Hydra development notes

1 Minimal compositor

We start with the bare minimum to get a compositor up and running with Mir. This was based on the example code here:

```
#include <miral/runner.h>
#include <miral/x11_support.h>
#include <miral/minimal_window_manager.h>

using namespace miral;

int main(int argc, char const* argv[]) {
  MirRunner runner{argc, argv};

  return runner.run_with({
      X11Support{},
      set_window_management_policy<MinimalWindowManager>()
    });
}
```

After referencing the more in-depth example, I added `X11Support{}` to the options.

1.1 Building

A basic cmake build script based on the example script:

```
include(FindPkgConfig)
pkg_check_modules(miral REQUIRED miral IMPORTED_TARGET)

add_executable(hydra main.cpp)
target_link_libraries(hydra PRIVATE PkgConfig::miral)
```

1.1.1 .deb builds

For consistency, .deb builds are done in a Docker container. The actual .deb build step is generated using CPack:

```
install(TARGETS hydra RUNTIME DESTINATION bin)

set(CPACK_PACKAGE_CONTACT jordan@carbondoes.dev)
set(CPACK_DEBIAN_PACKAGE_DEPENDS
  "libmiral5t64"
```

```
  "mir-graphics-drivers-desktop")
list(JOIN CPACK_DEBIAN_PACKAGE_DEPENDS "," CPACK_DEBIAN_PACKAGE_DEPENDS)

include(CPack)
```

1.2 Launching applications

At this point, the compositor launches, but there is no way (convenient) way to launch an application. Apps can be launched manually, though:

```
export WAYLAND_HOST=99
hydra
sleep 30
xfce4-terminal
```

2 Shell

I decided to work on an application launcher idea I've had for a while: a recreation of Emacs' Hydra. This provides 2 main features:

- Instead of using key combinations where you hit all the keys at once, a string of keys is entered, starting with a leader key that brings up the hydra.
- Commands are organized into a trie, where each node of the trie is visually presented as a table of options.
- Other UI elements (e.g. fuzzy search) can also be displayed to allow for selection from lists

2.1 Creating a Window

I chose to use Dear imgui to build the widgets. imgui needs a window backend to function, which handles creation of the window surface, keyboard events, etc.

2.1.1 Reusing an existing `wl_display` connection

The windowing backend needs to support using an existing display connection in order for the shell to act as an internal client.

(Initially I searched for a way to do this with GLFW, but there are only functions to get the `wl_display` GLFW creates, rather than passing in our own.)

SDL3 has improved wayland support and does support creating an `SDL_Window` with an existing `wl_display`:

```
SDLContext::SDLContext(wl_display* display) {
  if(display) {
```

```
    SDL_SetPointerProperty(SDL_GetGlobalProperties(),
    ↪  SDL_PROP_GLOBAL_VIDEO_WAYLAND_WL_DISPLAY_POINTER, display);
  }
```

Because SDL3 is not yet officially packaged, I chose to bundle it with the project and statically link it:

```
set(SDL_STATIC ON CACHE BOOL "Build SDL static target" FORCE)
set(SDL_SHARED OFF CACHE BOOL "Don't build SDL shared target" FORCE)
FetchContent_DeclareEx(
  SDL3
  GIT_REPOSITORY https://github.com/libsdl-org/SDL.git
  GIT_TAG        eb3fc0684c9957de108df5b338307418b0130c3b
  OVERRIDE_FIND_PACKAGE
)
```

While SDL3 supports using an existing `wl_display`, but this had a few quirks:

- This bug I reported where the `wl_display` was closed during initialization, even though if it was not created/owned by SDL. Initially I patched this locally, but they fixed it very quickly upstream.
- If the `WAYLAND_DISPLAY` environment variable is not set, init fails, even though we already have a `wl_display`. Setting it to empty string if it's not set is enough:

  ```
  // We know we have a display, so avoid failing Init() if the env is not set
  SDL_SetEnvironmentVariable(SDL_GetEnvironment(), "WAYLAND_DISPLAY", "",
  ↪  false);
  ```

2.1.2 RAII wrappers

SDL is a C library, and expects you to call `SDL_DestroyWindow()`. For this, and a couple other things, I wrote simple RAII wrappers that call the corresponding destroy function as the object exits scope. For simplicity, most of these wrappers are unmovable / uncopyable.

2.1.3 Custom window role with wlr-layer-shell

At this point, making a normal window with `SDL_CreateWindow()` is simple. However, this creates a normal floating window, but we want to make an overlay.

Referencing Waybar, I found that it uses the `wlr-layer-shell` extension to configure the overlay.

Out of the box SDL creates an `xdg-toplevel-surface` for its windows, unless you set `SDL_PROP_WINDOW_CREATE_WAYLAND_SURFACE_ROLE_CUSTOM_BOOLEAN`, in which case it will create a `wl_surface` without a role.

From the SDL3 wayland README, getting the `wl_surface` from the `SDL_Window` is simple:

```
auto* surface = static_cast<wl_surface*>(SDL_GetPointerProperty(window_props,
  SDL_PROP_WINDOW_WAYLAND_SURFACE_POINTER, nullptr));
```

Setting up the `wlr-layer-surface` role is a little more involved:

- I attempted to use wayland-scanner to generate interface code, but this had some conflicts with C++ keywords, so I ended up using hyperwayland-scanner. Referencing KDE's `ecm_add_wayland_client_protocol`, I wrote my a small cmake wrapper:

```
function(add_wayland_client_protocol target)
  set(options "")
  set(multiValueArgs PROTOCOLS)
  set(oneValueArgs "")
  cmake_parse_arguments(PARSE_ARGV 1 arg "${options}" "${oneValueArgs}"
  ↪  "${multiValueArgs}")

  foreach(protocol ${arg_PROTOCOLS})
    get_filename_component(_protocol ${protocol} ABSOLUTE)
    cmake_path(GET _protocol STEM _protocol_name)

    set(_outputs
      "${CMAKE_CURRENT_BINARY_DIR}/${_protocol_name}.hpp"
      "${CMAKE_CURRENT_BINARY_DIR}/${_protocol_name}.cpp")
    add_custom_command(
      OUTPUT  ${_outputs}
      COMMAND $<TARGET_FILE:hyprwayland-scanner> --client ${_protocol}
      ↪  ${CMAKE_CURRENT_BINARY_DIR}
      DEPENDS ${_protocol} VERBATIM
    )
    target_sources(${target} PUBLIC ${_outputs})
  endforeach()
endfunction()
```

- The `wl-layer-shell` proxy is obtained from the registry. I was intially skeptical that the registry listener callback was guaranteed to be called after one `wl_display_roundtrip`, but SDL appears to do the same thing. This makes it easier to write a wrapper that fetches a single global from the registry:

```
RegistryListener(struct wl_display* display, wl_interface const* target, int
↪  version)
  : target(target), version(version)
{
  auto* registry = wl_display_get_registry(display);
  wl_registry_add_listener(registry, &global_listener, this);
  wl_display_roundtrip(display);
  wl_registry_destroy(registry);
```

```
}

// other boiler plate
```

- After we get the `LayerShell` we use `get_layer_surface(wl_surface*, ...)` to get a `std::weak_ptr<LayerSurface>` for our `wl_surface`. I chose to have the `LayerShell` own the `LayerSurface`'s, so all of the surfaces are destroyed before the shell object.
- There is some additonal configuration for the surface (anchors, exclusive_zone, etc.)

I wrapped all of this up in a `LayerWindow` class extending the basic `Window` class.

2.1.4 Reacting to fullscreen clients

If another client enters fullscreen, we don't want to draw the shell on top of it (unless we are currently interacting with it).

(Initially, I considered doing this in a WindowManager, in `handle_modify_window()`. Whenever a client enters fullscreen, it should generate an event. However, when I tested this I wasn't getting any events. It turns out this was a bug in Mir, only affecting Xwayland clients, which I tracked down later.)

Referencing Waybar again, I found that they used `wlr-foreign-toplevel`, which provides information directly to clients about modifications to other surfaces, including fullscreen state.

I wrote a simple wrapper around it, counting the number of fullscreen toplevels as they are created/destroyed/modified. This is then used in `LayerWindow::should_hide() -> bool`, return true if the count is positive.

2.2 Widgets

The Shell UI is broken down into a couple different widgets, described by a couple concepts in `widget.h`

- IsDrawable: any object defining `should_draw() -> bool` and `draw()`
  - StatusLine: Always visible bar designed to show the window title, date, and error/status messages.
  - IsPrompt: drawable with `handle_key(Key)` and `try_result() -> optional<value>`
    - TablePrompt: A single page of commands
    - SearchPrompt: A list of options with fuzzy search input

These widgets are designed to be drawn within a `FrameContext`, an object managing the lifetime of each frame, essentially:

1. `ImGui::NewFrame()`
2. Widgets drawn here (generates imgui draw data)
3. `ImGui::EndFrame()`

4. Render the imgui draw data
5. Swap buffers

To facilitate this, `FrameContext::start_frame()`, does (1), and returns a guard object, which does (3), (4), and (5) on destruction. Optionally, a frame can be marked hidden, skipping (4).

The `TablePrompt` widget is relatively simple, but the `SearchPrompt` needs to filter the list of options based on the inputted text, so I used rapidfuzz.

2.3 Run loop

```
void Shell::run(Window& window, Callback& cb) {
  self->is_done = false;

  FrameContext fc(&window);
  while(!done()) {
    if(auto res = self->frame(window, fc)) {
      cb(res.value());
    }
  }
}
```

The main run loop of the Shell takes a `Window`, and a `Callback`:

- The `Window` is not owned by the `Shell` to maintain separation from the window configuration, and I wanted to avoid having to make `Window` moveable/copyable
- A `Callback` is a type-erased lambda where prompt results are returned. Displaying new prompts and other modifications to the `Shell` should be done through this callback for thread-safety reasons, except a few functions that have their own locks (e.g. `handle_key`).

  I chose not to use `std::function` primarily for ownership reasons: I did not want to limit the callbacks to copyable lambdas.

1. Each frame starts by handling window events, breaking early if there is a result:

   ```
   // Self::frame()
   fc.handle_events([this](const SDL_Event& e){
     handle_event(e);
   });

   if(auto res = pop_result()) {
     return res;
   }
   ```

2. Then the widgets are drawn:

```
// Self::frame():
auto frame_guard = fc.start_frame();

// Self::draw():
// ... imgui positioning code ...
status.draw();

if(cur_prompt.has_value() && should_draw(*cur_prompt)) {
  // ... imgui positioning code ...
  ::hydra::shell::draw(cur_prompt.value());
}
```

3 Integrating the shell

3.1 Starting an internal client

The Shell state management (starting the internal client, providing a callback, managing the lifetime of the Shell) is wrapped up in the `ShellLauncher` class.

From the mir documentation, internal clients provide 2 callbacks, one accepting the session pointer (the application), and one accepting the `wl_display` pointer.

The callback accepting the `wl_display` will run on its own thread, so this where the Shell's main loop will run:

```
void ShellLauncher::operator()(struct wl_display* display) {
  pthread_setname_np(pthread_self(), "HydraShell");

  // wait for the session to be set
  {
    std::unique_lock lock{session_lock};
    startup_cv.wait(lock, [this]{ return !weak_session.expired(); });
  }

  using namespace hydra::shell;

  SDLContext ctx(display);
  LayerWindow window(ctx, Window::Properties::FromConfig());

  auto cb = Shell::Callback::Create([](auto){ /* do nothing */ });
  shell.run(window, cb);
}
```

We also need to enable the required wayland extensions first:

```
void ShellLauncher::enable_extensions(miral::WaylandExtensions& extensions) {
  for(auto ext: std::array {
      miral::WaylandExtensions::zwlr_layer_shell_v1,
      miral::WaylandExtensions::zwlr_foreign_toplevel_manager_v1,
    }) {
    extensions.conditionally_enable(ext,
    ↪  [this](miral::WaylandExtensions::EnableInfo const& info) {
      if(auto session = weak_session.lock()) {
        return session == info.app();
      }

      return false;
    });
  }
}
```

`ShellLauncher` isn't copyable, so we wrap it in a lambda before passing it to the runner:

```
auto internal_client() {
  return [this](auto&& args) {
    operator()(std::forward<decltype(args)>(args));
  };
}
```

3.2 StateMachine

At this point the shell is drawn to the screen as we expect, but doesn't accept input. In order to simplify writing a `Callback`, I wrote a derived class: `StateMachine`.

```
template <typename Input>
template <typename T, State... states>
auto StateMachine<Input>::Create(auto&&... args) ->
↪  std::shared_ptr<StateMachine>;
```

- `T...` is a base class, defining the states
- `states...` represents a static map from an index (`std::size_t`) a state member function (`auto(T::*)()`)
- Upon jumping to a new state, the corresponding member function is called and the result becomes the new current state
- When the callback is called, the input is passed to the current state, which returns a new state to jump to, or -1 for no-op.

This allows us to define some basic interaction:

```
enum State {
  IDLE,
  COMMAND,
};

auto ShellLauncher::idle() {
  return [](auto) -> std::size_t {
    throw std::runtime_error("Unreachable");
  };
}

auto ShellLauncher::command() {
  enum Commands: Option::value_t {
    QUIT,
  };

  shell.show(hydra::Table{
      std::pair{hydra::Key::Keycode(SDLK_Q),
       ↪  hydra::Option{std::size_t(Commands::QUIT), "Quit"}},
    });

  return [this](auto res) -> std::size_t {
    // ... error handling ...

    switch(Commands(res)) {
      case Commands::QUIT:
        runner->stop();
        break;
      default:
        break;
    }

    return State::IDLE;
  };
}

ShellLauncher::ShellLauncher(MirRunner* runner)
  : runner(runner) {
  : runner(runner),
    state_machine(StateMachine::Create<
                  ShellLauncher,
                  hydra::util::State { State::IDLE, &ShellLauncher::idle },
                  hydra::util::State { State::COMMAND, &ShellLauncher::command }
                  >(this)) { /* ... */ }

 bool ShellLauncher::show_commands() {
   bool ret = false;
```

```
  state_machine->lock([&ret](std::size_t state) -> std::size_t {
    if(state == State::IDLE) {
      ret = true;
      return State::COMMAND;
    }

    return -1;
  });

  return ret;
}
```

Then we can call `show_commands()` if we receive the leader key to bring up the `command()` state and display the table.

3.3 Launching applications

Launching applications is done using the `miral::ExternalClientLauncher`, but it doesn't appear to provide a mechanism for reading desktop entries to get a list of applications.

Initially, I looked to rofi for an example, but they appeared to be parsing the desktop entries manually. I also looked at using gnome-menus but it doesn't really appear to be designed to be used as a standalone library. I briefly considered writing my own parser.

Eventually I stumbled into Gio (part of Gtk's Glib) which provides a very clean interface for this. With Giomm (the C++ wrapper for Gio), getting a list of applications is a one liner:

```
std::vector<Glib::RefPtr<Gio::AppInfo>> apps = Gio::AppInfo::get_all();
```

After this, launching an application from the list is simple:

```
auto cmds = ExternalClientLauncher::split_command(selected->get_commandline());
this->launcher->launch(cmds);
```

3.4 Window Management

`WindowManager` primarily handles listing windows, getting the focused window, and switching focus.

I primarily referenced egmde's WindowManagerPolicy for the basic setup. Every new window gets `WindowManager::Metadata` as userdata. The `Metadata` provides a public interface for a window, without having to worry about locks:

- Clients can iterate over the list of windows with `WindowManager::locked_list_windows` which provides a `std::weak_ptr<Metadata>`.
- Other locked functions accept `std::shared_ptr<Metadata>` as input, using the stored id to lookup the `WindowInfo`

There are only 2 workspaces at the moment. One for the shell, and another default workspace. Each workspace has an associated `WorkspaceInfo`.

3.4.1 `WorkspaceInfo`

`WorkspaceInfo` provides very similar functionality to `miral::ApplicationSelector` (internal). It tracks the list of windows in the workspace, in most-recently focused first order.

In order to make `WorkspaceInfo` easier to test alone, `WorkspaceInfoImpl` is templated with the `Workspace` and `Window` type, making mocking easy.

When the shell has exclusive focus, it is not possible to switch to a window in the default workspace. To be able to focus windows from shell commands,

3.4.2 Inputs, synchronization, and thread-safety

There are 2 different threads interacting with the Shell:

- The main shell thread, in `Shell::run()`
- Compositor keyboard events moving the Shell from `State::IDLE` to `State::COMMAND`

We guard mutation of the state with a lock managed by the `StateMachine`. It also provides a function `StateMachine::lock(functor)` to run a function under lock and potentially change the state.

Keyboard events are passed to the `Shell` whenever it is in focus. Pressing the leader key to bring up the command tree, bringing the `Shell` into focus, but it will then also receive the leader key event. To avoid this, keys are instead passed to the `Shell` with `Shell::handle_key(key, timestamp)` which adds the keys to a buffer:

```
void handle_key(Key key, uint64_t timestamp_ns) {
  std::lock_guard g{buffer_lock};

  const auto timeout = Config::Get().buffer_timeout<std::chrono::nanoseconds>();
  while(buffer.size() && timestamp_ns - buffer.front().second > timeout) {
    buffer.pop();
  }

  buffer.push({key, timestamp_ns});
  buffer_cv.notify_all();
}
```

3.5 Command Tree

We don't just want to display one page of commands, but a trie of commands, where each node is either a page or a command to execute, and the edges are keys.

`template <typneame Key, typename Value> struct Trie` holds this information.

- `template <typename trie_ptr_t> Trie::base_node_t` is a handle to a single node:
  - The trie_ptr_t is either `Trie*` or `Trie const*`. Const-correctness is simplified with deducing this:

```
auto& self(this base_node_t self) { return self.parent->nodes[self.idx];
↪  }
```

`self()` returns in `Trie const&` when called from const members or in `base_node_t<Trie const*>`, or `Trie&` otherwise. This means the non-const members of `base_node_t` cannot be called from `base_node_t<Trie const*>`.

- `root()` and `croot()` return mutable and const handles to the root node.
- Node values are default constructed, and can be accessed with the dereference operators
- Can be constructed from a `std::tuple` of values

With one `Trie`, we can list all our commands:

```
static const hydra::util::Trie<Key, Option> tree {
  std::tuple{Key::Keycode(SDLK_Q), opt("Quit/logout session"),
    std::tuple{Key::Keycode(SDLK_Q), opt("Quit", QUIT)}},
  std::tuple{Key::Keycode(SDLK_SPACE), opt("Launch application", LAUNCH)},
  std::tuple{Key::Keycode(SDLK_W), opt("Windows"),
    std::tuple{Key::Keycode(SDLK_W), opt("Find window", WINDOW_FIND)},
    std::tuple{Key::Keycode(SDLK_N), opt("Next window", WINDOW_NEXT)},
    std::tuple{Key::Keycode(SDLK_P), opt("Prev window", WINDOW_PREV)},
    std::tuple{Key::Keycode(SDLK_D), opt("Close window", WINDOW_CLOSE)}}
};
```

We can use a mutable lambda to keep track of the current node

```
return [this,cur=Commands::tree.croot(),status](auto res) mutable -> std::size_t
↪  {
  // ... error handling ...
  if(auto next = cur.try_get(key)) {
    cur = *next;
    switch(cur->value) {
      case Commands::QUIT:
        this->runner->stop();
        return States::IDLE;
      // ... handle other commands ...
      case Commands::NONE:
        show_node(&shell, *cur);
        return -1;
    }
  }
}
```