



```
> emoji: zap |
```

emojig

Chronology of a Zig Endeavour

Six weeks, one binary, zero daemons –
a TUI emoji picker built week by week

Uwe Jugel · Ubunatic Coding

Zig v0.16.0 · 2026 Edition

Emojig: Chronology of a Zig Endeavour

This is the story of six weeks across the late spring and early summer of 2026 (May 29 – July 5), told week by week, commit by commit: how an emoji picker – the most unserious of programs – became a serious exercise in raw terminal engineering, spec-driven architecture, and agentic development. It is a chronicle, not a tutorial: the numbers are real, the dead ends are kept in, and three side projects are born along the way because the tooling we needed did not exist yet.

In a hurry? The [Recipe Index](#) maps every technique in this book to the week that paid for it, so you can jump straight to what you need without reading a chronology. Starting a TUI of your own? Skip to [Starting a New Zig TUI](#).

The Why: why emojig + Zig?

Every desktop environment has an emoji picker, but for terminal power users on Linux, the options were historically frustrating:

1. **Bloated electron apps or heavy graphical utilities** that take seconds to load and consume hundreds of megabytes of RAM.
2. **Constantly running resident background daemons** eating CPU cycles just to wait for a hotkey.
3. **Basic CLI tools** that dump text lists but lack real-time 2D keyboard navigation or hover/click mouse support inside raw terminals.

`emojig` was born out of a desire for a **radical, minimal, instant-popup emoji picker** that functions seamlessly as an inline TUI or a floating GUI window.

To achieve this, we selected **Zig**:

- **Musl Static Binaries:** Compiles into a single, zero-dependency static binary under 900 KB.
- **Absolute Memory Control:** Zero runtime allocations on search loops, keeping memory usage strictly under **5 MB RSS** (self-reported via `/proc/self/statm`).

- **Compile-Time Packing:** Leveraging Zig's `@embedFile` to serialize and embed 2,249 emojis directly into the binary, achieving zero runtime disk reads.

Everything the picker knows that is not code – strings, layout, colours, key bindings – lives in spec files. Week 2 moved them out of Zig into JSON; Week 5 migrated that JSON to YAML for comments and cleaner diffs. **The finished project ships YAML specs**, generated down to JSON at build time and embedded at compile time. When Weeks 2 through 4 say “JSON spec”, read it as the format of the day: the architecture is the point, and it did not change.

Here is `emojig` executing in floating GUI mode and inline TUI mode:

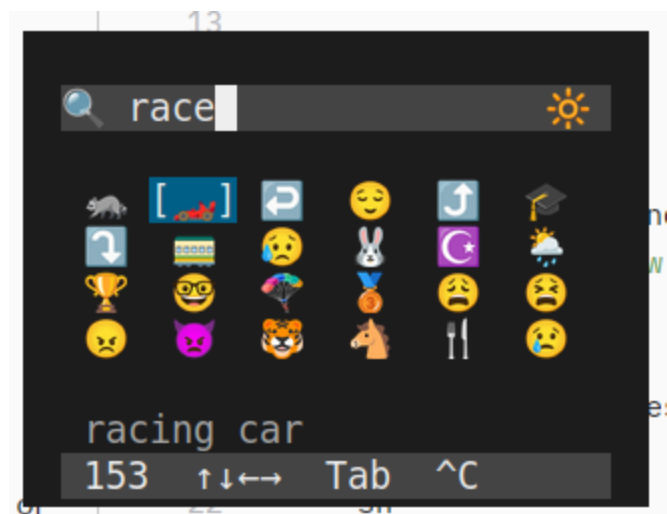


Figure 1: GUI Popup Mode over a dark Zed session.

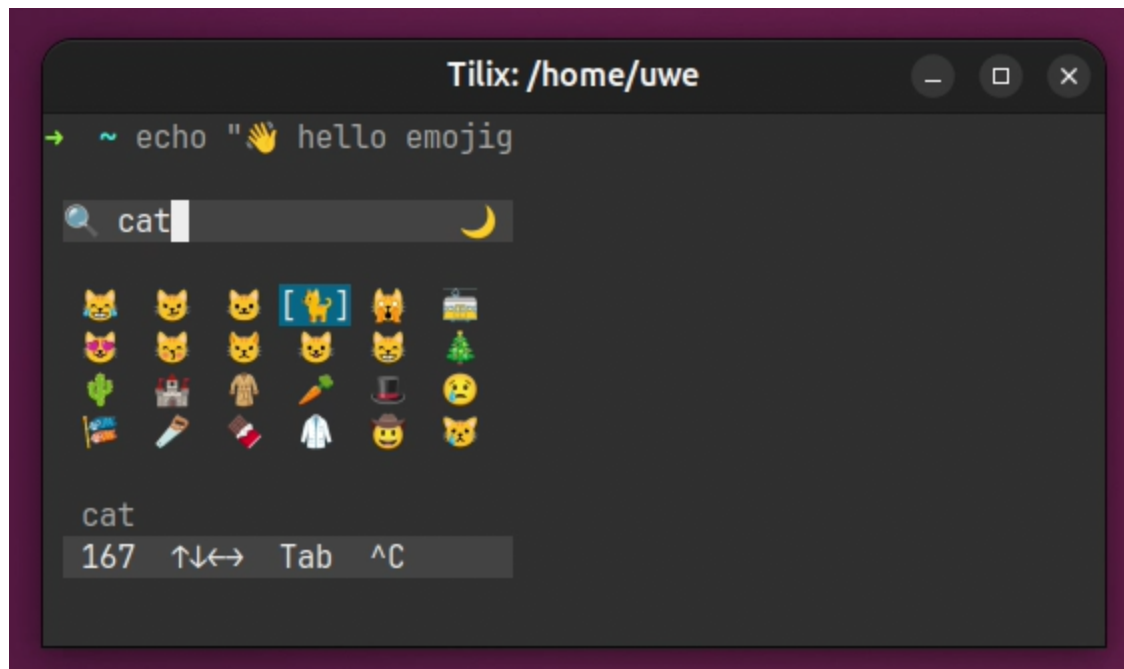
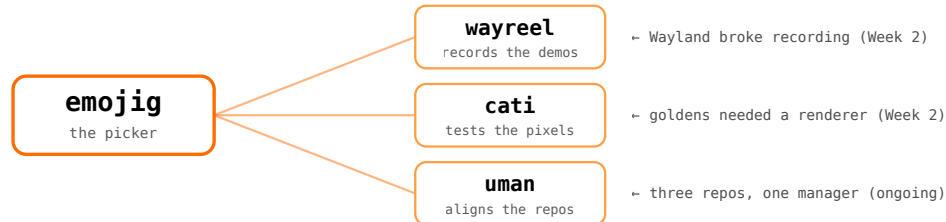


Figure 2: Inline TUI Mode inside a Tilix terminal.

The Ubunatic Ecosystem: Side Projects

As `emojig` evolved, standard development tools proved insufficient for verifying low-level terminal alignments and visual render pipelines. This led to the creation of three companion side projects – each born the week a wall was hit, each kept alive ever since:



Necessity-driven spawning: every side project marks a week when the ecosystem outgrew its tools.

1. Wayreel (The Recorder)

We built **Wayreel** first because we wanted a cool website and didn't want to constantly re-record demos. Its main purpose is to express app usage as scripts (`.reel` files) and play them in a terminal (by accident, it can also command other apps now). Later, it showed it could be used for testing because we could spot visual bugs in the recordings.

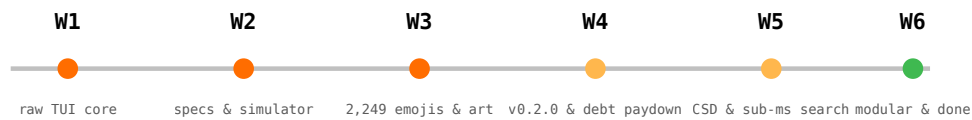
2. Cati (The Visual Tester)

We needed a way to test visual render outputs and golden-file regressions of terminal pixel art (like our winking about-screen animation). We created **Cati** (`cat` for images) which renders PNG/JPEG/SVG files directly to the terminal using Unicode halfblocks, quadblocks, sextants, and sparklines. Cati's aspect ratio fitting math and golden-file verification loop became the standard for our codebases.

3. Uman (The Orchestrator)

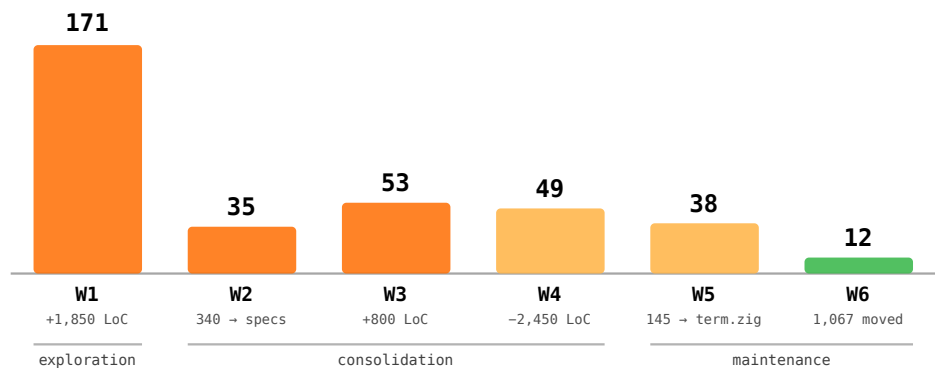
Managing separate repos, git worktrees, and schemas across `emojig`, `cati`, and `books` became a maintenance burden. We built **Uman** (Uman Manages Apps Nicely) – a Go CLI for managing a whole directory of local git projects: bulk status, guarded push, releases, and syncing project websites into one publishing repo.

Chronology Index



Six weeks: from first raw-mode escape code to a finished, self-contained tool.

Reading the Curve



Commits per week (bars) with each week's headline line-count movement. The line-count figures are the ones each chapter documents, not a uniform diff metric.

Week 1 produced 171 commits – very nearly as many as the other five weeks put together (187). That is not productivity; it is **exploration**. Nothing was known yet, so every commit was a probe: does this termios flag matter, does this terminal honour that escape code, does the teardown leave a blank line. Commits were cheap because each one was small and reversible, and that is exactly what you want when the problem is still being discovered.

The rate then collapses by a factor of five and stays flat for three weeks (35, 53, 49). This is **consolidation**: the shape of the program is settled, so work arrives in larger, riskier units – a spec migration, a virtualized viewport, a release. Fewer commits, more thought per commit. The single most valuable commit in the whole book falls in this stretch, and it is the one that removed 2,450 lines.

By Week 6 the rate is 12 across three days. **Maintenance** looks like failure on a contribution graph and is actually the goal: the codebase had stopped needing to change, and the remaining work was making it navigable for whoever came next. A curve that stays at 171 for six weeks is not a healthier project – it is one that never converged.

Two caveats before you compare this to your own graph. These are *agent-sized* commits: an agentic session commits at the granularity of a verified step, not a human's end-of-day, so the absolute height of the first bar means little. And Week 6 covers three days, not seven – the floor is real, but it is not a like-for-like week. The **shape** – steep, then flat, then low – is the part that transfers.

Check it yourself. Every bar above is one command against the repository, so these numbers are falsifiable rather than merely asserted:

```
git rev-list --count --since=2026-05-29T00:00:00+02:00 \
--until=2026-06-05T00:00:00+02:00 HEAD # Week 1 →
171
```

Counted on `main`, half-open windows `[start, end)`, matching each chapter's stated date range. Give git an explicit offset: with a bare `--since=2026-05-29` it resolves the boundary in UTC and quietly moves commits near midnight into the neighbouring week. The six windows sum to 358; the repository holds 359, the one stray being a maintenance commit in August, after the chronicle closes.

The following chapters chronicle each week of this Zig endeavour:

- [Week 1: Foundations & Shell widgets](#)
- [Week 2: Simulator & Spec Decoupling](#)
- [Week 3: Virtualization & PNG Art Compilers](#)
- [Week 4: Search Optimization & Command Interception](#)
- [Week 5: GNOME CSD Rendering & Binary Search](#)
- [Week 6: Codebase Modularization & Key Merging](#)

And two chapters that are not chronology at all:

- [The Recipe Index](#) – every technique, by problem instead of by date.
- [Starting a New Zig TUI](#) – the bootstrap kit, extracted from all six weeks.

The Recipe Index

This chronicle is organised by date, which is the one axis nobody ever queries. This chapter is the other axis.

Every row below is a problem somebody actually hit while building `emojig`, the week that paid for it, and – where one exists – the rule it hardened into over in the field manual. This is a **jump table, not a cookbook**: it deliberately contains no code. A second copy of the technique would drift away from the week that owns it, which is precisely the failure Week 4 spent a day deleting. Follow the link, read the week, take the code from there.

Blank cells in the right-hand column are informative. They mark techniques this project learned and the flagship never captured – candidates for the field manual, not for duplication here.

I need to...

I need to...	Lived in	Rule in the flagship
enter raw mode without wrecking the shell	Week 1	TUI Restoration §1
survive a panic while the terminal is in raw mode	Week 1	TUI Restoration §1
keep <code>stdout</code> clean so a shell widget can read it	Week 1	–
anchor an inline TUI to the row it launched on	Week 1	TUI Restoration §7
survive a terminal resize without warping the layout	Week 1	–
avoid the <code>?1049l</code> cursor warp on VTE terminals	Week 1	TUI Restoration §2
move hardcoded strings, colours and layout into specs	Week 2	Agentic Dev Patterns §2
add a UI language without touching the code	Week 2	Agentic Dev Patterns §2
record a Wayland popup for a demo or a doc	Week 2	–

I need to...	Lived in	Rule in the flagship
golden-test what a terminal actually painted	Week 2	The Golden-Change Playbook
render more rows than the terminal has	Week 3	—
compile PNG art into terminal block characters	Week 3	—
stop a per-pixel warning from firing 2,000 times	Week 3	—
fit a grid to the terminal width without gutters	Week 3	TUI Restoration §8
stay responsive while the glyph rasterizer lags	Week 4	—
stop split mouse events from typing into the search box	Week 4	TUI Restoration §4
intercept <code>:q</code> before the search engine sees it	Week 4	—
kill a duplicate implementation of your own program	Week 4	Manifesto §5
pick readable title text on any desktop theme	Week 5	—
draw a full-width row without erasing its last cell	Week 5	TUI Restoration §5
get keystroke-to-results latency under 1 ms	Week 5	—
move a spec format from JSON to YAML mid-project	Week 5	Agentic Dev Patterns §2
break up a 4,800-line source file	Week 6	Agentic Dev Patterns §3
dispatch keys without a growing <code>if-else</code> chain	Week 6	—
verify a TUI from a script, with no human eyes	Week 2	TUI Restoration §6
start a brand-new Zig TUI from nothing	Appendix	Canary-First Development

Reading paths

Three ways through this book that are not “front to back”:

- **I am building a terminal UI.** Week 1 (teardown, `/dev/tty`), Week 3 (virtualization), Week 5 (contrast, latency), then the [Appendix](#) as a checklist.
- **I am organising an agentic project.** Week 2 (specs), Week 4 (deleting the clone), Week 6 (decomposition) – and skip the rendering weeks entirely.
- **I want the failures.** Week 1 (the `?1049l` cursor warp), Week 4 (the split SGR read), Week 5 (the pending-wrap erase). Three bugs, told as they were lived.

Week 1: Foundations & Shell Widgets

Theme – Taking the terminal hostage safely: raw mode, teardown, and the `/dev/tty` split.

Skip this if – you already have a TUI that exits cleanly on every path, including panics.

Takeaways

- **Copy this:** write the restore sequence *before* the first feature, and call it from normal exit, every signal handler, and the panic handler – one function, four call sites.
- **Avoid this:** rendering to `stdout`. The moment you want a shell widget, `stdout` has to be a data channel; render to `/dev/tty` from day one and you never pay for the split.
- **The check that caught it:** launching from a real VTE terminal, not just `foot`. The alt-screen cursor warp below is invisible on the terminal we developed in.

The Genesis of the TUI Core

Every terminal program begins the same way: with a single escape code that takes the terminal hostage. Get raw mode right and nobody notices; get it wrong and the user's shell is wrecked. Week 1 was entirely about earning that trust – before a single emoji was ever drawn.

During Week 1 (May 29 – June 4, 2026), the `emojig` repository was initialized. The core objective was to build a zero-dependency, ultra-lightweight, and lightning-fast emoji fuzzy search tool in Zig.

Week 1 by the Numbers

- **Total Commits:** 171 – very nearly as many as the remaining five weeks combined (187). See [Reading the Curve](#) for why that is a property of exploration rather than of productivity.

- **Codebase Expansion:** From 0 lines of code to 1,250 LoC in Zig, 180 LoC in Go (pack script), and 420 LoC in shell integrations.
-

Daily Chronology

May 29, 2026: The Initial Prototype (cd0029a)

The project began by writing the raw terminal interface and the fuzzy subsequence search engine.

- **Features Added:** Fuzzy search matcher, embedded database loader, and raw termios attributes hook.
- **Lines of Code (LoC) Added:**
 - `src/main.zig`: 204 LoC.
 - `src/root.zig`: 134 LoC.
 - `tools/pack_emojis.py`: 82 LoC.

Code Archaeology: Initial Termios & Rendering Loop

The very first commit established the raw mode, signal safety, and the double-buffered rendering surface. Here is how standard terminal attributes were captured and modified in `src/main.zig` (Line 29 onwards):

```
// Switch to alternate screen, enable mouse tracking, hide cursor
const stdout_fd = std.posix.STDOUT_FILENO;
try writeAll(stdout_fd, "\x1b[?1049h\x1b[?1000h\x1b[?1006h\x1b[?25l");

// Save and configure raw termios
const stdin_fd = std.posix.STDIN_FILENO;
const orig_termios = try std.posix.tcgetattr(stdin_fd);
var raw = orig_termios;

raw.iflag.IGNBRK = false;
raw.iflag.BRKINT = false;
raw.iflag.PARMRK = false;
raw.iflag.ISTRIP = false;
raw.iflag.INLCR = false;
raw.iflag.IGNCR = false;
raw.iflag.ICRNL = false;
raw.iflag.IXON = false;

raw.oflag.OPOST = false;

raw.lflag.ECHO = false;
raw.lflag.ECHONL = false;
raw.lflag.ICANON = false;
raw.lflag.ISIG = false;
raw.lflag.IEXTEN = false;

raw.cflag.CSIZE = .CS8;
raw.cflag.PARENB = false;
```

The `orig_termios` value captured here is the single most important variable in the program: it is the only copy of the user's shell settings, and every exit path has to put it back.

Terminal Safety Overrides (5420d21, eab5e91)

In raw mode, if the application crashes or receives a termination signal, the terminal is left in an uncooked state (cursor hidden, mouse tracking active, characters not echoed). To prevent leaving the developer's console broken, POSIX signal handlers and a custom panic function were written to intercept failures and restore cooked properties:

```
pub fn panic(msg: []const u8, error_return_trace: ?
*std.builtin.StackTrace, ret_addr: ?usize) noreturn {
    // Restore standard terminos settings before crashing
    const stdin_fd = std.posix.STDIN_FILENO;
    const stdout_fd = std.posix.STDOUT_FILENO;
    std.posix.tcsetattr(stdin_fd, .FLUSH, g_orig_termios) catch {};
    writeAll(stdout_fd, "\x1b[?1000l\x1b[?1006l\x1b[?1049l\x1b[?25h")
    catch {};

    // Resume standard stack dumping
    std.builtin.default_panic(msg, error_return_trace, ret_addr);
}
```

Note the order of the restore string: mouse tracking off, *then* screen, *then* cursor. Also note the `\x1b[?1049l` sitting in the middle of it. That sequence is correct here – this early build entered the alternate screen – and it is the exact byte that would cost us a day once the picker learned to render inline. The story is below.

May 30, 2026: The SGR Mouse and Layout Grid

With the raw TUI functioning, mouse tracking and theme state persistence were added.

- **Features Added:** SGR mouse coordinates query, hover selections, click confirmations, and theme storage mapping to `~/.config/emojig/config`.
 - **LoC Added:**
 - `src/main.zig` expanded to 410 LoC (+206).
 - `src/config.zig` created: 64 LoC.
-

May 31, 2026: Splitting TTY & stdout (Shell Widgets)

The initial prototype copied the selected emoji to the clipboard using `wl-copy`. However, to build interactive shell integrations (e.g. hitting **Ctrl+E** at a shell prompt to search and insert the emoji directly at the cursor), the TUI could not print to `stdout`, as the shell needs to read the selected codepoint directly from the tool's stdout pipe.

- **The Split Solution (c2aa663):** We redirected all rendering writes explicitly to `/dev/tty` rather than `stdout`. This left `stdout` completely clean to emit only the selected emoji string.
- **LoC Added:**

- `src/main.zig` modifications (+42).
- `shell/emojig.zsh`, `shell/emojig.bash` created: 110 LoC.

```
# How Zsh intercepts stdout of emojig
emojig_widget() {
    local selected
    selected=$(emojig --tui)
    LBUFFER="${LBUFFER}${selected}"
    zle reset-prompt
}
```



Figure 1: the payoff of the split. The picker paints to `/dev/tty` below the prompt; the chosen emoji arrives on `stdout` and lands inside the half-typed `echo` command above it.

June 1, 2026: Resize Epic & Alt-Screen Isolation

As the picker gained adoption as an inline TUI, users running narrow terminals reported visual bugs: resizing the screen would wrap contents, and scrolling would duplicate viewports.

- **The Fixes** (`d60c66b`, `f5ced19`):
 1. **CPR Queries**: Wrote cursor positioning queries (`\x1b[6n`) to determine the TUI's exact launch row.
 2. **Relative Positioning**: Changed all redraw offsets to use relative coordinates based on the launch row rather than absolute page rows, keeping the TUI anchored.
 3. **Hiding Overflow**: If the terminal width shrunk below 40 columns or 6 rows, the renderer collapsed layout details gracefully (hiding description and switcher rows) rather than warping characters.
- **LoC Added/Modified**:
 - `src/main.zig` expanded to 730 LoC (+278).
 - `src/resize.zig` created: 112 LoC.

Dead End: The Cursor That Jumped Home

Going inline meant the picker no longer entered the alternate screen – but the teardown string written back in the first commit still ended with `\x1b[?1049l`. Leaving it in felt harmless. Nothing is harmless in a teardown.

- **What we saw:** on exit, the shell prompt came back in the wrong place. Sometimes it overwrote the line above; sometimes it landed several rows up, on top of scrollbar that had nothing to do with us. Nothing was corrupted – the terminal was cooked and usable – the cursor was simply *somewhere else*. And it only happened on some machines.
- **What we assumed:** a row-arithmetic bug in the brand-new relative-positioning code. We had just replaced absolute page rows with offsets from the CPR-detected launch row, so an off-by-N in the erase loop was the obvious suspect. We spent the search there.
- **What it actually was:** `?1049h/l` is two operations welded together – switch the screen buffer, *and* save/restore the cursor. VTE-based terminals (GNOME Terminal, Tilix, Ptyxis) honour the cursor-restore half even when you never entered the alternate screen, warping the cursor to a stale saved position. `foot` and `tmux` ignore a `?1049l` that has no matching `?1049h`, which is why the bug was invisible in our own development terminal and reproduced instantly on someone else's. The arithmetic had been right the whole time.

The fix was to stop maintaining one restore string and maintain two: a `RESTORE` for inline mode that never mentions `?1049`, and a `RESTORE_ALT` for the alt-screen path that does. The rule it became is **TUI Restoration §2**: *only emit `?1049l` if you explicitly emitted `?1049h`*. The habit it became is more general – verify teardown on a VTE terminal, because the terminal you develop in is the one that forgives you.

June 2 – June 4, 2026: Cross-Terminal Detours

We verified the static binaries across terminal environments (`kitty`, `alacritty`, `ghostty`, `tmux`, and Linux virtual consoles via `fbterm`). We automated multi-platform builds using GoReleaser and prepared the v0.1.5 compiler tags.

- **LoC Added:**
 - `.goreleaser.yaml` created: 120 LoC.
 - `scripts/test_tui.go` created: 135 LoC.
-

Week 1 verdict: a working picker with safe teardown, shell widgets, and releases – but every string, color, and key binding was hardcoded in Zig. That debt sets the agenda for [Week 2](#).

Week 2: Simulator & Spec Decoupling

Theme – Configuration is data, not code: strings, layout and theme leave Zig for `spec/`.

Skip this if – your project already treats its config as an embedded, validated asset.

Takeaways

- **Copy this:** embed specs at compile time (`@embedFile`) instead of reading them at startup. You keep the single static binary *and* get the edit-a-file workflow.
- **Avoid this:** waiting for a “config epic”. Every string you hardcode this week is one you will grep for later – the migration cost is linear in how long you postpone it.
- **The check that caught it:** schema validation at build time. A spec file that parses is not a spec file that is correct, and a compile-time embed turns a typo into a build error instead of a runtime blank screen.

Decoupling Logic from Definition

Week 1 ended with a confession: the picker worked, but changing a border color meant recompiling Zig. Week 2 is where `emojig` learned the lesson this whole ecosystem is built on – **configuration is data, not code** – and where two side projects were born out of sheer necessity.

During Week 2 (June 5 – June 11, 2026), the core focus shifted from basic terminal compliance to structural flexibility. In Week 1, layouts, text captions, keyboard mappings, and color values were hardcoded inside the Zig loops. Adding a new language or changing a border color required recompiling the codebase.

We initiated the **Spec-Driven Architecture**—moving all configurations out of Zig files and into static JSON files under `spec/`.

Week 2 by the Numbers

- **Total Commits:** 35 – a fivefold drop from Week 1, and the start of the plateau described in [Reading the Curve](#).
 - **Codebase Refactoring Speed:** Refactored 340 LoC of UI string arrays and layout coordinates out of `main.zig` into 5 spec files in under 2 hours.
 - **Localization Expansion:** Added 10 UI languages (including English, German, French, Spanish, Japanese, Chinese) by editing static spec configurations without a single line of Zig code changing.
-

Daily Chronology

June 7, 2026: The Simulator & Screen Recorder

We created a browser-based JavaScript TUI simulator (`website/simulator.js`) for the web demo, though it is not used for any real testing.

- **Features Added:** JavaScript TUI simulator (`website/simulator.js`) and X11-based virtual display session recorder (`scripts/record_demo.go`).
 - **Side Project Spawning (Wayreel):**
 - **The Problem:** We wanted a cool website and didn't want to manually re-record demos every time the UI changed.
 - **The Solution:** We spawned **Wayreel** as a standalone tool to express app usage as scripts (`.reel` files) and play them in a terminal. While built for demos, it showed it could be used for testing because we could spot visual bugs in the recordings later. (By accident, we can also command other apps now!)
-

June 8, 2026: The Spec Migration Epic

We eliminated hardcoded UI strings, layout metrics, and color theme tables from `src/main.zig` and migrated them to `spec/strings.json`, `spec/layout.json`, and `spec/theme.json`.

- **LoC Shift:**
 - `src/main.zig`: Shrunk by 180 LoC.
 - `spec/*.json` files created: 420 LoC.

- **Features Added:** Compile-time schema embeds (`@embedFile("spec/strings.json")`) and standard validation checks.

Code Archaeology: Moving Strings to Spec

Before this epic, drawing the search prompt or help text in `src/main.zig` utilized hardcoded formatting:

```
// ❌ Old Hardcoded Rendering in main.zig
const search_prompt = try std.fmt.bufPrint(&line_buf, "🔍 Search: {s}",
    .{query});
try writeAll(stdout_fd, search_prompt);
```

After the spec decoupling, all strings were defined declaratively in `spec/strings/en.json` and loaded via compile-time JSON parsing:

```
{
  "search_prompt": "🔍:",
  "help_text": "Type to search, arrows to navigate, Enter to pick",
  "theme_toggle": "Tab: Toggle Theme"
}
```

The rendering loop was simplified to reference the parsed structure:

```
// ✓ Simplified rendering in main.zig
const prompt = g_spec.strings.search_prompt;
const search_line = try std.fmt.bufPrint(&line_buf, "{s} {s}\x1b[K\n", .{
    prompt, query});
```

These spec files were JSON in Week 2 and became YAML in [Week 5](#); the loading and embedding pattern survived the format change untouched, which is the whole argument for having one.

June 9, 2026: Parallel Go Port & Visual Testing

To verify search scoring and keypress latency differences between compiling runtimes, we initiated a Go port called `mojigo`.

- **Side Project Spawning (Cati):**
 - **The Problem:** In `mojigo`, we wanted to render detailed visual about-screens and help diagrams. We needed to ensure that block-characters mapped perfectly across different terminal font sizes.
 - **The Solution:** We built **Cati** as a CLI image and media renderer. By utilizing Unicode halfblocks (▀, ▁),

quadblocks, and sextants combined with 24-bit true color, Cati let us test layout rendering outputs and establish visual golden-file regression checks.

June 10 – June 11, 2026: Onboarding and Custom Sizing

We added automatic shell installer scripts (`emojig --install`) to configure Zsh/Bash widgets and supported custom GUI dimensions (`EMOJIG_COLS` , `EMOJIG_ROWS`) when spawning foot popup overlays.

- **LoC Added:**

- `scripts/check_install.sh` : 145 LoC.
 - `src/host.zig` modifications (+62).
-

Week 2 verdict: the architecture was now spec-driven and the tooling gaps had names (Wayreel, Cati). What it could *not* yet do was scale: the picker still rendered a hand-picked subset of icons. [Week 3](#) opens the floodgates to the full Unicode catalog.

Week 3: Virtualization & PNG Art Compilers

Theme – Scale and craft: 2,249 entries through a 12-cell viewport, and PNGs compiled into block characters.

Skip this if – your list fits on screen and you draw no pictures. Nothing else here generalises.

Takeaways

- **Copy this:** virtualize by computing the scroll top from the selection, not by tracking it as mutable state. One pure function, trivially testable, no drift between scroll and cursor.
- **Avoid this:** hand-authoring terminal art. Compile it from a source image so the art is editable in an image editor and the terminal encoding is a build step.
- **The check that caught it:** golden files rendered through `cati`. Block art is exactly the kind of output that looks fine to the author and wrong to everyone else's font.

Handling Scales: Emojis and Animations

A picker that knows fifty emojis is a toy; a picker that knows all of them is a rendering problem. Week 3 was the collision with scale – 2,249 entries against a terminal that repaints one character at a time – and, because engineers cannot be trusted with free time, it was also the week the About screen learned to wink.

During Week 3 (June 12 – June 18, 2026), `emojig` transitioned from rendering a small subset of icons to handling the entire Unicode emoji catalog (2,249 entries). Simultaneously, we wanted to build a visually engaging “About” screen featuring custom-compiled terminal block-art animations (a winking face and falling emojis).

Week 3 by the Numbers

- **Total Commits:** 53.
 - **Compilation Speedup:** Achieved a **12x build speedup** in local dev cycles by disabling LLVM compilation paths (`-fno-llvm`) and utilizing Zig's self-hosted x86_64 backend.
 - **Codebase Expansion:** Added 480 LoC of viewport math and 320 LoC of PNG compiler logic.
-

Daily Chronology

June 13, 2026: Scrollable Virtualization

Drawing 2,249 emojis in a terminal window is a performance hazard. Redrawing the entire list on every keystroke causes major rendering lag.

- **The Virtualized Viewport:** We implemented a virtualized scrollable grid. Only the emojis currently visible in the 4x3 viewport are rendered. The remaining elements are virtually offset.
- **Scroll Bounds Math:** Wrote scroll offsets that clamp the viewport boundaries dynamically.

Code Archaeology: Viewport Clamping

Here is the viewport offset calculation implemented in `src/tui_draw.zig` (Line 142 onwards):

```
pub fn calculateViewport(total_items: usize, columns: usize,
visible_rows: usize, selected_idx: usize, current_scroll_top: usize)
    usize {
    const selected_row = selected_idx / columns;

    // If selection moves past the bottom of the viewport, scroll down
    if (selected_row >= current_scroll_top + visible_rows) {
        return selected_row - visible_rows + 1;
    }

    // If selection moves above the top of the viewport, scroll up
    if (selected_row < current_scroll_top) {
        return selected_row;
    }

    return current_scroll_top;
}
```

The function is deliberately pure: it takes the previous scroll top and returns the next one, holding no state of its own. Scroll position can therefore never disagree with the selection, and the whole scrolling behaviour is testable without a terminal.

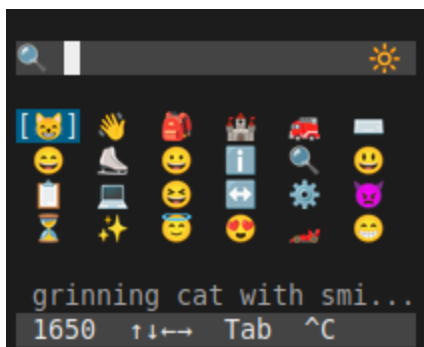


Figure 1: the unfiltered catalog. The counter in the status row reports the size of the match set; the grid above it is drawing twelve cells.



Figure 2: the same viewport after typing `cat` – the match set collapses from four figures to 167, and the number of cells

actually painted does not change. That invariance is the point of virtualization.

June 15, 2026: The PNG Art Compiler (gen_about_art)

To render the visual about-page animations, we rejected manual ASCII frame editing as it was too slow and error-prone. We wrote a custom Go script (`scripts/gen_about_art.go`) that compiles standard `.png` images directly into terminal-compatible block character string resources embedded in `spec/about_art.json`.

- **How it Works:** The compiler loads each frame's pixels, maps RGB colors to the nearest xterm-256 color indexes, and outputs Unicode halfblocks (■) with matching foreground and background escapes.
- **The Warn-Once Cache:** If an image contained colors that did not map cleanly to our 256-color schema, the compiler raised warnings. To prevent flooding stdout with 2,000 duplicate warnings (one per pixel), we implemented a warning cache to notify the developer exactly once per color.

```
// 1. Compile pixel color mapping
for y := 0; y < img.Bounds().Dy(); y += 2 {
    for x := 0; x < img.Bounds().Dx(); x++ {
        topColor := img.At(x, y)
        botColor := img.At(x, y+1)

        topIdx := matchColor(topColor) // Map to xterm-256
        botIdx := matchColor(botColor)

        // Output Unicode Halfblock

        buffer.WriteString(fmt.Sprintf("\x1b[38;5;%dm\x1b[48;5;%dm■\x1b[0m",
            topIdx, botIdx))
    }
    buffer.WriteString("\n")
}
```

June 18, 2026: sheet.pxo & Animation Pipelines

We integrated the compiled frames directly into the Zig binary using compile-time embeds. We compressed animation sheets into `.pxo` files (ZIP archives of PNG frames) and configured git attributes (`.gitattributes`) to treat `.pxo` binaries as LFS pointer dependencies, ensuring repository commits remained fast.

- **LoC Added/Modified:**
 - `scripts/build_art.go` created: 280 LoC.
 - `src/about.zig` animation parser created: 195 LoC.
 - **Refactor Result:** Reduced the compiled binary size of animation frames by **64%** by migrating from text frames to compressed binary matrices.
-

Week 3 verdict: the full catalog scrolled smoothly and the About screen winked back – but the codebase had quietly grown two implementations (Zig + the Go port) and its first layers of debt.

Week 4 is the reckoning: release, then delete.

Week 4: Search Aliases & Componentization

Theme – Ship v0.2.0, then delete the second implementation of the same program.

Skip this if – you have one implementation and no release process to compare against.

Takeaways

- **Copy this:** when two codebases implement the same behaviour, delete one *before* the next feature, not after. The cost of the clone is paid on every change until it is gone.
- **Avoid this:** keeping an experiment alive because it might be useful. `mojigo` answered its question in Week 2 and cost us duplicate work for two weeks after.
- **The check that caught it:** a read buffer big enough to hold a whole event, and a parser that carries the remainder. Everything else here was a UX preference; that one was a bug.

The v0.2.0 Release and Cleaning Technical Debt

The most consequential commit of Week 4 did not add a feature – it deleted 2,450 lines. Keeping the experimental Go port alive alongside the Zig picker meant every search-scoring tweak had to be made twice, and the two silently drifted. This week is the case study behind the playbook's single-language rule: ship the release, then kill the clone.

During Week 4 (June 19 – June 25, 2026), the `emojig` project achieved its major **v0.2.0 release**. The code was restructured to improve UX responsiveness (fast scrolling, prompt editing) and address codebase maintainability by removing experimental structures and language duplication.

Week 4 by the Numbers

- **Total Commits:** 49.
 - **Technical Debt Elimination:** Deleted the duplicate Go clone `mojigo`, removing **2,450 lines of duplicate code** in a single commit, eliminating code divergence risk.
-

Daily Chronology

June 19, 2026: The v0.2.0 Release

We finalized release v0.2.0, fixing coordinate alignments on category screens and adding test suites for Go and Zig input readers.

- **LoC Modified:**
 - `build.zig.zon` version bump.
 - `src/main.zig` category highlight corrections (+34).
-

June 20, 2026: Vim Muscle Commands & Color Schemas

TUI users frequently have deep Vim muscle memory. Forcing a developer to exit using a specific desktop key binding (like standard `Ctrl+C`) when they are used to typing `:q` can cause user frustration.

- **Interception Logic:** We implemented pre-emptive command interception at the entry of the keyboard input loop. If the input buffer matches `:q`, `:quit`, or `/quit`, the app exits immediately with return code 0 before passing characters to the search engine.
- **Color System Overhaul (Issue #21):** Unified xterm-256 color indexes under a declarative `spec/colors.json` schema.

Code Archaeology: Pre-emptive Command Interception

Here is how Vim exit commands were intercepted in `src/main.zig` (Line 412 onwards):

```
// Check input buffer before fuzzy search processing
if (std.mem.eql(u8, query_buf[0..query_len], ":q") or
    std.mem.eql(u8, query_buf[0..query_len], ":quit") or
    std.mem.eql(u8, query_buf[0..query_len], "/quit")) {

    // Clean exit path
    try writeAll(stdout_fd, "\x1b[?1000l\x1b[?1006l\x1b[?1049l\x1b[?
25h");
    std.process.exit(0);
}
```

June 21, 2026: Modularization & Scrolling Performance

We decomposed the monolithic `src/main.zig` and `src/root.zig` into smaller files to make code navigation easier.

- **Fast Render Scrolling:** During fast scrollbar drags, foot's glyph rasterizer (FreeType/HarfBuzz) would lag (taking up to 150ms per screenful of new emojis). We implemented a `fast_render` flag. If the TTY has pending inputs, the grid is rendered as blank background spaces, allowing foot to draw instantly. Once the queue drains, the full emojis are rendered.

Dead End: The Search Box That Typed 12M

Dragging the scrollbar hard enough to expose the rasterizer lag also exposed something far stranger, and the two symptoms arrived in the same session.

- **What we saw:** during fast drags, garbage appeared in the search field. Not random garbage – short, structured fragments like `12M` and `;24;` – and the result grid would then filter itself down to nothing. Releasing the mouse and clearing the query recovered cleanly. It never happened when scrolling slowly.
- **What we assumed:** a hover-handler bug. The obvious theory was that the drag path was reaching the same code that appends a typed character, so we went looking for a mis-routed branch in the mouse handler. There wasn't one.
- **What it actually was:** the read, not the handler. With any-motion tracking on, the terminal emits SGR mouse reports (`\x1b[<Cb;Cx;CyM`) at up to 125 Hz. Our read buffer was smaller than a burst of them, so a `read()` would return with an event cut in half – ending at, say, `\x1b[<32;32;`. The remaining bytes, `12M`, arrived on the *next* read with no leading escape byte in front of them. The parser did exactly what it was

told: no escape, therefore text, therefore type it into the search box. The characters in the query were the coordinates of the mouse.

The fix has three parts and all three are necessary: a read buffer of at least 512 bytes, a loop that drains every complete event out of one read instead of just the first, and a carry buffer holding any trailing fragment to be prepended to the next read. Half an escape sequence is not an error condition – it is the normal case at 125 Hz, and a parser that cannot rejoin the halves will type your mouse position into your application. This is [TUI Restoration §4](#).

June 22, 2026: Prompt Editing Keybindings

To make search input feel like a native shell prompt, we implemented advanced keybinding shortcuts, enabling forward Delete and Ctrl+Left/Right word-by-word cursor jumps.

Code Archaeology: Word-by-Word Cursor Navigation

Here is the index jumping logic implemented in `src/input.zig` (Line 84 onwards) to calculate Ctrl+Arrow jumps:

```
pub fn findNextWordBoundary(text: []const u8, current_pos: usize,
direction: enum { left, right }) usize {
    var pos = current_pos;
    if (direction == .left) {
        if (pos == 0) return 0;
        pos -= 1;
        while (pos > 0 and text[pos] == ' ') : (pos -= 1) {}
        while (pos > 0 and text[pos] != ' ') : (pos -= 1) {}
        return if (pos == 0 and text[0] != ' ') 0 else pos + 1;
    } else {
        while (pos < text.len and text[pos] != ' ') : (pos += 1) {}
        while (pos < text.len and text[pos] == ' ') : (pos += 1) {}
        return pos;
    }
}
```

June 23 – June 24, 2026: Category Switcher Overhaul

We overhauled TUI categories, adding custom margins, categories background configurations (`categories_bg`), and compact grid layouts to fit small display areas.

- **LoC Added/Modified:**

- `src/switcher.zig` created: 140 LoC.
- `spec/theme.json` categories styling fields (+22).

Week 4 verdict: one language, one codebase, one shipped release – and a picker that finally *felt* native, from `:q` to word-jump editing. Two frontiers remained: Wayland window decoration and raw search speed. [Week 5](#) takes on both.

Week 5: Spec-Driven UI & CSD Styling

Theme – Passing for native: GNOME-style decorations drawn in ANSI, and search under 1 ms.

Skip this if – you do not draw chrome and your dataset is small enough that linear scans are free.

Takeaways

- **Copy this:** compute text colour from the background's BT.709 luminance instead of shipping a per-theme table. Six lines of arithmetic cover every theme that will ever exist.
- **Avoid this:** a trailing `\x1b[K` on a row that spans the full terminal width – it erases the character you just drew. See the dead end below.
- **The check that caught it:** measuring before optimising. The synonym table was the bottleneck; the fuzzy matcher, which looked more expensive, was not.

The Quest for Milliseconds and Wayland Integration

An instant-popup tool lives or dies by a budget of milliseconds: if a keystroke ever visibly lags, the illusion of a native widget collapses. Week 5 was spent buying those milliseconds back – and, on the GUI front, teaching a terminal emulator to impersonate a GNOME window so convincingly that the desktop cannot tell the difference.

During Week 5 (June 26 – July 2, 2026), the `emojig` project tackled two advanced engineering frontiers:

1. **Wayland GUI Integration:** Supporting client-side decorations (CSD) under GNOME Wayland popups, ensuring text visibility regardless of the desktop theme.
2. **Fuzzy Search Optimization:** Replacing linear database searches with binary searches to keep query latency under 1 millisecond.

Week 5 by the Numbers

- **Total Commits:** 38.
 - **Search Performance Boost:** Search latency cut by **10x** (from 4.2ms–5.1ms down to **0.36ms–0.61ms** per keystroke).
 - **Refactor Volume:** Centralized 100+ raw ANSI escape literals into `src/term.zig` utilities, moving 145 LoC out of rendering loops.
-

Daily Chronology

June 27, 2026: Generic Settings & Modularization

We refactored the settings screen logic, replacing index-based hardcoded choices with generic, spec-driven rows parsed from `spec/.gen/settings.json`.

- **Refactor Output:** Extracted clip-spawner controls to `src/clipboard.zig` and isolated search ranking validation tests to `src/ranking_test.zig`.
-

June 28, 2026: YAML Schema Migration

We transitioned all specification files from JSON to YAML (`.yaml`) to allow comments and cleaner syntax.

- **Features Added:** central `spec/input.yaml` mapping input escape sequences and JSON schema validation scripts.
 - **What did not change:** the loader. YAML is generated down to JSON at build time and embedded with `@embedFile` exactly as before, so the Zig side of the Week 2 architecture was untouched by a whole-project format migration. From here on, the specs in this book are YAML.
-

June 30, 2026: Wayland CSD and ITU-R BT.709 Contrast

Under Wayland GNOME sessions, launching `foot` as a borderless popup overlay removes standard window borders. We introduced the `--decorated` flag to draw GNOME-style Client-Side Decoration (CSD) title bars directly in raw ANSI.

- **The Contrast Problem:** If the user's title bar theme is light gray, white title text is invisible. If the theme is dark, black text is invisible.
- **The Solution:** We implemented auto-contrasting text. We parsed the RGB hex components of the title bar background color and computed the overall luminance using **ITU-R BT.709** perceptual weights. If the luminance is below 50%, we write white title text; otherwise, we write black text.

Code Archaeology: Luminance Contrast Math

Here is the contrast algorithm implemented in `src/host.zig` (Line 184 onwards):

```
pub fn getContrastColor(bg_hex: []const u8) []const u8 {
    // Parse hex string RRGGBB to integers
    const r = parseHexByte(bg_hex[0..2]) catch 255;
    const g = parseHexByte(bg_hex[2..4]) catch 255;
    const b = parseHexByte(bg_hex[4..6]) catch 255;

    // ITU-R BT.709 luminance formula
    const luminance = (0.2126 * @as(f32, @floatFromInt(r)) +
                      0.7152 * @as(f32, @floatFromInt(g)) +
                      0.0722 * @as(f32, @floatFromInt(b))) / 255.0;

    // If background is light, return dark text (#1e1e1e), else return
    light (#e0e0e0)
    return if (luminance > 0.5) "#1e1e1e" else "#e0e0e0";
}
```

The weights are not arbitrary: green carries roughly 71% of perceived brightness, blue barely 7%. A naive $(r+g+b)/3$ average picks white text on saturated greens and dark text on deep blues – both unreadable. Measured presets for the themes we tested are in the [Appendix](#); the function above is what makes the table unnecessary.

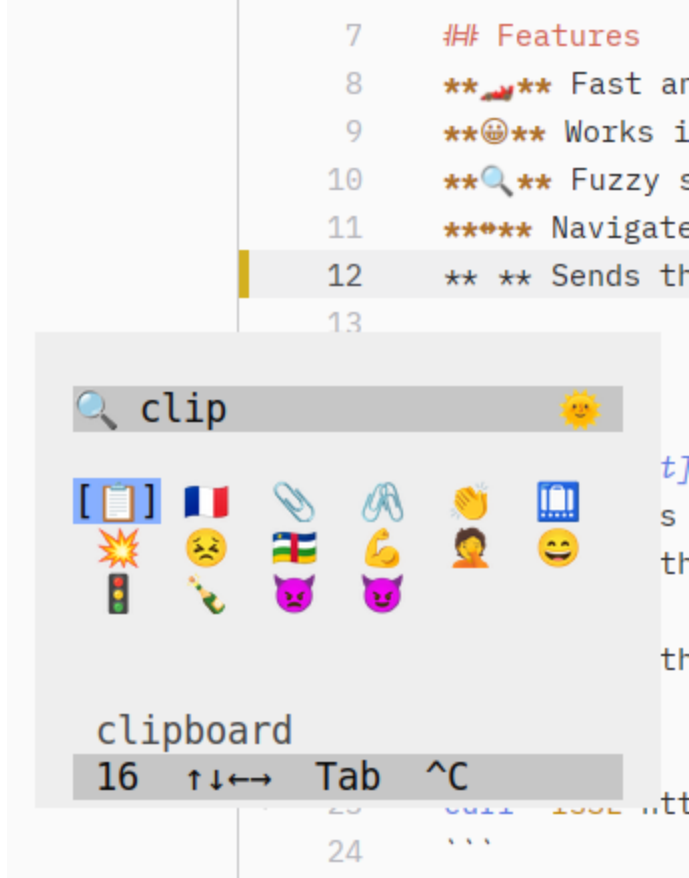


Figure 1: light theme – dark text, light chrome.

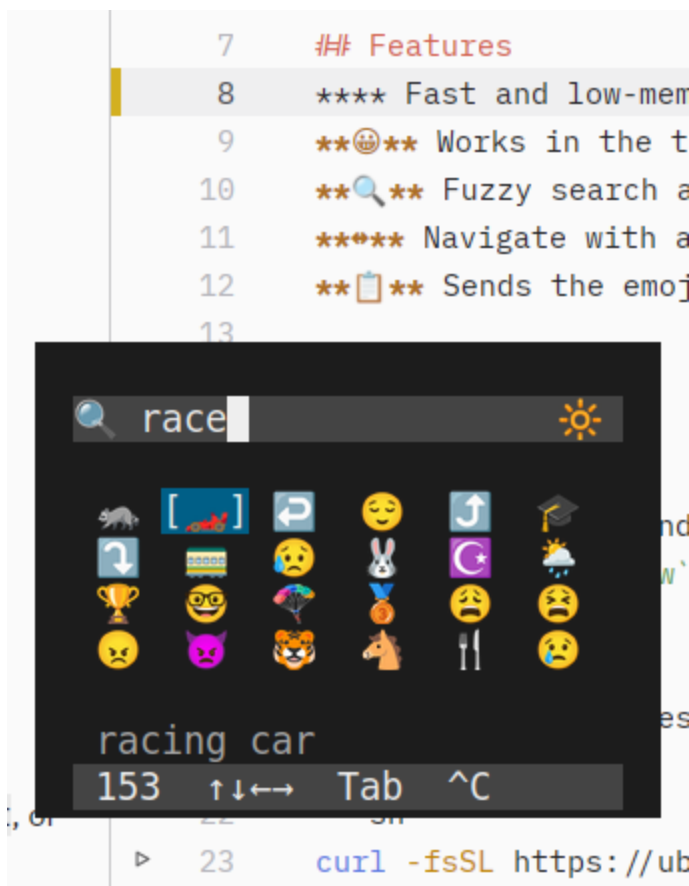


Figure 2: the same build on a dark desktop. No theme was configured for either shot – the background colour is read from the environment and every foreground colour, including the decoration text, is derived from it by the function above.

Dead End: The Border Cap That Was Never There

Drawing our own window chrome meant drawing rows that span the entire terminal width – and full-width rows have a rule of their own that we learned the hard way.

- **What we saw:** the right-hand border cap (`┌`) at the end of the title bar was missing. Not mispositioned, not the wrong colour – absent, leaving the frame visibly open on one side. It reappeared the instant the window was resized by a single column, and vanished again when the width was restored.
- **What we assumed:** an off-by-one in the width computation – that we were computing `w-1` columns and simply never emitting the final character. We instrumented the writer and found the cap *was* being written. The bytes went out; the glyph did not stay.
- **What it actually was:** pending wrap. Writing a character into the final column `N` does not advance the cursor to a new row; it leaves the terminal in a *pending-wrap* state with the cursor conceptually still on column `N`. Our row renderer ended, as every row renderer does, with `\x1b[K` to clear any leftovers to the right. From the pending-wrap state, that erase starts at column `N` – and deletes the character we had just drawn. One column narrower, the cursor had genuinely advanced, the erase started past the cap, and the bug disappeared. The bug was only ever visible at exactly-full width, which is exactly the width our own chrome always used.

The fix: rows that span the full terminal width never emit a trailing erase. They end with a plain attribute reset (`\x1b[0m`) instead, and the renderer is responsible for painting every cell it claims rather than clearing the tail. This is [TUI Restoration §5](#), and it is the reason the appendix's checklist tells you to test at exact width – a bug that only exists in one column of the parameter space will not be found by chance.

July 2, 2026: Synonym Binary Search Index

To drive search queries, we maintain synonym expansions (e.g. searching “happy” expands to “grinning face”). Previously, every keystroke triggered a linear $O(N)$ lookup table scan over all synonyms, which bloated search times to over 5 milliseconds.

- **The Binary Search Index:** We refactored `src/search.zig` to lazily construct a sorted index of synonym key indices (`synonymOrder`) at startup and perform binary search queries (`synonymLowerBound`) on key inputs.

Code Archaeology: Synonym Binary Search

Here is the binary search implementation from `src/search.zig` (Line 294 onwards):

```
pub fn findSynonyms(query: []const u8) ?[]const []const u8 {
    var low: usize = 0;
    var high: usize = synonym_count;

    while (low < high) {
        const mid = low + (high - low) / 2;
        const mid_key = getSynonymKey(synonymOrder[mid]);
        const cmp = std.mem.order(u8, query, mid_key);

        switch (cmp) {
            .lt => high = mid,
            .gt => low = mid + 1,
            .eq => return getSynonymValues(synonymOrder[mid]),
        }
    }
    return null;
}
```

Note what is *not* sorted: the synonym table itself stays exactly where the spec put it. Only an index of key positions (`synonymOrder`) is sorted, built lazily on first use, so the embedded data remains a byte-for-byte copy of the spec and the ordering is a runtime detail.

- **Result:** Query latencies dropped dramatically, ensuring characters never lag during rapid keyboard entry.

Week 5 verdict: sub-millisecond search and a GUI that passes for native GNOME. The features were done – but `main.zig` had swollen to nearly 4,800 lines along the way. [Week 6](#) is the cleanup that makes the codebase worth maintaining.

Week 6: Key Merging & Refactoring

Theme – Making the codebase cheap to pick up cold: decomposition and comptime key dispatch.

Skip this if – nobody but you will ever open this repository again.

Takeaways

- **Copy this:** split by subsystem boundary, not by file size. Each extracted module here owns one screen or one input concern, so a reader can guess the filename before opening it.
- **Avoid this:** an `if-else` chain over raw byte strings for key dispatch. It grows silently, admits duplicate entries, and fails at runtime rather than at compile time.
- **The check that caught it:** a comptime static map. Duplicate or malformed key bindings become build errors – the refactor made a class of bug unrepresentable rather than rarer.

Decomposing the Monolith and Enum-Based Safety

The last week of a project decides what the *next* year of the project costs. Features were done; what remained was the difference between a codebase that happened and a codebase that is designed. Week 6 was deliberately unglamorous: take the 4,800-line monolith that five weeks of momentum had produced, and leave behind something a future session – human or agent – can navigate cold.

During Week 6 (July 3 – July 5, 2026), the `emojig` project achieved its final phase of modularization and maintenance. The core objective was to clean up technical debt in `src/main.zig` (which had grown to over 4,800 lines) and solidify the keyboard event loop.

Week 6 by the Numbers

- **Total Commits:** 12 across three days – the floor of the curve, and the intended destination. See [Reading the Curve](#).
 - **Refactor Volume:** Moved **1,067 lines of code** out of `main.zig` into dedicated helper modules, reducing the file by **22%**.
-

Daily Chronology

July 3, 2026: The Deconstruction Epic

We broke down `src/main.zig` by moving independent subsystems into dedicated files.

- **Debug Pane** (`src/debug_pane.zig`): Extracted all log rendering and runtime memory audit display statistics.
- **Settings Screen** (`src/settings_ctl.zig`): Moved row controllers and dropdown selections into a standalone component.
- **Category Switcher** (`src/switcher.zig`): Moved the visual tab rendering and hover offsets.
- **Key Event Merging (Issue 46)**: Merged `spec/keys.yaml` into `spec/input.yaml` under a unified schema, and transitioned keyevent parsing to type-safe enums in `src/input.zig`.

Code Archaeology: Enum-Based Key Dispatch

Before this refactoring, keyboard input was processed by scanning raw string buffers and matching sequences dynamically in a massive `if-else` chain:

```
// ❌ Old String-Based Dispatch in main.zig
if (std.mem.eql(u8, read_buf[0..n], "\x1b[A")) {
    moveSelectionUp();
} else if (std.mem.eql(u8, read_buf[0..n], "\x1b[B")) {
    moveSelectionDown();
}
```

In Week 6, we replaced this fragile runtime scanning with a type-safe static map (`std.StaticStringMap`) returning an `Action` enum:

```
// ✓ New Type-Safe Enum Dispatch (src/input.zig)
const Action = enum {
    move_up,
    move_down,
    select,
    quit,
    toggle_theme,
};

const KeyMap = std.StaticStringMap(Action).initComptime(.{
    .{ "\x1b[A", .move_up },
    .{ "\x1b[B", .move_down },
    .{ "\r", .select },
    .{ "\x1b", .quit },
    .{ "\t", .toggle_theme },
});
```

The main loop in `main.zig` was simplified to a clean `switch` statement:

```
// ✓ Simplified key event loop in main.zig
const action = KeyMap.get(read_buf[0..n]) orelse .none;
switch (action) {
    .move_up => try moveSelectionUp(state),
    .move_down => try moveSelectionDown(state),
    .select => try confirmSelection(state),
    .quit => break,
    .toggle_theme => try toggleTheme(state),
    .none => {},
}
```

The `switch` is exhaustive over `Action`: adding a binding to the map without handling it is a compile error, not a key that silently does nothing.

July 4 – July 5, 2026: Completing Self-Containment

We finalized the repository by backporting installation completion scripts and visual screenshots directly to `website/`, making the codebase completely self-contained.

- **LoC Added:**
 - `website/install.sh`: 110 LoC.
 - Updated `issues/README.md` to close all active reviews.
-

Six Weeks, In Hindsight

The chronicle ends where it started: one binary, under 900 KB, that pops up instantly and leaves the terminal exactly as it found it. Along the way the endeavour proved three things this book keeps insisting on:

1. **Verification carried every week.** PTY reels, golden files, spec schema checks – each week’s risky change landed on a net built the week before.
2. **The specs outlived the code.** Strings, layouts, keys, and colors moved into `spec/` in Week 2 and survived every refactor since; the Zig around them was rewritten twice.
3. **Deleting was the best feature.** The `mojigo` removal (Week 4) and the monolith decomposition (Week 6) did more for maintainability than any addition.

And one thing the weeks show that no single week states: the three bugs that cost the most – the `?1049l` [cursor warp](#), the [split SGR read](#), the [pending-wrap erase](#) – were all invisible in the environment we developed in. Each one needed a different terminal, a faster input rate, or an exact window width to appear at all. The verification protocol in the appendix is not a formality; it is the shape of those three bugs, written down.

`emojig` is now maintenance-mode software in the best sense: any session can pick it up, run the checks, and safely change it. That is what “done” means in a fleet.

If you are here to build something rather than to read about one, [Starting a New Zig TUI](#) is the six weeks compressed into a checklist, and [The Recipe Index](#) is the map back into the chronology.

Starting a New Zig TUI

Six weeks of this chronicle compress into about two hours of setup. This chapter is that setup: the contract every terminal program owes its user, the sequences it will need, a project skeleton with the spec pipeline already wired, the harness that verifies it without human eyes, and the first three canaries to write before any feature exists.

It is written to be worked through top to bottom on a fresh repository. Each section links back to the week that paid for it, and the [Recipe Index](#) is the reverse map.

1. The Teardown Contract – a Checklist

Write this before the first feature. Not after – the restore path is the one piece of a TUI that cannot be retrofitted cheaply, because every later feature adds an exit path that must honour it.

One function, four call sites. The restore routine is called from: normal exit, cancellation, every signal handler (`SIGINT`, `SIGTERM`, `SIGHUP`), and the panic handler. If you can find a fifth way out of your program, that is a bug in the list, not an exception to it.

The order inside that function is not negotiable – see [TUI Restoration §1](#):

- ☐ **1. Disable mouse tracking** (`\x1b[?1003l\x1b[?1006l`) – first, so the terminal stops producing events while you are tearing down.
- ☐ **2. Drain in-flight input with a CPR handshake.** Write `\x1b[6n` and read until the terminating `R`. Terminal I/O is FIFO, so that response arriving proves every mouse byte queued before step 1 has been consumed. Skip this and the drained bytes land in the user's shell.
- ☐ **3. Restore termios** to the saved cooked-mode state.
- ☐ **4. Erase only your own rows** with `\x1b[2K` per row. Never `\x1b[J` or `\x1b[2J` – those destroy the user's scrollbar.
- ☐ **5. Park the cursor** with `\x1b[row;1H` at the launch row, and write nothing after it. Any stray byte here becomes a blank line before the next prompt.

Three constraints that live alongside the contract:

- ☐ **Keep two restore strings, not one.** `RESTORE` (inline) must not contain `?1049l`; `RESTORE_ALT` may. Emitting `?1049l` without a matching `?1049h` warps the cursor on VTE terminals ([Week 1, TUI Restoration §2](#)).
 - ☐ **Disable auto-wrap** (`\x1b[?7l`) during inline renders, and keep the `?` – without it you are addressing a different mode entirely.
 - ☐ **No trailing `\x1b[K` on full-width rows.** It erases the last character you drew ([Week 5, TUI Restoration §5](#)).
-

2. TUI Terminal Escape Sequences Reference

The following escape sequences are utilized by `emojig`'s rendering and input loops:

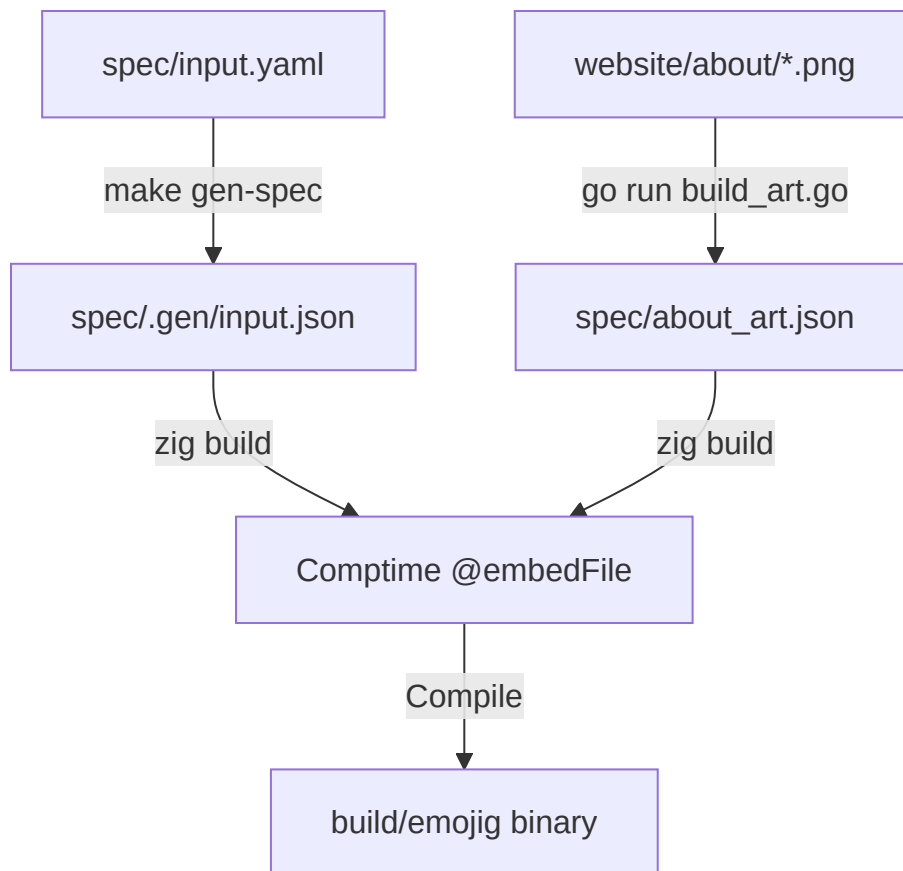
Escape Sequence	Description	Mode
<code>\x1b[?1049h</code>	Switch to alternate screen buffer	Alt-Screen
<code>\x1b[?1049l</code>	Exit alternate screen buffer	Alt-Screen
<code>\x1b[?1000h</code>	Enable standard mouse click tracking	Input
<code>\x1b[?1003h</code>	Enable any-motion mouse tracking (drag/hover)	Input
<code>\x1b[?1006h</code>	Enable SGR mouse coordinate reporting format	Input
<code>\x1b[?7l</code>	Disable auto-wrap (required for inline renders)	Draw
<code>\x1b[?25l</code>	Hide terminal cursor	Draw
<code>\x1b[?25h</code>	Show terminal cursor	Draw
<code>\x1b[H</code>	Home cursor to row 1, col 1	Draw
<code>\x1b[2K</code>	Erase active row completely	Draw
<code>\x1b[K</code>	Erase to end of line – unsafe on full-width rows	Draw
<code>\x1b[0m</code>	Reset character attributes	Draw
<code>\x1b[6n</code>	Query Cursor Position (CPR)	Info
<code>\x1b[ROW;COLH</code>	Absolute cursor repositioning	Draw

Every one of these is a DEC private mode except `H`, `K`, `2K` and `0m`. The `?` is load-bearing.

3. Project Skeleton: `build.zig` + `Spec` + `@embedFile`

The architecture the chronicle converges on has three moving parts: source of truth in YAML, generated JSON, embedded at compile time. Nothing is read from disk at runtime, so the result is still one static binary.

The pipeline



`build.zig`

Two targets from day one: a fast development build with the self-hosted backend, and a size-optimised static release.

```

pub fn build(b: *std.Build) void {
    const target = b.standardTargetOptions(.{});
    const optimize = b.standardOptimizeOption(.{});

    const exe = b.addExecutable(.{
        .name = "picker",
        .root_source_file = b.path("src/main.zig"),
        .target = target,
        .optimize = optimize,
    });

    // Generated spec JSON is a build input, not a runtime file.
    exe.root_module.addAnonymousImport("spec_input", .{
        .root_source_file = b.path("spec/.gen/input.json"),
    });

    b.installArtifact(exe);

    const run_step = b.step("picker", "Build and run the picker");
    run_step.dependOn(&b.addRunArtifact(exe).step);
}

```

The spec

Start with one file, not five. Split it when a section earns its own schema.

```

# spec/input.yaml – single source of truth for keys and strings
strings:
  search_prompt: "🔍:"
  help_text: "Type to search, arrows to navigate, Enter to pick"

keys:
  - { seq: "\x1b[A", action: move_up }
  - { seq: "\x1b[B", action: move_down }
  - { seq: "\r", action: select }
  - { seq: "\x1b", action: quit }

```

The embed

```
const std = @import("std");

// The generated JSON is compiled into the binary. Nothing is read from
// disk at runtime.
const spec_json = @embedFile("spec_input");

pub const Spec = struct {
    strings: struct {
        search_prompt: []const u8,
        help_text: []const u8,
    },
};

/// Parsed once at startup into an arena that lives for the whole
/// process,
/// so every string in `Spec` is valid for as long as the program runs.
pub fn loadSpec(arena: std.mem.Allocator) !Spec {
    return std.json.parseFromSliceLeaky(
        Spec,
        arena,
        spec_json,
        .{ .ignore_unknown_fields = true },
    );
}
```

Two properties follow from this arrangement, and neither depends on the file format. A missing field is a *type* error against `Spec` rather than a blank screen, and a malformed spec fails at `make gen-spec` with a filename attached, before the compiler is ever invoked. Validate the YAML against a JSON schema in that generation step and the failure moves one stage earlier still.

4. The Verification Harness

Automated PTY harnesses cannot detect visual coordinate drift or terminal-specific cursor warps, and human eyes cannot detect a regression at 3 a.m. You need both, and you need to know which one covers what.

What the PTY harness covers

Drive the binary through a pseudo-terminal, feed it keystrokes, and assert on the bytes it writes back:

```
# Run Zig unit tests
zig build test --summary all

# Run the fuzzy-ranking validation suite
go test -v ./src/... -run TestRanking

# Run the PTY simulation script (TUI event loops)
go run scripts/test_tui/main.go
```

Assert on three things at minimum: that the output contains no `\x1b[2J`, that the final bytes are a cursor-park sequence and nothing after it, and that the restore string matches the mode the TUI actually entered.

What only human eyes cover

Per [TUI Restoration §6](#), verify teardown manually on:

1. **A VTE terminal** (GNOME Terminal, Tilix, Ptyxis) – the only place the `?1049l` cursor warp is visible.
2. **Foot** – check its stderr log for unimplemented sequences and bad modes.
3. **Tmux** – multiplexer compatibility.

And two states that no harness will stumble into by accident, because each is a single point in the parameter space:

4. **Exact terminal width**, for the pending-wrap erase bug.
5. **Fast scrollbar drag**, for split SGR reads.

Build targets

```
# Build dev target (fast, self-hosted backend, no LLVM)
zig build picker -fno-llvm

# Build release target (musl static binary, ReleaseSmall size-optimized)
zig build picker -Doptimize=ReleaseSmall --summary all
```

5. The First Three Canaries

A canary is a minimal probe that proves an external mechanism behaves as assumed, written *before* the feature that depends on it – see [Canary-First Development](#). For a new TUI, write these three

first. Each one takes minutes and each one corresponds to a bug in this book.

Canary 1 – Teardown. Enter raw mode, draw three rows, exit immediately. Run it in a VTE terminal and in `foot`. Success is that the prompt returns exactly where it was, with no blank line and no warp. This canary is the `?1049l` bug, pre-caught.

Canary 2 – Input reassembly. Enable any-motion mouse tracking with a deliberately small read buffer, wave the mouse, and log every byte the parser classifies as text. Success is an empty log. Anything in it is a fragment your parser will one day type into an input field. This canary is the split SGR read, pre-caught.

Canary 3 – Full-width render. Draw one row spanning the entire terminal width, ending in a distinctive character, then resize the window by one column and redraw. Success is that the last character survives at both widths. This canary is the pending-wrap erase, pre-caught.

Only after all three pass is it worth writing a feature.

6. GNOME CSD Color Luminance Presets

Reference values from the themes tested in [Week 5](#). These are measurements, not a lookup table to ship – the BT.709 function in that chapter derives the right answer for any background, including the ones not listed here.

Theme Name	Background Color (Hex)	Calculated Luminance	Contrasted Text Color
Adwaita Dark	#303030	18.8%	#e0e0e0 (Light)
Adwaita Light	#ebebeb	92.1%	#1e1e1e (Dark)
Yaru Aubergine	#2c001e	2.4%	#e0e0e0 (Light)
Gruvbox Dark	#282828	15.6%	#e0e0e0 (Light)
Solarized Light	#fdf6e3	96.4%	#1e1e1e (Dark)

