     exit(errno);
```

```
15
16   state->shm_pool_data =
17       mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
18   assert((void*)-1 != state->shm_pool_data);
19   assert(state->shm_pool_data != NULL);
20   state->shm_fd = fd;
21 }
```

We use `shm_open(3)` to create a POSIX shared memory object, so that we later can send the corresponding file descriptor to the compositor so that the latter also has access to it. The flags mean:

- `O_RDWR`: Read-write.
- `O_CREAT`: If the file does not exist, create it.
- `O_EXCL`: Return an error if the shared memory object with this name already exists (we do not want that another running instance of the application gets by mistake the same memory buffer).

We alternatively could use `memfd_create(2)` which spares us from crafting a unique path but this is Linux specific.

We craft a unique, random path to avoid clashes with other running applications.

Right after, we remove the file on the filesystem with `shm_unlink` to not leave any traces when the program finishes. Note that the file descriptor remains valid since our process still has the file open (there is a reference counting mechanism in the kernel behind the scenes).

We then resize with `ftruncate` and memory map this file with `mmap(2)`, effectively allocating memory, with the `MAP_SHARED` flag to allow the compositor to also read this memory.

Later, we will send the file descriptor over the UNIX domain socket as ancillary data to the compositor.

Alright, we now have some memory to draw our frame to, but the compositor does not know of it yet. Let's tackle that now.

Chatting with the compositor

We are going to exchange messages back and forth over the socket with the compositor. Let's use plain old blocking calls in `main` like it's the 70's. We read as much as we can from the socket:

C

```
1   while (1) {
2     char read_buf[4096] = "";
3     int64_t read_bytes = recv(fd, read_buf, sizeof(read_buf), 0);
4     if (read_bytes == -1)
5       exit(errno);
6
7     char *msg = read_buf;
```

```
8     uint64_t msg_len = (uint64_t)read_bytes;
9
10     while (msg_len > 0)
11       wayland_handle_message(fd, &state, &msg, &msg_len);
12     }
13   }
```

The read buffer very likely now contains a sequence of various messages, which we parse and handle with `wayland_handle_message` eagerly until the end of the buffer. This might break if a message is spanning two different read buffers - a ring buffer would be more appropriate to handle this case gracefully, but again, for this article this is fine.

`wayland_handle_message` reads the header part of every message as described in the beginning, and reacts to known opcodes and objects:

C

```
1 static void wayland_handle_message(int fd, state_t *state, char **msg,
2                                    uint64_t *msg_len) {
3   assert(*msg_len >= 8);
4
5   uint32_t object_id = buf_read_u32(msg, msg_len);
6   assert(object_id <= wayland_current_id);
7
8   uint16_t opcode = buf_read_u16(msg, msg_len);
9
10   uint16_t announced_size = buf_read_u16(msg, msg_len);
11   assert(roundup_4(announced_size) <= announced_size);
12
13   uint32_t header_size =
14       sizeof(object_id) + sizeof(opcode) + sizeof(announced_size);
15   assert(announced_size <= header_size + *msg_len);
16
17   if (object_id == state->wl_registry &&
18       opcode == wayland_wl_registry_event_global) {
19       // TODO
20   }
21   // Following: Lots of `if (opcode == ...) {... } else if (opcode =
...) { ... } [...]`
22 }
```

Reacting to events: binding interfaces

At this point we have sent one message to the compositor: `wl_display@1.get_registry()` thanks to our C function `wayland_wl_display_get_registry`. The compositor responds with a series of

events, listing the available global objects, such as shared memory support, extension protocols, etc.

Each event contains the interface name, which is a string. Now, in the Wayland protocol, the string length gets padded to a multiple of four, so we have read those padding bytes as well.

If we see a global object that we are interested in, we create one of this type, and record the new id in our `state` structure for later use. While we're at it, we also handle error events. If the compositor does not like our messages, it will complain with some useful error messages in there:

C

```
1   if (object_id == state->wl_registry &&
2       opcode == wayland_wl_registry_event_global) {
3     uint32_t name = buf_read_u32(msg, msg_len);
4
5     uint32_t interface_len = buf_read_u32(msg, msg_len);
6     uint32_t padded_interface_len = roundup_4(interface_len);
7
8     char interface[512] = "";
9     assert(padded_interface_len <= cstring_len(interface));
10
11     buf_read_n(msg, msg_len, interface, padded_interface_len);
12     // The length includes the NULL terminator.
13     assert(interface[interface_len - 1] == 0);
14
15     uint32_t version = buf_read_u32(msg, msg_len);
16
17     printf("<- wl_registry@%u.global: name=%u interface=%.*s
version=%u\n",
18            state->wl_registry, name, interface_len, interface, version);
19
20     assert(announced_size == sizeof(object_id) + sizeof(announced_size)
+
21                              sizeof(opcode) + sizeof(name) +
22                              sizeof(interface_len) +
padded_interface_len +
23                              sizeof(version));
24
25     char wl_shm_interface[] = "wl_shm";
26     if (strcmp(wl_shm_interface, interface) == 0) {
27       state->wl_shm = wayland_wl_registry_bind(
28           fd, state->wl_registry, name, interface, interface_len,
version);
29     }
30
```

```
31      char xdg_wm_base_interface[] = "xdg_wm_base";
32      if (strcmp(xdg_wm_base_interface, interface) == 0) {
33        state->xdg_wm_base = wayland_wl_registry_bind(
34            fd, state->wl_registry, name, interface, interface_len,
version);
35      }
36
37      char wl_compositor_interface[] = "wl_compositor";
38      if (strcmp(wl_compositor_interface, interface) == 0) {
39        state->wl_compositor = wayland_wl_registry_bind(
40            fd, state->wl_registry, name, interface, interface_len,
version);
41      }
42
43      return;
44    } else if (object_id == wayland_display_object_id && opcode ==
wayland_wl_display_error_event) {
45      uint32_t target_object_id = buf_read_u32(msg, msg_len);
46      uint32_t code = buf_read_u32(msg, msg_len);
47      char error[512] = "";
48      uint32_t error_len = buf_read_u32(msg, msg_len);
49      buf_read_n(msg, msg_len, error, roundup_4(error_len));
50
51      fprintf(stderr, "fatal error: target_object_id=%u code=%u
error=%s\n",
52              target_object_id, code, error);
53      exit(EINVAL);
54    }
```

Remember: Since the Wayland protocol is a kind of RPC, we need to create the objects first before calling remote methods on them.

In terms of robustness, we do not have guarantees that every feature (i.e.: interface) we need in our application will be supported by the compositor. It could be a good idea to bail if the interfaces we require are not present.

Using the interfaces we created

We can now call methods on the new interfaces to create more entities we will need, namely:

- A `wl_surface`
- A `xdg_surface`
- A `xdg_toplevel`

The last two being entities from extension protocols, which is inconsequential in our implementation since we do not link against any libraries. This is just the same logic as the other messages and events from

the core protocol.

Once we have done that, the surface is setup, and we commit it, to signal to the compositor to atomically apply the changes to the surface.

C

```
1     while (msg_len > 0)
2       wayland_handle_message(fd, &state, &msg, &msg_len);
3
4     if (state.wl_compositor != 0 && state.wl_shm != 0 &&
5         state.xdg_wm_base != 0 &&
6         state.wl_surface == 0) { // Bind phase complete, need to create
surface.
7       assert(state.state == STATE_NONE);
8
9       state.wl_surface = wayland_wl_compositor_create_surface(fd,
&state);
10       state.xdg_surface = wayland_xdg_wm_base_get_xdg_surface(fd,
&state);
11       state.xdg_toplevel = wayland_xdg_surface_get_toplevel(fd, &state);
12       wayland_wl_surface_commit(fd, &state);
13     }
14   }
```

Reacting to events: ping/pong

For some entities, the Wayland compositor will send us a ping message and expect a pong back to ensure our application is responsive and not deadlocked or frozen.

We just have to add one more `if` to the long list of `if`s to handle each event from the compositor:

C

```
1 if (object_id == state->xdg_wm_base &&
2             opcode == wayland_xdg_wm_base_event_ping) {
3     uint32_t ping = buf_read_u32(msg, msg_len);
4     printf("<- xdg_wm_base@%u.ping: ping=%u\n", state->xdg_wm_base,
ping);
5     wayland_xdg_wm_base_pong(fd, state, ping);
6
7     return;
8   }
```

Akin to the previous ping/pong mechanism, we receive a `configure` event for the `xdg_surface` and we reply with a `ack_configure` message.

This is an important milestone since from that point on, we can start rendering our frame! We thus advance our little state machine:

C

```
1 if (object_id == state->xdg_surface &&
2             opcode == wayland_xdg_surface_event_configure) {
3     uint32_t configure = buf_read_u32(msg, msg_len);
4     printf("<- xdg_surface@%u.configure: configure=%u\n", state-
>xdg_surface,
5            configure);
6     wayland_xdg_surface_ack_configure(fd, state, configure);
7     state->state = STATE_SURFACE_ACKED_CONFIGURE;
8
9     return;
10   }
```

Once the configure/ack configure step has been completed, we can render a frame.

To do so, we need to create two final entities: a shared memory pool (`wl_shm_pool`) and a `wl_buffer` if they do not exist yet.

Finally, we fiddle with the pixel data anyway we want, remembering the color format we picked (XRGB8888), attach the buffer to the surface, and commit the surface.

This acts as synchronization mechanism between the client and the compositor to avoid presenting a half-rendered frame. To sum up:

1. The `ack_configure` event signals us that we can start rendering the frame
2. We render the frame client-side by setting the pixel data to whatever we want
3. We send the `attach` + `commit` messages to notify the compositor that the frame is ready to be presented
4. We advance our state machine to avoid writing to the frame data while the compositor is presenting it

So let's show a red rectangle as a warm-up. The alpha component is completely ignored as far as I can tell in this color format:

C

```
1     if (state.state == STATE_SURFACE_ACKED_CONFIGURE) {
2       // Render a frame.
3       assert(state.wl_surface != 0);
4       assert(state.xdg_surface != 0);
5       assert(state.xdg_toplevel != 0);
6
7       if (state.wl_shm_pool == 0)
```

```
8         state.wl_shm_pool = wayland_wl_shm_create_pool(fd, &state);
9       if (state.wl_buffer == 0)
10         state.wl_buffer = wayland_wl_shm_pool_create_buffer(fd, &state);
11
12       assert(state.shm_pool_data != 0);
13       assert(state.shm_pool_size != 0);
14
15       uint32_t *pixels = (uint32_t *)state.shm_pool_data;
16       for (uint32_t i = 0; i < state.w * state.h; i++) {
17         uint8_t r = 0xff;
18         uint8_t g = 0;
19         uint8_t b = 0;
20         pixels[i] = (r << 16) | (g << 8) | b;
21       }
22       wayland_wl_surface_attach(fd, &state);
23       wayland_wl_surface_commit(fd, &state);
24
25       state.state = STATE_SURFACE_ATTACHED;
26     }
```

Result:

Let's render something more interesting. We download the Wayland logo, but we do not want to have to deal with a complicated format like PNG (because we then have to uncompress the image data with `zlib` or similar).

We thus convert it offline to a simpler image format, PPM6, and then embed the raw pixel data in our code as a byte array, skipping over the first 15 bytes which are metadata:

Shell

```
$ file wayland.png
wayland.png: PNG image data, 117 x 150, 8-bit/color RGBA, non-interlaced
$ convert wayland.png wayland.ppm
$ file wayland.ppm
wayland.ppm: Netpbm image data, size = 117 x 150, rawbits, pixmap
$ xxd -s +15 -i wayland.ppm  > wayland-logo.h
$ sed -i 's/wayland_ppm/wayland_logo/g' wayland-logo.h
```

The resulting C array created by `xxd` will be named after the input file i.e. `wayland_ppm`. We rename it with the last command to something more human-readable.

The image is now in the `RGB` format (3 bytes per pixel), which we have to convert to the `XRGB` format (4 bytes per pixel). Our frame rendering loop becomes:

C

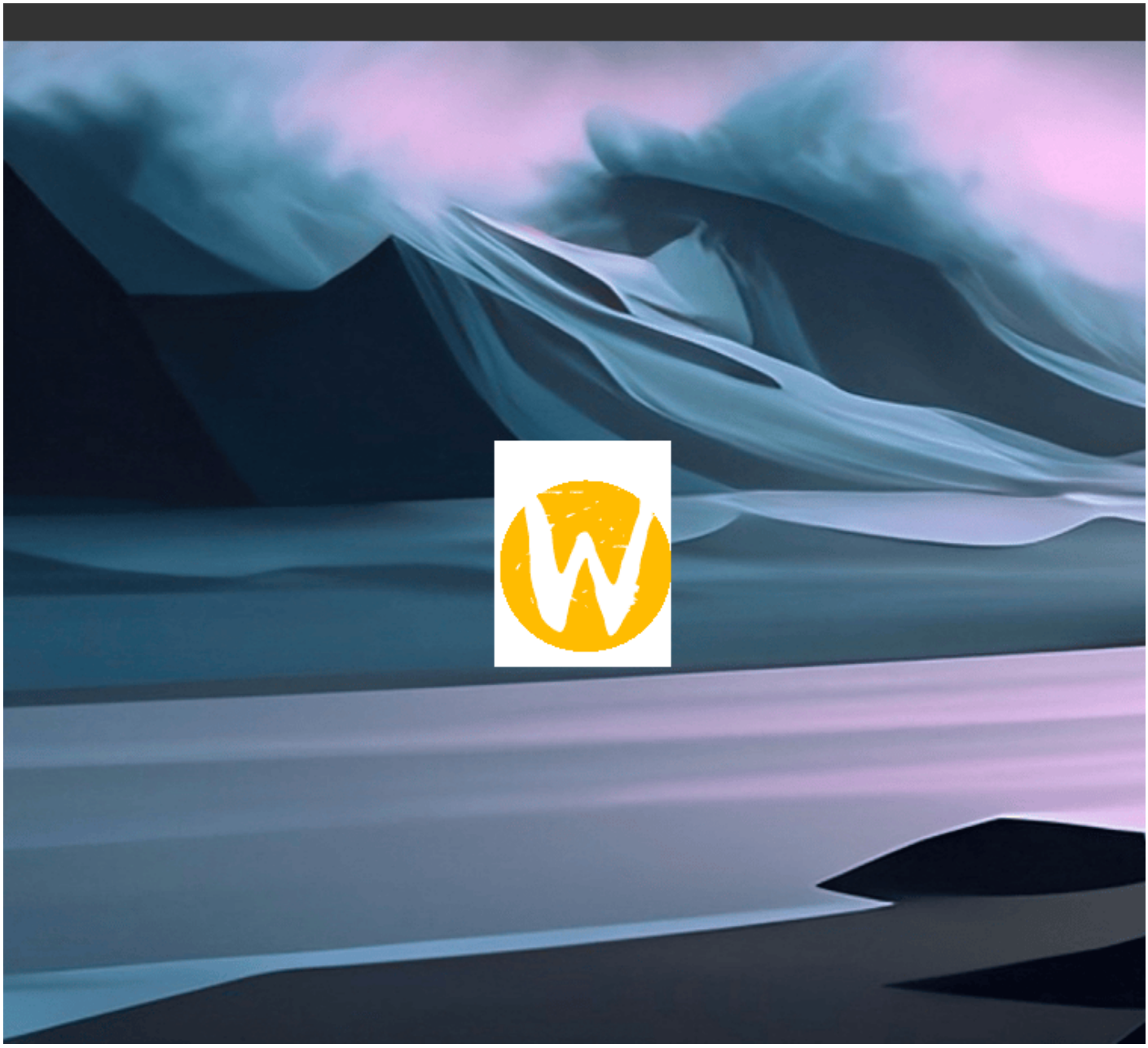
```
#include "wayland-logo.h"

[...]

      for (uint32_t i = 0; i < state.w * state.h; i++) {
        uint8_t r = wayland_logo[i * 3 + 0];
        uint8_t g = wayland_logo[i * 3 + 1];
        uint8_t b = wayland_logo[i * 3 + 2];
        pixels[i] = (r << 16) | (g << 8) | b;
       }
```

And finally we see the result.

Tiled:



Floating:

Note: We handle the absolute minimum set of events coming from the compositor to make it work in a simple way. If your particular compositor sends more events, they will have to be read (and possibly ignored). Since the Wayland protocol uses a Tag-Length-Value (TLV) encoding, one can simply skip over `<length>` bytes if the opcode is unknown. But some events will demand a reply (e.g. ping/pong)!

The end

It was not that much work to go from zero to a working GUI application, albeit a simplistic one.

Compared to X11, it was a bit more work, but not that much. The barrier of entry is higher but the concepts and architecture are more sound, it seems to me.

The setup is a bit tedious but once this is done, we are in practice going to spend all of our time in the frame rendering code, and perhaps add support for a few additional events (we do not yet support keyboard or mouse events, for example, or animations, which would require us to notify the compositor that a region was 'damaged' meaning modified, and needs re-rendering).

Thus, I have the feeling that Wayland really goes out of the way once the initial scaffolding is done.

As for the next steps, I would like to draw some text, and react to user input events. Maybe even port something like microui, which only needs a few drawing routines, to our application.

Addendum: the full code

Do not forget to generate `wayland-logo.h` with the aforementioned commands!

Compile with: `cc -std=c99 wayland.c -Ofast`.

▶ The full code

Back to all articles