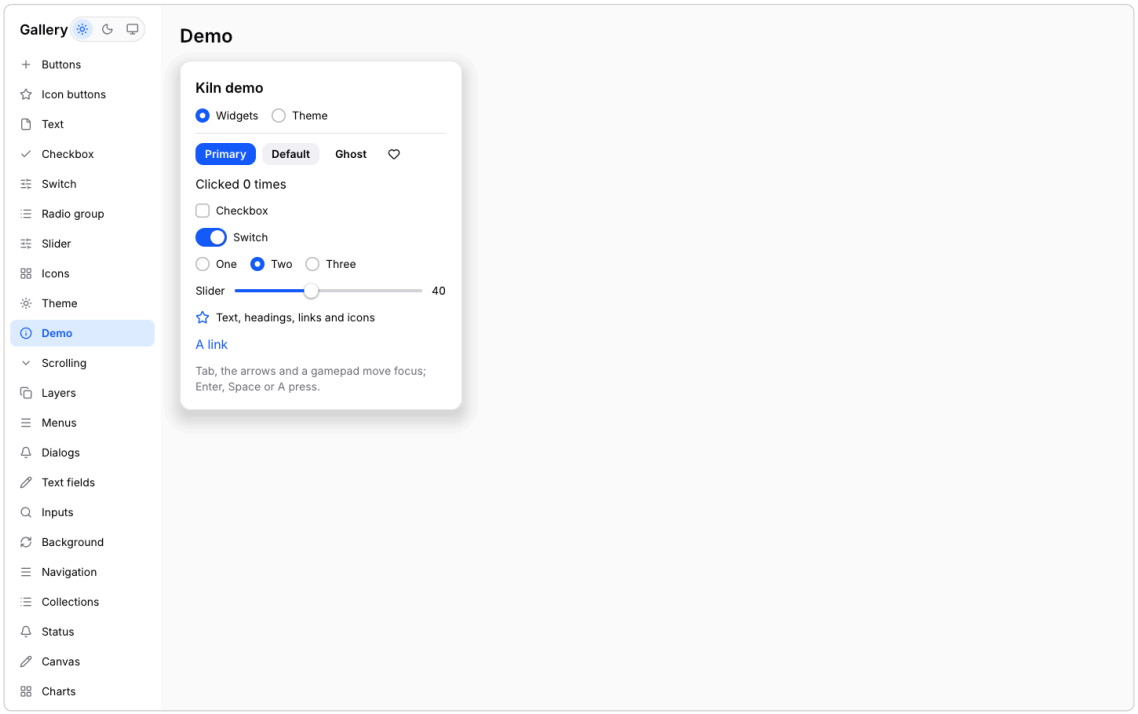
Kiln

A native UI library for C3



Contents

1 Introduction 7
2 Overview 8
2.1 The chapters 8
2.2 Layers 9
2.3 Dependencies 10
2.4 Modules and libraries 10
3 The frame 13
3.1 `begin()`: take in the input 13
3.2 Your frame function: describe the interface 14
3.3 `end()`: lay out and settle 14
3.4 `draw()`: hand over the draw list 14
3.5 What survives between frames 14
3.6 Compared with React 15
3.7 A custom component is just a function 15
4 App style and Game style 16
4.1 App style: Kiln owns the loop 16
4.2 Game style: your raylib loop 19
4.3 On the web 21
5 Background work 23
5.1 One UI thread 23
5.2 The calls 23
5.3 Lifetime 24
5.4 Memory 25
5.5 The worker pool 25
5.6 Waking the window 25
5.7 A worked example 26
6 Widgets 28
6.1 The five-step pattern 28
6.2 The core calls 30
6.3 Rules 32
6.4 Return values and binding 32
6.5 Options 32
6.6 The widget set 33
6.7 Loading looks 35
7 Input and focus 37
7.1 The input of a frame 37
7.2 Pointers and layers 39
7.3 Context menus 40
7.4 Focus 41
7.5 Shortcuts 44
7.6 Key routing 44
7.7 Scrolling 50
7.8 Drops 51
7.9 Drag and drop inside the app 51
7.10 Who has the keyboard 52
7.11 The focus ring 52
8 State and memory 53

8.1 Lifetimes ... 53
8.2 Widget state ... 54
8.3 Allocators ... 55
8.4 No allocations per frame ... 56
9 Motion ... 57
9.1 Duration and bounce ... 57
9.2 Springs find their frame ... 57
9.3 Motion tokens ... 58
9.4 Colors ... 59
9.5 Looping motion ... 59
9.6 Idle ... 60
9.7 Reduced motion ... 60
10 Theme ... 61
10.1 Tokens ... 61
10.2 Colors in OKLCH ... 63
10.3 Status tints ... 64
10.4 Chart series colors ... 64
10.5 Resolution ... 64
10.6 Scales ... 64
10.7 Presets ... 66
10.8 Customizing ... 66
11 Layout ... 68
11.1 From Tailwind to options ... 68
11.2 Containers ... 70
11.3 The element budget ... 71
11.4 Responsive layout ... 72
11.5 Container queries ... 72
11.6 Scrolling ... 73
11.7 Conventions ... 75
12 Layers ... 76
12.1 Declaring a layer ... 76
12.2 Z-order ... 77
12.3 Placement ... 77
12.4 Hit testing by layer ... 79
12.5 Dismissing ... 80
12.6 Presence and motion ... 81
12.7 The backdrop ... 82
12.8 Across platforms ... 82
13 Menus, popovers and tooltips ... 83
13.1 Two ways to write a menu ... 83
13.2 Rows ... 83
13.3 Keyboard ... 85
13.4 Submenus ... 85
13.5 Context menus ... 85
13.6 Popovers ... 87
13.7 Tooltips ... 88
13.8 Timing ... 88
13.9 Across platforms ... 88
14 Dialogs, sheets and toasts ... 90

14.1 Dialogs ... 90
14.2 Keys and focus ... 91
14.3 Button order ... 92
14.4 Bottom sheets ... 92
14.5 Toasts ... 94
14.6 Across platforms ... 96
15 Text editing ... 97
15.1 The editor ... 97
15.2 Graphemes and words ... 98
15.3 The text field ... 99
15.4 The mouse ... 102
15.5 Keys ... 102
15.6 The text area ... 103
15.7 Read-only fields and areas ... 105
15.8 Edit actions ... 105
15.9 The edit menu ... 106
15.10 The caret ... 107
15.11 Input method composition ... 107
15.12 The platform and key routing ... 108
16 Inputs ... 110
16.1 The field core ... 110
16.2 Search ... 110
16.3 Numbers ... 111
16.4 Select ... 112
16.5 Type-ahead ... 114
16.6 Segmented control ... 114
16.7 Dates ... 115
16.8 Times ... 117
16.9 Colors ... 117
16.10 Across platforms ... 119
17 Navigation ... 121
17.1 Tabs ... 121
17.2 Sidebar ... 122
17.3 Disclosures ... 124
17.4 Fitting the container ... 125
18 Collections ... 127
18.1 Selection ... 127
18.2 Lists ... 128
18.3 Tables ... 131
18.4 Reordering rows ... 134
18.5 Removing rows ... 135
18.6 Grids ... 135
19 Status ... 138
19.1 Tones ... 138
19.2 Progress bars ... 139
19.3 Badges ... 140
19.4 Alerts ... 141
19.5 Metrics ... 142
19.6 High contrast ... 142

20 Images 144
20.1 Making an image 144
20.2 Loading a picture 145
20.3 Drawing an image 146
20.4 How it's drawn 147
20.5 Memory 147
21 Canvas 148
21.1 Drawing in a canvas 148
21.2 Paths 149
21.3 Strokes 150
21.4 How a canvas is drawn 150
21.5 Text 151
21.6 Clips 152
21.7 SVG 152
22 Charts 154
22.1 A chart 154
22.2 The frame 155
22.3 Ticks 156
22.4 Series colors 157
22.5 Drawing 157
22.6 Hover and keys 157
22.7 Choosing a value 158
22.8 Legend 158
22.9 Animation 159
22.10 High contrast 159
22.11 Bar charts 160
22.12 Strip charts 161
22.13 Zoom and pan 162
22.14 Sparklines 162
23 Rendering 164
23.1 The draw list 164
23.2 One shape, one quad 165
23.3 Meshes 169
23.4 Drawing calls 169
23.5 From Clay to the draw list 172
23.6 Batching 174
24 Text 175
24.1 Fonts 175
24.2 Shaping and lines 177
24.3 Grapheme and word boundaries 178
24.4 Why not Clay's text 178
24.5 Caches 178
24.6 Drawing 179
24.7 The API 179
24.8 Selectable text 180
25 Backends 182
25.1 Roles a backend library plays 182
25.2 The interfaces 182
25.3 Choosing a backend 185

25.4 Kiln as a library ... 187
25.5 The raylib backend ... 187
25.6 The SDL3 backend ... 189
25.7 The software renderer and the headless platform ... 193
25.8 One thread ... 194
26 Glass ... 195
26.1 What Liquid Glass is ... 195
26.2 The WebGPU recreation ... 196
26.3 The render pipeline ... 196
26.4 Backends ... 196
26.5 Material stages ... 197
26.6 Legibility and accessibility ... 197
26.7 Performance ... 197
26.8 Integration ... 197
27 The operating system ... 199
27.1 One call, one implementation per OS ... 199
27.2 The system theme ... 199
27.3 The decimal separator ... 201
27.4 The calendar settings ... 202
27.5 Sorting text ... 203
27.6 Linux: the desktop portal ... 203
27.7 Windows and macOS ... 204
27.8 The web: the browser ... 204
27.9 Following the system ... 205
27.10 Overriding a setting ... 207
27.11 File dialogs ... 207
27.12 Notifications ... 209
27.13 Progress on the taskbar or Dock ... 209
27.14 Opening a URL ... 210
27.15 Trying it ... 210
28 Testing ... 211
28.1 The built-in demo ... 211
28.2 The game example ... 212
28.3 The gallery ... 212
28.4 Tests ... 214
28.5 Frame statistics ... 216
29 Performance ... 218
29.1 Where a frame's time goes ... 218
29.2 Benchmarks ... 220
29.3 The baseline ... 221
29.4 The counts test ... 225
30 C3 features in Kiln ... 227
30.1 The map ... 227
30.2 Containers: a macro around Clay's open and close ... 228
30.3 Typed state from a `$Type` parameter ... 229
30.4 Scoped ids: `defer` and `$if` together ... 229

1 Introduction

Kiln is a UI component library for the C3 language. Clay lays it out, and a backend draws it: raylib for windows, a software renderer for tests. You describe the interface every frame in plain C3 code, and Kiln keeps the state, motion and focus between frames. It serves apps, where Kiln owns the loop, and games, where Kiln draws menus and HUDs over your own raylib loop (App style and Game style). Its design follows CUI, a UI library whose components and conventions Kiln adapts to C3.

This book is written for programmers who know C3 or C. It explains how Kiln works and why it is built that way. Every public symbol is listed in a companion book, the Kiln API reference (`api.pdf`, built beside this one); a symbol's name in this book, such as `Ui.button`, links to its entry there. The README has the build commands, and CONTRIBUTING.md has the repository layout and the checks a change must pass.

```
fn void frame(Ui* ui, void* user)
{
    ui.@card(pad: 5, gap: 3)
    {
        ui.heading("Hello", 2);
        if (ui.button("Save", PRIMARY)) save();
    };
}
```

2 Overview

Kiln is an immediate-mode UI library for C3. Each frame your code calls widget functions on a `Ui` context; Kiln lays the result out with Clay, shapes the text, and hands a list of shapes to a backend that draws them. You keep your own state in ordinary variables, and Kiln keeps only what a widget needs between frames (hover, focus, an animation in flight).

Kiln serves two kinds of programs. When the UI **is** the program (a utility, a dashboard, an editor), Kiln owns the loop, opens the window and sleeps while nothing happens: this is App style. When the UI sits **on top of** something else (a game, a HUD, an engine tool), your raylib loop stays in charge and calls Kiln inside it: this is Game style. Both share one core, so widgets behave the same either way; App style and Game style explains them.

A Kiln app behaves like a native app on the system it runs on. It follows the user's settings: dark or light, the accent color, reduced motion, high contrast, and whether scrollbars stay shown. It also keeps following them when they change. It uses the system's own services for file dialogs, notifications and opening links, rather than drawing imitations. An app can override any setting, and a system without one gets a sensible default; The operating system describes how each system is asked.

2.1 The chapters

Chapter	Explains
The frame	What happens in one frame, from `begin()` to `draw()`
App style and Game style	App style and Game style, and how to choose
Widgets	How a widget is written, the core calls it uses, and the widget set
Input and focus	Focus, keyboard and gamepad navigation, shortcuts, how keys are routed
State and memory	What lives how long, widget state, allocators, why frames don't allocate
Motion	Springs, motion tokens, reduced motion, and idling
Theme	Tokens, colors, scales, and customizing the look
Layout	Sizes, containers, Clay's limits, responsive layouts
Rendering	The draw list, signed-distance-field (SDF) shapes, batching, and how Clay's render commands become the draw list
Text	Fonts, shaping, line breaking, bidirectional text, the glyph atlas and text caches
Backends	The `Platform` and `Renderer` interfaces and the backends that implement them
The operating system	What `kiln::os` asks the operating system, such as the system theme, and how each OS answers
Testing	The built-in demo, the examples, the gallery, and the tests, screenshot tests included
C3 features in Kiln	How C3's features map onto the design

2.2 Layers

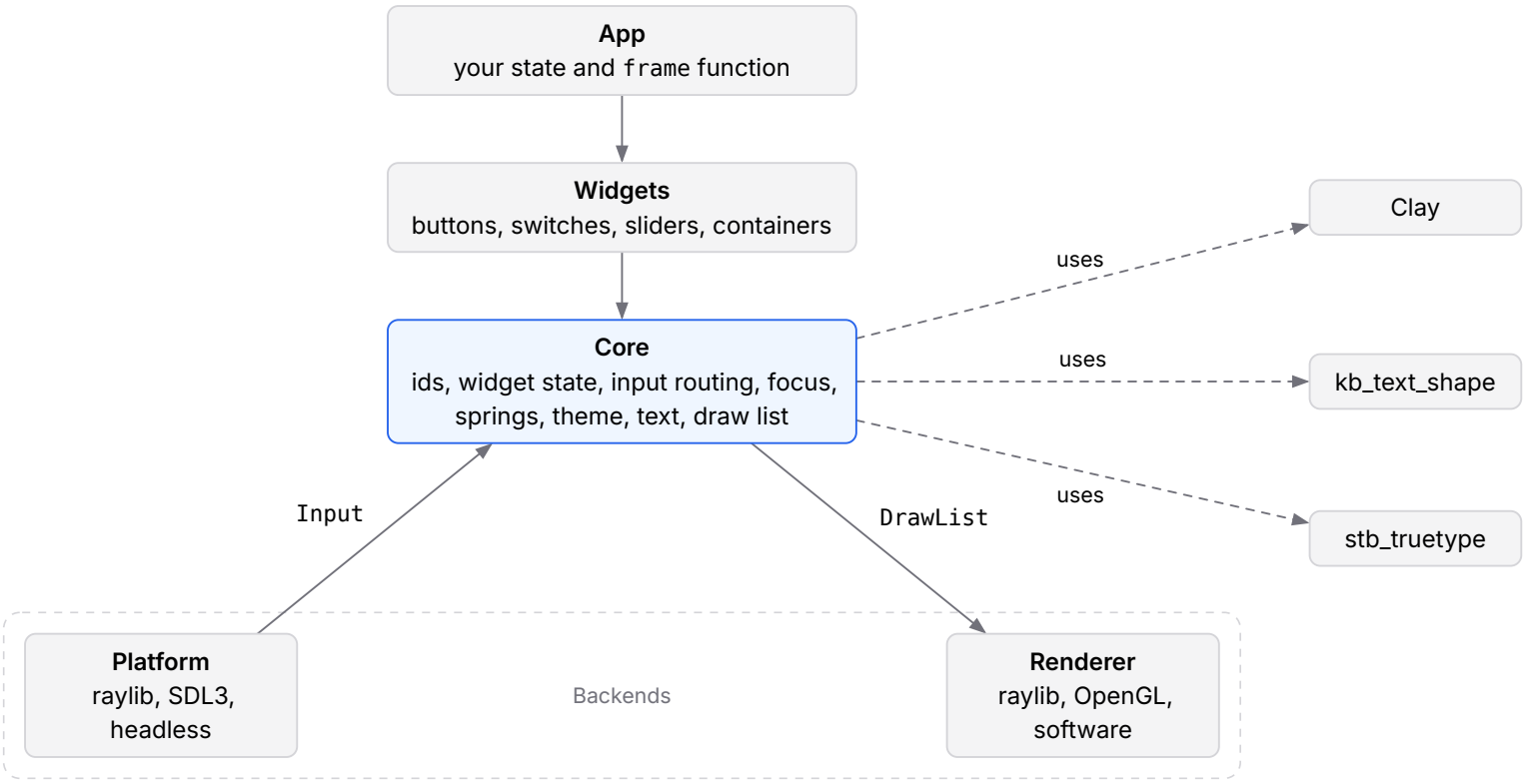


Figure 2: Kiln's layers. Your app calls widgets, widgets call the core, and the core uses Clay, kb_text_shape and stb_truetype. A Platform backend gives the core each frame's `Input` the core gives a Renderer backend a `DrawList`.

From the top down:

Layer	Owns	Depends on
App	Your variables (the app state) and the `frame` function	Widgets
Widgets	Buttons, switches, sliders, containers; each is a function on `Ui`	Core
Core	The `Ui` context: ids, widget state, input routing, focus, springs, theme, text, the draw list	Clay, kb_text_shape, stb_truetype
Backends	A `Platform` (window, input, time, clipboard) and a `Renderer` (drawing)	raylib, or SDL3 with OpenGL or SDL_GPU, for a window and the GPU; the software renderer and headless platform need neither, for tests

The core never calls raylib or SDL3. It consumes an `Input` value and produces a `DrawList`; backends supply the first and draw the second. Dear ImGui (platform and renderer backends) and Clay (render commands) draw the same line. Because the core sees only these two values, the same widgets run in a raylib or SDL3 window and in a test with no window and no GPU, where the headless platform plays a script of `Input` frames and the software renderer draws into memory. Backends describes the two interfaces and each backend.

2.3 Dependencies

- **Compiler:** C3, the 0.8.x line.
- **Clay:** layout, through the clay-c3 binding in `lib/clay.c3l` (module `clay::cl`), which compiles `clay.h` from source.
- **raylib:** a window and GPU backend, the one for Game style and the browser, and for App style when a target chooses it, built as a shared library from a pinned commit by `native/raylib`; the C3 binding in `lib/raylib6.c3l` is module `raylib6::rl`. Only the raylib backend and the targets built on it depend on it, never the core.
- **SDL3:** the App-style default backend's window and input, built static from a pinned release by `native/sdl3/build.sh`, with only the parts Kiln uses; the C3 binding in `lib/sdl3.c3l` is module `sdl3`, and `scripts/sdl3_symbols.sh` checks that the library defines every function it declares. OpenGL itself is not linked: the OpenGL renderer looks its functions up at run time through the platform. SDL_GPU reaches Vulkan, Metal or Direct3D 12 through SDL, with no graphics API linked directly. Only the SDL3 backend and the targets built on it depend on SDL3, never the core.
- **kb_text_shape** and **stb_truetype:** text segmentation, shaping and rasterization, from their single headers in `native/kb` and `native/stb`, compiled by `src/text/shim.c`. stb_truetype is compiled `STBTT_STATIC`, because raylib carries its own copy.
- **miniz:** PNG compression for the screenshot tests, through `lib/miniz.c3l`. Only test code imports it: the `kiln-tests` target lists it, so `c3c test` links it, and the `kiln` library carries none of it.

`THIRD_PARTY.md` at the repository root lists every third-party component with its version and license.

2.4 Modules and libraries

The public module is `kiln`. `import kiln;` brings in its submodules:

Module	Source	Holds
`kiln::ui`	`src/ui/`	The `Ui` context, the core calls, the widgets (`src/ui/widgets/`), the draw list and the backend interfaces
`kiln::text`	`src/text/`	Fonts, shaping and the glyph atlas, with `shim.c` as the C side, and the `Icon` enum
`kiln::themes`	`src/themes/`	The color palette and the theme presets
`kiln::raylib`	`backends/raylib/`	The raylib platform and renderer, and `raylib::create_ui` for Game style
`kiln::sdl`	`backends/sdl3/platform.c3`	The SDL3 platform, `SdlPlatform`, and `sdl::create_ui`
`kiln::gl`	`backends/gl/gl.c3`, `renderer.c3`	The OpenGL renderer, `GlRenderer`, and the

Module	Source	Holds
		OpenGL functions it looks up at run time
`kiln::sdlgpu`	`backends/sdlgpu/`	The SDL_GPU renderer, `SdlGpuRenderer`, and its SPIR-V shader
`kiln::glsl`	`backends/gl/shader.c3`	The shader the raylib and OpenGL renderers draw with (One shape, one quad)
`kiln::app`	`backends/app/common.c3` and each backend's `app.c3`	The App-style loop, `app::run`: what every backend shares in `common.c3`, and each backend's own loop in `backends/raylib/app.c3` or `backends/sdl3/app.c3`
`kiln::software`	`backends/software/`	The software renderer and the headless platform

The core (`src/`) never imports a backend. Code that names rectangles imports `std::math` for `Rect`; the core uses the standard library's type.

`project.json` builds four static libraries, all from the core's sources (`src/**`, with `src/text/shim.c` and Clay):

- `kiln`: the core, plus miniz for the tests.
- `kiln-raylib`: the core plus `backends/raylib`, `backends/gl` and `backends/app`, depending on `raylib6`.
- `kiln-sdl3`: the core plus `backends/sdl3`, `backends/gl` and `backends/app`, depending on `sdl3`.
- `kiln-software`: the core plus `backends/software`, with no native dependency beyond the core's.

An executable compiles Kiln from source rather than linking one of these libraries. The `game` target, like the other raylib targets, adds `backends/raylib/**`, `backends/gl/**`, `backends/app/**` and its own directory to the sources, depends on `raylib6`, and links `libraylib` from `build/raylib` (statically on Linux, with unused sections dropped, so the executable needs nothing beside it):

```
"game": {
  "type": "executable",
  "sources": [ "backends/raylib/**", "backends/gl/**", "backends/app/**", "examples/
game/**" ],
  "dependencies": [ "raylib6" ],
  "linker-search-paths": [ "build/raylib" ]
}
```

The App-style examples on SDL3 (`hello`, `gallery`, `dashboard` and their `-release` and `-system` targets) list `backends/sdl3/**` in place of `backends/raylib/**` and depend on `sdl3`; all but the `-system` targets link `libSDL3.a` from `build/sdl3`. Their `-raylib` targets list `backends/raylib/**`, as `game` does. The example's own sources are the same, and an app carries one backend, never both (Choosing a backend).

Each example also has a `-release` target, the same sources built at O3 without panic messages: what an app ships, and what `scripts/bench.sh` measures (Performance). The plain targets build at O0 with every safety check, for debugging and the tests.

The `dashboard` target is the largest example: a music service's analytics in App style, run with `c3c run dashboard`. A seeded generator makes a year of data in a `Ui.job` while skeletons and the taskbar show progress. A sidebar switches between Overview, Tracks, Listeners and Settings, and a toolbar holds the filters, Refresh, and an Export menu that saves the plays chart as SVG through `Ui.canvas_svg` or the tracks as CSV, through the system's save dialog. Overview lays metric cards with sparklines in a `Ui.@grid` over a zoomable line chart, stacked genre areas and a top-tracks bar chart whose bars open a track sheet; Tracks is a sortable, searchable table that shows fewer columns on a phone; Listeners has a live strip chart; and Settings pins or follows high contrast and reduced motion and shows what the OS reported.

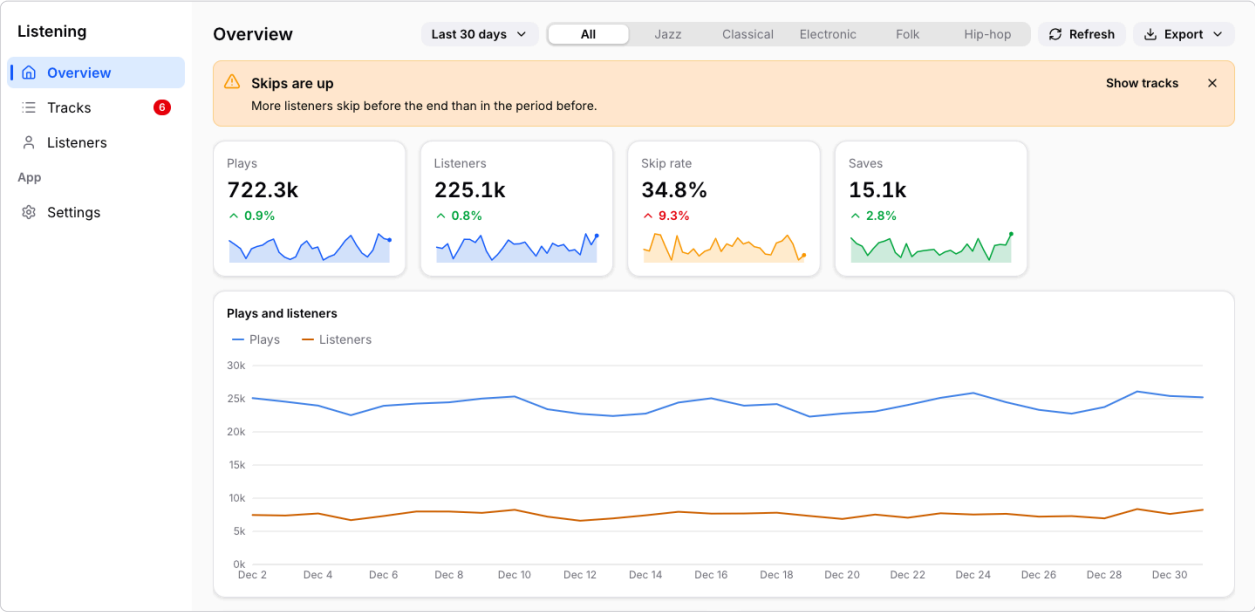


Figure 3: The dashboard's Overview page: the sidebar with the Tracks badge, the toolbar of filters and actions, the alert raised when skips rise, the metric cards with their sparklines, and the plays and listeners chart.

The tests run with `c3c test`, which builds the first target, `kiln-tests` (the library with miniz), and adds the test sources (`test/**`, `backends/software/**`, the gallery's `examples/gallery/pages.c3` and `switcher.c3`, and the dashboard's `examples/dashboard/data.c3`, `app.c3` and `pages.c3`). They draw with the software renderer, so they need no window.

3 The frame

Kiln is **immediate mode**: your app describes its whole interface every frame, and widgets are function calls that return what happened. There is no tree of widget objects to build, update or tear down. A button exists in a frame because you called `Ui.button` in that frame, and `ui.button()` returns `true` in the frame it was clicked.

Every frame is the same four calls on the `Ui`: `Ui.begin`, your frame function (in Game style, the widget code in your own loop), `Ui.end` and `Ui.draw`. Who makes those calls, Kiln's loop or yours, is the subject of App style and Game style; what they do is the same either way.

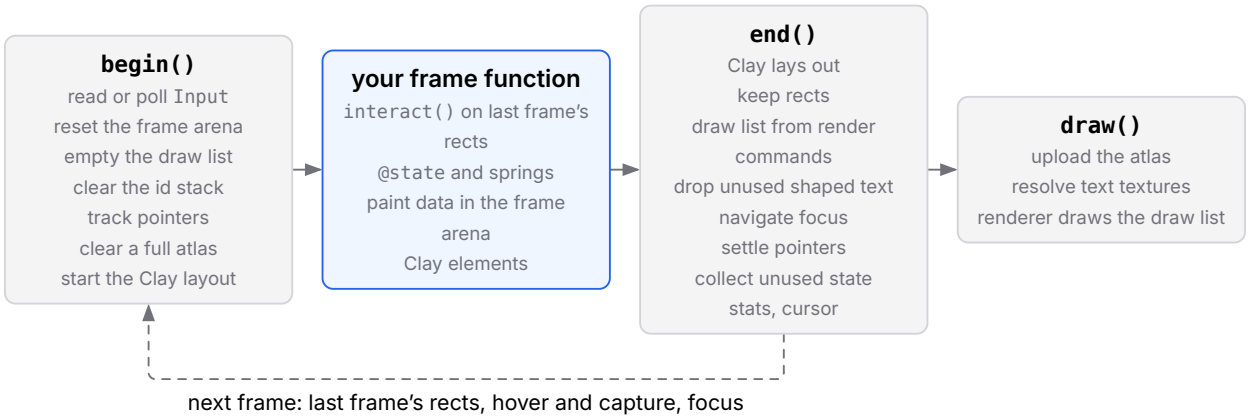


Figure 4: One frame. Your frame function runs between `begin()` and `end()`; `draw()` can run later, inside the backend's own drawing. What `end()` settles is what the next frame's widgets read.

3.1 `begin()`: take in the input

`begin()` starts the frame. You can hand it an `Input` (a test or a custom platform does); without one, it clears its `Input` and asks the attached `Platform` to poll into it, passing along whether the poll may sleep until something happens. From the input it takes the time step, the time (adding up `dt` when the input carries no clock), the layout size and the pixel scale; a zero size or scale keeps the last one.

Then it resets everything that belongs to one frame:

- The **frame arena** is reset, so the paint data of the last frame is gone (see State and memory).
- The draw list is emptied.
- The id stack and the set of ids seen this frame are cleared.
- Pointers are folded into their tracks: which pointer went down or up this frame, where a press started, and which pointers vanished without a release.
- This frame's focus stops, scopes and groups, the keys and gamepad buttons widgets used, and the shortcuts asked for are cleared (see Input and focus).
- If the glyph atlas filled up during the last frame, it is cleared now, and glyphs are rasterized again as text asks for them (see Text).
- The cursor goes back to an arrow, and the `Ui` becomes this thread's current one, which is how springs find the frame's time step.
- Clay starts a layout at the new size. If the last frame needed more elements than Clay had room for, Clay is first re-created with a larger budget (see Layout).

3.2 Your frame function: describe the interface

Between `begin()` and `end()` your code calls widgets. Each one reads its input against last frame's rectangle, fetches its state and moves its springs, fills paint data in the frame arena and declares Clay elements; The five-step pattern walks through the switch step by step.

Nothing is drawn yet, and no rectangle is known for this frame: the frame function only describes.

3.3 `end()`: lay out and settle

`end()` finishes the frame, in this order:

1. **Layout.** Clay lays out the elements. For every element that has an id, its box goes into the rect table, where the next frame's `Ui.interact` finds it. Clay's render commands then become the draw list: rectangles, borders and clips directly, and custom elements by calling their paint callbacks with their final rectangles (see Rendering). A frame that overflowed Clay's budget skips this step and gets its room at the next `begin()` (see Layout).
2. **Text.** The font cache counts the frame and, every sixty frames, drops shaped text that has not been used for a while (see Text).
3. **Navigation.** Escape, Tab, the arrows and the d-pad that no widget used move focus or close the top layer; then focus is settled: a layer that disappeared hands focus back to its opener, a new layer takes it, and a focused widget that is gone passes it to a neighbor (see Input and focus).
4. **Pointers.** Each pointer's candidate becomes its hovered or capturing widget (The core calls).
5. **Collection.** State and rectangles of widgets that stopped being called are freed (see Widget state).
6. **Stats.** The frame's numbers are kept for `Ui.stats` (see Frame statistics).
7. **Cursor.** If a hovered widget asked for another cursor, the platform is told.

`end()` also checks that every `Ui.push_id` was popped and every container, focus scope and group was closed.

3.4 `draw()`: hand over the draw list

`draw()` uploads the glyph atlas pages that changed, gives the draw list's text commands their textures, and passes the draw list to the attached `Renderer` with the layout size and pixel scale. It draws the frame `end()` built, so it can run later than `end()`, inside the backend's own drawing (see Rendering and Backends). Without a renderer, it does nothing.

3.5 What survives between frames

The interface itself does not survive: next frame, your code describes it again. What carries over is what the widgets cannot rebuild from your variables:

- each widget's rectangle from the last layout, which `interact()` reads;
- each widget's state, such as its springs, scroll offset or text edit;
- the pointers' hover and capture, and keyboard focus with last frame's focus stops.

Alongside those, the `Ui` keeps its caches (shaped text, the glyph atlas) and the theme. Rectangles and state are keyed by id and collected once a widget stops being called (see Widget state).

Because `interact()` reads last frame's rectangles and `end()` settles hover and capture for the next frame, a press reaches its widget one frame after the input arrives, against the layout the user was looking at.

3.6 Compared with React

	React	Kiln
When it runs	When state or props change	Every frame
Output	Virtual DOM, then a diff	Clay elements for this frame
Events	Callbacks	Return values
Local state	`useState`, matched by hook order	`Ui.@state`, keyed by id
App state	Lifted state, context, stores	Your variables, passed by pointer
Lifecycle	Mount, unmount, effects	None

3.7 A custom component is just a function

A component of your own is a function that takes the `Ui` and calls other widgets. It needs no registration and no type of its own; like a widget, it can return what happened, and it keeps its state in your variables or in `@state`:

```
fn void counter(Ui* ui, String label, int* count)
{
    ui.@id(label)
    {
        ui.@row(gap: 2, align: CENTER)
        {
            ui.textf("%s: %d", label, *count);
            if (ui.button("+")) (*count)++;
        };
    };
}
```

`Ui.@id` gives the widgets inside their own ids, so two counters on one page keep separate state (see Widgets).

4 App style and Game style

Kiln runs in two loop styles that share one core, so a widget behaves the same in either. The only difference is who owns the loop: Kiln, or you. The rule for choosing is short: **if the UI is the app, use App style; if the UI sits on top of something else, use Game style.** Music players, dashboards, utilities and editors are App style. Games, HUDs, in-game menus and engine tools are Game style.

Both styles run the same frame (The frame): `Ui.begin`, your widget calls, `Ui.end`, then `Ui.draw`. They differ in where that sequence lives and in what happens around it.

4.1 App style: Kiln owns the loop

`app::run` (module `kiln::app`) opens a window, creates a `Ui` for it, and calls your frame function every frame between `begin()` and `end()`. On the desktop it returns when the window closes, or after the frame that calls `Ui.request_quit` in a browser it never returns (On the web).

Each App-style backend has its own `app::run`, with the same signature: `backends/sdl3/app.c3` opens an SDL3 window and draws with SDL_GPU on Linux by default, or OpenGL elsewhere and as the fallback, and `backends/raylib/app.c3` opens a raylib window (Backends). Which one an app gets is a build choice, not a code change: the `hello`, `gallery` and `dashboard` targets (with their `-release` variants) build with SDL3, the App-style default, and `hello-raylib`, `gallery-raylib` and `dashboard-raylib` (with theirs) build the same sources with raylib. What both share, the function types below, `KILN_WINDOW`, the screenshot and the frame times, is in `backends/app/common.c3`. Game style and the browser use raylib (Game style: your raylib loop, On the web).

```
alias FrameFn = fn void(Ui* ui, void* user);
<* A function `app::run` calls while drawing a frame, before or after `Ui.draw`, with
the `user` pointer given to `app::run`. *>
alias DrawFn = fn void(Ui* ui, void* user);
```

```
fn void run(String title, int width, int height, FrameFn frame, void* user = null, Theme
theme = themes::SYSTEM,
    DrawFn draw_under = null, DrawFn draw_over = null, bool resizable = true, FontSet
fonts = {}, String app_id = "",
    char[] icon = {}, bool title_bar = false, BackgroundFn background = null)
{
```

- `frame` is called once per frame, with `user` passed through untouched: point it at your app state.
- `theme` picks the theme the `Ui` starts with (Theme). The default, `themes::SYSTEM`, follows the operating system: its dark or light setting and accent, kept up to date while the app runs (Following the system). Pass any other theme to override it.
- `draw_under` and `draw_over` are draw hooks for custom drawing: `draw_under` runs after the window is cleared and before the UI is drawn (backgrounds, custom scenes), `draw_over` after it.
- `resizable` makes the window resizable.
- `app_id` is the name of the app's `.desktop` file without `.desktop`, such as `"glaze"` for `glaze.desktop`. Wayland compositors and GNOME match a window to its `.desktop` file by this id, and take its icon, name and dock entry from there; X11 desktops do the same through the window's class. Without it, a window shows a generic icon and isn't grouped with the app's launcher. SDL3 sets it before it starts (`SDL_SetAppMetadata`). raylib starts GLFW and opens its window in one call, leaving no point to set it, so an app that needs its id uses SDL3.
- `icon` is a PNG or JPEG, typically `$embed`-ded, decoded with `os::window_icon` and given to the window: X11 shows it in the window menu and alt-tab, and Windows in the taskbar when the

executable has no icon of its own. On Wayland, most desktops show the `.desktop` file's icon instead, which is why `app_id` matters more there. Bytes that don't decode print a warning, and the window opens without an icon. The web ignores both: a page's icon and name are its own (`<link rel="icon">`, its manifest).

- `title_bar` draws Kiln's own title bar on Wayland and skips libdecor, the library SDL otherwise uses to draw one there. Elsewhere the OS title bar stays and this is ignored. The bar is a `ui::TITLE_BAR_HEIGHT`-point strip with the title and the window's minimize, maximize or restore, and close buttons; The title bar Kiln draws has the details.

Around your frame function, the loop clears the window with the theme's `SURFACE` color before drawing, and Escape reaches your widgets instead of closing the window. On SDL3 it clears with `gl::clear_to` and presents with `SDL_GL_SwapWindow`, which waits for the display's refresh. SDL skips that wait on Wayland while the compositor isn't showing the window, and a driver setting can turn vsync off, so `SdlPlatform.present` also paces frames itself: it returns no sooner than the display's refresh period, less a tenth for timing jitter, after the last frame, and an animation never draws as fast as the CPU can. raylib's loop paces frames too, at a fixed 60 frames per second, and turns off raylib's exit key. Recording frame times (`KILN_FRAME_TIMES`, below) leaves frames unpaced on both.

The `hello` example is a whole App-style program. Its `main` builds the app state and hands it to `app::run`:

```
fn void main()
{
    Demo demo = { .enabled = true, .sound = true, .volume = 60, .recent = -1, .show_ring
= @ok(env::tget_var("KILN_SCREENSHOT")) };
    app::run("Kiln: hello", 960, 600, &frame, &demo);
    demo.name.free();
    demo.query.free();
}
```

and its frame function receives that state back through `user`:

```
fn void frame(Ui* ui, void* user)
{
    Demo* demo = user;
    // The sound dialog answers once; until then result() reports os::PENDING.
    if (try paths = demo.sound_dialog.result())
    {
        if (paths.len)
        {
            demo.sound_file.free(mem);
            demo.sound_file = file_name(paths[0]).copy(mem);
        }
        os::free_paths(paths);
    }
    for (int i = 0; i < PAGES.len; i++)
    {
        if (ui.shortcut(Key::from_ordinal(Key.D1.ordinal + i), primary: true))
demo.selected = i;
    }
    // For screenshots: show the focus ring that keyboard use would show.
    if (demo.show_ring) ui.focus_visible = true;
    // The sidebar runs along the window's edge; the content keeps its margin.
    ui.@stack(ui.at(MD, Dir.ROW, Dir.COLUMN), width: ui::GROW, height: ui::GROW)
```

```
    {
        sidebar(ui, demo);
        ui.@column(width: ui::GROW, height: ui::GROW, pad: 6) { cards(ui, demo); };
    };
    ui.toasts();
}
```

4.1.1 The title bar Kiln draws

An App-style app can ask for Kiln's own title bar:

```
app::run("Glaze", 1280, 800, &frame, &app, title_bar: true);
```

On Wayland, `app::run` then opens a window without the OS title bar, skips libdecor, and puts a Kiln title bar above the app's frame. The bar is `ui::TITLE_BAR_HEIGHT` points tall, uses the app's theme and its focus state, and returns a `WindowAction` when one of its buttons is pressed. A right-click on the bar returns `MENU`, and `app::run` opens the system's window menu at the pointer (`SDL_ShowWindowSystemMenu`), as a native title bar does. On GNOME that menu has "Always on Top", which a Wayland app can't set itself. The app's frame builds below it, so its own content keeps the rest of the window.

Double-clicking any empty part of the bar maximizes or restores the window. The strip left of the window buttons drags it; its first press waits for a few points of motion, so a second press can maximize instead of moving. The first six points of each edge resize the window. The hit test is in the SDL platform: the strip becomes a drag region, the buttons stay normal, and the edges become resize handles. The minimize, maximize or restore, and close buttons are widgets: they take focus, respond to Enter and Space, and show tooltips. The close button sets the app's quit flag; the other two call SDL's window functions. The title area shrinks when the window is narrow, so the three buttons stay on screen.

Elsewhere the OS title bar stays. The bar holds the title and the window buttons; the app's frame starts below it.

4.1.2 Sleeping when idle

An App-style app uses no CPU while nothing happens. After each frame, the loop checks whether the frame was idle: no spring moved (`ui.animating`, see Motion) and no input event arrived (`ui.input.events`). If so, the loop sleeps: on SDL3 it sets `ui.wait_for_events`, and the platform's poll at the next `begin()` sleeps in `SDL_WaitEventTimeout` until the next event or the deadline in `Ui.wake_time`, rounded up to whole milliseconds. raylib's loop sleeps too, as far as raylib lets it: it turns on event waiting, and `EndDrawing` sleeps until the next input event, with no deadline (below). Any other frame keeps the loop awake, so animations run at the full frame rate.

Two things that aren't input also need a frame: a deadline a widget asked for with `Ui.wake_after` (Pointers and layers), and a background job (Background work) finishing on a worker thread. `Ui.begin` passes the deadline to the platform's `poll`, and a worker calls the platform's `wake()` when its function returns. What happens next depends on `Platform.wakes()` (`Ui.platform_wakes()`), which says whether `wake()` really ends a waiting poll from another thread and whether `poll` stops waiting at its deadline. SDL3 does both: its `wake()` pushes an event, so the loop sleeps through a deadline or a running job and draws a frame when it comes. raylib does neither: its wait has no timeout and its `wake()` does nothing, so at `end()` the `Ui` counts as animating until the deadline has passed or the job is done, and the loop draws frames until then. A job that waits to run on the UI thread, when the `Ui` has no worker threads, keeps frames coming on every platform, since nothing else would run it. `end()` also lets go of jobs nobody has asked about for `ui::STATE_DATA_SECONDS`, as

widget state is let go (Widget state), and `begin()` runs the jobs that wait for the UI thread when the `Ui` has no worker threads. A Game-style loop needs none of this: it draws every frame anyway, and the same `begin()` and `end()` tend its jobs.

4.1.3 Environment variables

`app::run` reads five environment variables, for checking what an app draws without a person looking at it, for trying other window sizes, and for measuring real frames:

Name	Effect	Default
`KILN_SCREENSHOT`	A `.png` path. The app saves the screen to it after the frame given by `KILN_SCREENSHOT_FRAME` is shown, prints `kiln: saved <path>` to standard error, and exits. While a screenshot is pending, the loop never sleeps.	unset: no screenshot
`KILN_SCREENSHOT_FRAME`	The frame to save, counted from the first `begin()`. A value that isn't a number counts as 10.	10
`KILN_WINDOW`	`<width>x<height>`, such as `390x844`: opens the window at that size instead of the one passed to `app::run`. A malformed value or a size that isn't positive is ignored.	unset: the size passed to `app::run`
`KILN_FRAME_TIMES`	A `.csv` path. Frames come back to back, with no idle sleeping and no frame cap; the app records each one's build, draw and present times and the startup time (from `app::run` to the end of the first presented frame), writes them to the file and exits (Real frames on a GPU).	unset: no recording
`KILN_FRAME_TIMES_FRAMES`	The frames to record. A value that isn't a number counts as 300.	300

The screenshot logic is public, so a Game-style loop can use it too: `app::screenshot_request` reads the two screenshot variables into a `Screenshot`, and `Screenshot.due(ui.frame)`, called after `EndDrawing`, saves the screen on the right frame and returns `true` when the app should exit. The `overlay` and `game` examples do this.

4.2 Game style: your raylib loop

In Game style you keep your own raylib loop, and Kiln is one more thing it updates and draws. After `InitWindow`, `raylib::create_ui` returns a `Ui` with the raylib platform and renderer attached (Backends):

```
fn Ui* create_ui(Allocator allocator = mem, Theme theme = themes::SYSTEM, FontSet fonts
= {})
{
    require_window();
    Ui* ui = ui::create(allocator, theme, fonts: fonts);
    ui.platform = new_platform(allocator);
    attach_renderer(ui, allocator);
    return ui;
}
```

Each frame you call `ui.begin()`, build widgets, call `ui.end()`, and then call `ui.draw()` between your own `BeginDrawing` and `EndDrawing`, wherever the UI belongs in your draw order. The `overlay` example is the canonical Game-style loop. It sets up the window and the `Ui`:

```
fn void main()
{
    rl::set_config_flags(RLConfigFlag.WINDOW_RESIZABLE);
    rl::init_window(1000, 640, "Kiln: overlay");
    defer rl::close_window();
    rl::set_target_fps(60);

    Ui* ui = raylib::create_ui(theme: themes::DARK);
    defer ui.destroy();
    app::Screenshot shot = app::screenshot_request();
    defer shot.free();
```

and then runs its scene, a field of spinning squares, with a panel on top:

```
    while (!rl::window_should_close())
    {
        ui.begin();
        ui.@column(width: ui::GROW, height: ui::GROW, pad: 6, align: END)
        {
            ui.@card(width: ui::px(280), dir: ROW, gap: 4, align: CENTER, shadow: LG)
            {
                if (ui.icon_button(playing ? Icon.PAUSE : Icon.PLAY, "play", PRIMARY,
LG)) playing = !playing;
                // After the button, so a focused button gets Space first.
                if (ui.shortcut(SPACE) || ui.shortcut(P)) playing = !playing;
                ui.@column(width: ui::GROW, gap: 1)
                {
                    ui.text(playing ? "Playing" : "Paused", weight: SEMIBOLD);
                    ui.textf("%d squares · click to add", squares.len());
                    ui.checkbox("Spin", &spin);
                    ui.slider("Speed", &speed, 0.25f, 3, 0.25f, format: "%.2f×",
disabled: !spin);
                };
            };
        };
        ui.end();

        if (rl::is_mouse_button_pressed(LEFT) && !ui.wants_pointer())
        {
            squares.push({ rl::get_mouse_position(), 40, 0 });
        }
```

```
        if (playing && spin)
        {
            foreach (&s : squares) s.spin += rl::get_frame_time() * (60 + s.size) *
speed;
        }

        rl::begin_drawing();
        rl::clear_background({ 24, 24, 27, 255 });
        foreach (i, s : squares)
        {
            char shade = (char)(90 + (i * 23) % 120);
            rl::draw_rectangle_pro({ s.pos.x, s.pos.y, s.size, s.size }, { s.size / 2,
s.size / 2 }, s.spin, { 60, shade, 200, 255 });
        }
        ui.draw();
        rl::end_drawing();

        if (shot.due(ui.frame)) break;
    }
```

You control the loop, its timing and the draw order. What Kiln adds:

- **Input.** `ui.begin()` reads input from raylib through the platform, including `GetFrameTime()` for the frame's time step. `ui.begin(&input)` takes an `Input` you fill instead (`begin()`: take in the input).
- **Drawing that composes.** `ui.draw()` flushes raylib's batch first, so what you drew before it stays underneath, and hands raylib its state back afterwards (Backends lists exactly what it resets, including that face culling stays off).
- **Keeping UI input out of the game.** Call these after `ui.end()`. `Ui.wants_pointer` is true while any pointer (mouse, pen or touch) is over a widget or a widget holds it (during a drag, for instance); the overlay uses it so a click on the panel doesn't add a square. `Ui.wants_keyboard` is true while a widget has keyboard focus with its focus ring showing, or when a widget or shortcut took a key this frame; the game example stops reading movement keys while it is true.
- **Idle sleeping, if you want it.** A Game-style `Ui` draws every frame, as games do. Set `ui.wait_for_events = true` on frames where nothing moves: the next `begin()` then turns on raylib's event waiting, and that frame's `EndDrawing` sleeps until input arrives. Setting it back to `false` turns event waiting off again at the next `begin()`.

The `game` example (`examples/game`) is a whole small game in this style, whose menus are focus scopes, so focus lands on their first button when they open; Testing describes the examples and Focus describes focus scopes.

4.3 On the web

In the browser, Kiln runs on raylib, which builds for it where the SDL3 backend does not: C3 compiles to WebAssembly through Emscripten (c3c's `emscripten` target), and raylib draws with WebGL. The `gallery-web` and `dashboard-web` targets in `project.json` build the gallery and the dashboard this way, and `scripts/web.sh` builds both, the gallery into `build/web` (`index.html`, `index.js`, `index.wasm` and the `fonts` it downloads for other scripts) and the dashboard into `build/web/dashboard`, prints the download sizes, and with `--serve` serves them on localhost. raylib itself is built for the web by `native/raylib/build.sh --web`, into `build/raylib-web`. The same Kiln code runs on both: what differs is below.

4.3.1 The browser owns the loop

A page can't run a `while` loop of its own: the browser draws, and delivers input, only between calls into the page. So under Emscripten, raylib's `app::run` opens the canvas and creates the `Ui` as on the desktop, then hands one frame at a time to the browser with `emscripten_set_main_loop_arg`, which calls it on each `requestAnimationFrame`. Each call runs the same frame as the desktop loop: `begin()`, your frame function, `end()`, then the draw.

- `app::run` returns before the first frame and never returns again, so its `user` pointer must outlive `main`: make the app state a global, as the gallery's `main` does. Code after `app::run` doesn't run on the web.
- The browser sets the pace: one frame per display refresh while the tab is visible, none while it is hidden. There is no idle sleeping, and no 60 frames per second cap.
- A Game-style loop can't run as written either. Move its body into a function and hand that to `emscripten_set_main_loop_arg`; or link with `-sASYNCIFY`, which lets a `while` loop yield to the browser in `EndDrawing`, at a cost in download size and speed.

4.3.2 The canvas

The page is `backends/raylib/shell.html`: one canvas filling the window, a background that follows `prefers-color-scheme` so the page doesn't flash white before the first frame, and no browser context menu on the canvas, so a right-click reaches Kiln's. raylib sizes the canvas in CSS pixels; at the start of each frame the platform sizes it again to its CSS size times `devicePixelRatio` and reports that ratio as the `Ui`'s scale, so text is drawn in the screen's own pixels. Browser zoom changes `devicePixelRatio`, so zooming scales the whole UI.

The browser composites the canvas over the page by its alpha channel, so the renderer blends alpha as "over" (one for the source, one minus the source's alpha for the destination): the framebuffer stays opaque wherever the background is, and translucent shapes keep their colours.

4.3.3 Input

A browser may have a mouse, touch or both. The platform reads touch points while a finger is down and the mouse otherwise. After a touch, browsers also send the mouse events older pages expect; reading those as well would click twice, so the mouse is read again only once fingers have been off the screen for a moment.

4.3.4 Background work

Web builds have no threads: threads in a browser need the page served with cross-origin isolation headers, which many hosts can't set. So `Ui.job` runs every job on the UI thread at the next `begin()`, as with `Ui.set_job_threads(0)` (Background work). A job then holds up the frame it runs in, so keep the work of one job short, or split it into several. The gallery's Background page runs smaller jobs on the web for this reason.

4.3.5 The browser's services

The browser stands in for the operating system. `kiln::os` asks it through one JavaScript library, `src/os/web.js`, which the web targets link with emcc's `--js-library`: the theme follows the browser's media queries and its font size, files open into the page's memory and save as downloads, and notifications, links and sorting text use the browser's own (The operating system). A page can't reach the system's fonts, so a web build downloads fallback fonts for other scripts when its text needs them (Text).

A page reads the clipboard only when it receives a paste, so the platform keeps the last text the page copied or pasted: Ctrl+V brings the system clipboard's text, and a menu's Paste the last text copied or pasted.

5 Background work

Some work takes longer than a frame: loading a file, decoding an image, searching a library, counting. Done inside the frame function, it would freeze the window until it ends. `Ui.job` runs such work on a worker thread instead, and the frame asks each time how it is doing: still running, done with a result, or failed. This chapter describes the threading model, the calls, who owns what, and how the window learns that a job has finished.



Figure 5: Worker threads run the jobs and leave their results in the job's record; the UI thread reads them when a frame asks, and a finished job wakes the platform so that frame comes soon.

5.1 One UI thread

The `Ui` stays on one thread. Every widget call, the layout, the state between frames and the draw list belong to the thread that calls `Ui.begin` and `Ui.end`, and none of it is locked. A job never touches it: its function sees only its own `Job`, which carries the app's input in and the result out. So a job can't build widgets, read widget state or raise a toast; it computes a value, and the next frame turns that value into interface.

This keeps the cost of threads in one place. The only data both threads touch is the job's record and the app's input. The fields that change while both hold the record are atomics: the status, the progress, the cancel flag and the count of holders. The result and the fault are plain fields, written before the status leaves `PENDING` and read only after.

5.2 The calls

A job's work is a `JobFn`, `fn void?(Job* job)`. It reads `job.data`, does the work, and stores what it made in `job.result`, allocated in `job.allocator`. Returning a fault fails the job. The function sees these fields of `Job`:

Field or method	Meaning
`data`	the app's input, as passed to `Ui.job`
`result`	what the function made; the UI reads it once the job is done
`allocator`	the allocator for the result, safe from any thread (Memory)
`progress`	0 to 1, set with `Job.set_progress(done)`, which clamps it, when the function can tell how far along it is
`Job.cancelled()`	true once the UI no longer wants the result: long work checks it now and then and returns early

The frame starts and follows a job with one call:

```
JobState s = ui.job("primes", &count_primes, &g.limit, key: g.run, free_result: null);
```

The first call with an id starts the job; later calls with the same id and `key` only report it. The returned `JobState` is a copy made for this frame:

`status`	Meaning
`PENDING`	queued, or running; `progress` says how far it has got
`DONE`	finished; `result` is the function's result
`FAILED`	the function returned a fault, which is in `error`

The status is a `JobStatus`. A job is never restarted on its own: a `DONE` or `FAILED` job keeps reporting so, frame after frame, until the app asks for a new one.

5.2.1 Keys: a job per input

`key` is a hash of what the job reads, such as a search query or a file name, and `0` by default. When a frame passes a different `key` under the same id, the UI lets go of the old job, which cancels it, and starts a new one. A search box therefore needs no bookkeeping: each frame passes the hash of the current query, and only the job for the latest query survives. A counter works as a key too: the gallery increments one each time the user asks to run a job again.

`Ui.job_restart(id)` lets go of a job without starting another; the next `Ui.job` call with that id starts it again. A Cancel button calls it, and stops asking for the job.

5.2.2 Taking the result

A `DONE` job's `result` belongs to the job: the UI frees it when it lets go of the job, so the app may read it in every frame but must not keep the pointer. To keep the result past the job, `Ui.job_take(id)` hands it over: it returns the pointer once, and from then on the app owns it and frees it, and `JobState.result` is null. It returns null for a job that isn't done, or whose result was already taken.

5.3 Lifetime

A job's record is shared by the UI and the worker that runs it. Neither can free it alone: the UI may stop wanting a job that is still running, and a worker may finish a job the UI has already forgotten. So the record keeps an atomic count of its holders, the UI and the worker, and whichever lets go last frees it, with its result. A `JobFree` passed as `free_result` frees the result and whatever it owns; without one, the result is freed as one block in `job.allocator`. A result the app took is the app's, and is left alone.

The worker lets go when the function returns. The UI lets go in three cases, without being asked in the third:

- the job is restarted with `Ui.job_restart`
- a different `key` replaces it;
- no frame has asked about it for `ui::STATE_DATA_SECONDS`, as unused widget state is let go (Widget state). A page the user left takes its jobs with it.

Letting go of a running job sets its cancel flag: `Job.cancelled` turns true, and a function that checks it returns early. A job still in the queue when it is let go is never run. Either way its result, if any, is freed when the worker lets go.

`Ui.destroy` ends every job before anything else: it cancels them all, drops the ones still queued, and joins the workers, waiting for running functions to return. Then it frees the records. A long job that never checks `Job.cancelled` therefore delays closing the window by as long as it takes; the examples' jobs sleep in short naps and check between them for this reason.

5.4 Memory

A job's result is made on one thread and freed on another, so it can't come from the frame arena or the temp allocator, which belong to one thread and one frame (Allocators). `ui::job_allocator` is the allocator jobs use: the C heap where there is one, which is safe from any thread. `job.allocator` is that allocator, and a result taken with `Ui.job_take` is freed with it:

```
if (try long* count = (long*)ui.job_take("primes"))
{
    g.count = *count;
    alloc::free(ui::job_allocator(), count);
}
```

`job.data` is the app's, and Kiln never copies it. The app keeps it alive and unchanged while the job runs: a worker may read it at any time until the function returns. Data that changes while the user types is copied into the job's own block first, or passed as an immutable value, and its hash becomes the `key`.

An app that wants to let go of a job without waiting for it, as `Ui.image_load` does when an image is destroyed while it decodes, gives the job its input: `data` allocated with `ui::job_allocator()` and a `free_data` (a `JobFree`) that frees it. The job then frees its input with its record, whichever side lets go last, so the input outlives a worker still reading it and never outlives the job.

5.5 The worker pool

The workers are a `std::thread::ThreadPool`, made on the first job, so an app that runs no jobs starts no threads. The pool starts up to `clamp(cpus - 1, 1, MAX_JOB_THREADS)` workers as work arrives, one less than the CPUs so the UI thread keeps a core, and at most `ui::MAX_JOB_THREADS`, 4. Jobs beyond that wait in the pool's queue. `Ui.set_job_threads(n)` sets another count, before the first job starts.

With `set_job_threads(0)`, or where threads can't start, as on the web without pthreads, there is no pool. A job then runs on the UI thread, at the start of the next `Ui.begin`, and is already `DONE` or `FAILED` when the frame asks about it. The frame stalls for as long as the job takes, but the app's code is the same either way: it still sees `PENDING` in the frame that starts the job and the result in a later one. Without threads, a job's outcome also no longer depends on timing: it is done exactly one frame after it starts.

5.6 Waking the window

An App-style loop sleeps while nothing changes (App style and Game style), and a job that finishes on a worker thread isn't an input event. So a worker calls `Platform.wake()` when its function returns, which ends the sleep on a platform whose `wakes()` is true (SDL3; Backends), and the next frame sees the result. The wake isn't counted as input, so on such a platform a running job costs no frames, and a finished one costs one.

A pending job can't wake the loop in two cases: it waits to run on the UI thread, because the `Ui` has no worker threads (`Ui.set_job_threads(0)`, or the web), or the platform doesn't wake (raylib: its `wake()` does nothing). In those cases `Ui.end` counts the `Ui` as animating while the job is pending, and an App-style loop keeps drawing frames, as it does while something moves, until it is done (App style: Kiln owns the loop). A Game-style loop draws every frame anyway. A pending job therefore costs frames at most while it runs, never a missed result.

5.7 A worked example

The gallery's Background page counts the primes below five million on a worker, with a progress readout and a Cancel button. The function reads its limit from `job.data`, reports progress and checks for cancelling every 65,536 numbers, and allocates its result in `job.allocator`:

```
fn void? count_primes(Job* job)
{
    long limit = *(long*)job.data;
    long count;
    for (long n = 2; n < limit; n++)
    {
        if (n % 65536 == 0)
        {
            if (job.cancelled()) return;
            job.set_progress((float)n / limit);
        }
        // ... count n if it is prime ...
    }
    long* result = alloc::new(job.allocator, long);
    *result = count;
    job.result = result;
}
```

The page asks about the job each frame while `g.primes_run` is not zero, with that run number as the key, and draws whatever the status calls for:

```
JobState s = ui.job("primes", &count_primes, &g.primes_below, key: g.primes_run);
switch (s.status)
{
    case PENDING:
        ui.@row(gap: 2, align: CENTER)
        {
            ui.spinner();
            ui.textf("Counting… %.0f%%", s.progress * 100);
            if (ui.button("Cancel", GHOST, SM))
            {
                ui.job_restart("primes");
                g.primes_run = 0;
            }
        };
    case DONE:
        ui.textf("%d primes below 5 million.", *(long*)s.result);
        if (ui.button("Count again")) g.primes_run++;
    case FAILED:
        g.primes_run = 0;
}
```

`g.primes_below` lives in the app's state, not on the stack, so it stays valid while the worker reads it. Cancel lets go of the job, which the function notices at its next check; the record and any result are freed when the worker returns. Count again changes the key, which lets go of the finished job, frees its result, and starts a new count. The result is read in place and never taken, so the app frees nothing.

The page's second card stands in for loading content. Its job, `slow_load`, makes no result; the card only needs to know when it is done. While the job is pending, `Ui.@loading` draws a skeleton in the card's place, at the card's size, and the button shows a spinner (Loading looks):

```
JobState s = ui.job("load", &slow_load, null, key: g.load_run);
ui.@loading("song card", s.status == PENDING, width: ui::GROW)
{
    // ... the card's content ...
};
if (ui.button("Load again", loading: s.status == PENDING)) g.load_run++;
```

The third card's job, `failing_lookup`, returns a fault, and the page shows `FAILED` as a message. The `hello` example's Save button runs a job the same way: `loading` shows a spinner on the button while it is pending, and the frame that sees it finish raises the "Saved" toast.

6 Widgets

A widget in Kiln is a function you call every frame. It reads last frame's input for its id, moves its springs, opens a few Clay elements and hands the renderer the numbers it needs to paint. Nothing is retained apart from a small state struct and last frame's rectangle per id. This chapter shows the pattern every widget follows, the calls it is built from, and the widget set built on them.

6.1 The five-step pattern

Every widget goes through the same five steps:

1. **Id:** hash the label into an `Id` with `Ui.make_id`.
2. **Input:** ask `Ui.interact` what the pointers and keys did to that id.
3. **State and motion:** fetch the widget's state struct with `Ui.@state` and move its springs toward the targets this frame's input sets.
4. **Paint data:** fill a paint struct from the frame arena with the spring values and resolved colors. It comes before the elements open, because they are opened with it.
5. **Build:** open Clay elements for the layout, with the id on the element whose rectangle should take input, passing the paint struct to a `Ui.custom` element or a box's `.paint`.

The switch shows all five. It shares its machinery with the checkbox and radio button, so it is split into three pieces. The state struct survives between frames and holds one spring per animated property:

```
struct ToggleState
{
    Spring on;       // 0 off, 1 on (checked, selected)
    Spring hover;
    Spring ring;
}
```

`Ui.toggle` itself makes the id (step 1), calls the shared helpers, and changes your variable when the row was clicked or activated:

```
fn bool Ui.toggle(&self, String label, bool* value, bool disabled = false)
{
    Id id = self.make_id(label);
    Interact r;
    TogglePaint* p = self.toggle_row(id, *value, disabled, &r);
    self.toggle_box(id, label_text(label), p, &paint_switch, 40, 24);
    if (r.clicked || r.activated)
    {
        *value = !*value;
        return true;
    }
    return false;
}
```

`toggle_row` does steps 2, 3 and 4. It calls `interact`, sets the pointer cursor on hover, moves the `on`, `hover` and `ring` springs with their motion tokens (`STATE`, `HOVER`, `FOCUS`, see Motion), and fills a `TogglePaint` from the frame arena. Theme tokens become `Color`s here, through `Ui.c`, already faded to the disabled opacity when needed:

```
fn TogglePaint* Ui.toggle_row(&self, Id id, bool on, bool disabled, Interact* out)
{
    Interact r = self.interact(id, focusable: !disabled, disabled: disabled);
    if (r.hovered && !r.shown) self.cursor = POINTER;
    *out = r;
    ToggleState* st = self.@state(ToggleState, id);
    st.on.go(on ? 1 : 0, STATE);
    st.hover.go(r.hovered ? 1 : 0, HOVER);
    st.ring.go(r.ring ? 1 : 0, FOCUS);
    float opacity = disabled ? DISABLED_OPACITY : 1;
    TogglePaint* p = self.frame_new(TogglePaint);
    *p = {
        .ui = self,
        .on = st.on.value,
        .hover = st.hover.value,
        .ring = st.ring.value,
        .off_fill = self.c(SURFACE_RAISED).alpha(opacity),
        .off_border = self.c(BORDER).mix(self.c(TEXT_SECONDARY), 0.35f).alpha(opacity),
        .hover_border = self.c(TEXT_SECONDARY).alpha(opacity),
        .on_fill = self.c(ACCENT).alpha(opacity),
        .mark = self.c(ON_ACCENT).alpha(opacity),
        .focus = self.c(FOCUS_RING),
        .opacity = opacity,
    };
    return p;
}
```

`toggle_box` is step 5: a row box carrying the id, so the whole row, control and label, takes the pointer and focus. The control isn't an element of its own. The row leaves room for it at its left with `.lead`, and the row's paint function draws it there through the `TogglePaint`'s `control`, centered on the row's height. A toggle is two layout elements, the row and its label, which matters on a settings page of thousands of them:

```
fn void Ui.toggle_box(&self, Id id, String label, TogglePaint* p, PaintFn control, float width, float
height)
{
    p.control = control;
    p.control_size = { width, height };
    float lead = label.len ? width + self.steps(TOGGLE_LABEL_GAP) : width;
    self.@box_ex({ .id = id, .height = fit(min: height), .dir = ROW, .align = CENTER, .bg =
NO_TOKEN, .lead = lead,
        .paint = &paint_toggle_box, .paint_data = p })
    {
        if (label.len) self.text(label, SM, opacity: p.opacity);
    };
}
```

The paint function runs later, while the core builds the draw list, with the control's final rectangle. It only draws from the paint data:

```
fn void paint_switch(DrawList* list, Rect r, void* data) @private
{
    TogglePaint* p = data;
    float radius = r.height / 2;
    list.rect(r, p.off_border.mix(p.on_fill, p.on), radius);
    float d = r.height - 6;
```

```
    float x = r.x + 3 + (r.width - 6 - d) * p.on;
    Color black = { 0, 0, 0, 255 };
    list.shadow({ x, r.y + 3, d, d }, black.alpha(0.25f * p.opacity), 2, d / 2, 1);
    list.circle({ x + d / 2, r.y + r.height / 2 }, d / 2 + p.hover * 0.75f, p.mark);
    list.focus_ring(r, p.focus, radius, p.ring);
}
```

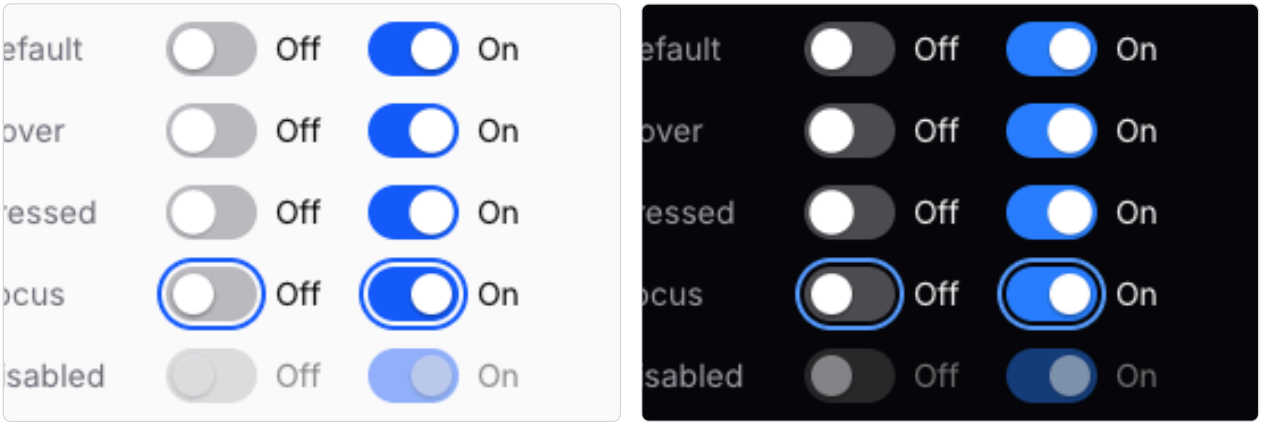


Figure 6: The gallery's switch page in the light and the dark theme. Each row is one interaction state (default, hover, pressed, focus, disabled) for an off and an on switch. The focus row shows the ring from the `ring` spring.

6.2 The core calls

6.2.1 Ids: `make_id` and `##`

`make_id(label)` hashes the whole label with FNV-1a, seeded with the top of the id stack. Everything after `##` is hashed but not shown: `"Save##toolbar"` displays "Save" and has a different id from a plain `"Save"`. `ui::label_text` returns the shown part.

Loops and repeated sections scope their ids with the `Ui.@id` macro, whose key is a `String` or an integer such as a loop index. `Ui.push_id` and `Ui.pop_id` (with `Ui.push_id_index` for integers) do the same without a body; `Ui.end` checks that every push was popped.

Two widgets with the same id in one frame would share state and input. Safe builds catch it: `make_id` counts it in `ui.duplicate_ids` and prints a warning on stderr naming the label. `Ui.make_id_quiet` computes an id without counting it as used, for code that needs another widget's id.

6.2.2 Input: `interact`

`interact(id, focusable:, disabled:, autofocus:)` reports what happened to `id` this frame, as an `Interact` bitstruct:

```
bitstruct Interact : ushort
{
    bool hovered;      // a mouse or pen is over it, and no other widget holds that pointer
    bool pressed;      // a pointer was pressed on it (the first frame only)
    bool held;         // a pointer pressed on it is still down
    bool released;     // the pointer holding it was released this frame
    bool clicked;      // released over it without dragging
    bool dragging;     // held, and moved past the drag threshold
    bool focused;      // has keyboard focus
    bool activated;    // Enter, Space or gamepad A while focused
    bool ring;         // focused, and focus is being moved by keyboard or gamepad: draw the focus ring
    bool shown;        // hover, hold or focus here is only shown, by ui.@show_state: don't act on it
    bool context;      // a context menu was asked for on it: a secondary press, a long touch, the Menu key
```

```
    bool long_press;   // the context menu came from a long touch
    bool touch;        // a finger is holding it
}
```

It tests the pointers against the widget's rectangle from the last frame, so a press lands on what the user saw. Only widgets in the layer the pointer is over take part: a floating layer above the page shields the widgets under it (Pointers and layers). Within that layer, widgets called later are drawn on top, so when rectangles overlap, the last widget containing a pointer becomes that pointer's candidate. At `end()` the candidate becomes the hovered widget and, on a fresh press, the widget that captures the pointer. Widgets see both from the next frame. A captured pointer stays with its widget until release: dragging across other widgets doesn't hover or press them, and the captured widget itself reads as hovered only while the pointer is over it. `clicked` means released inside the widget without having dragged.

Each pointer is tracked on its own (The input of a frame), so two fingers can hold two widgets. A pointer that vanishes without a release, such as a cancelled touch, lets go without clicking. Touch differs from mouse and pen in three ways:

- Nothing hovers.
- Hit areas grow to at least 44 × 44 pt around the widget's center.
- A press becomes a drag after moving 10 pt (6 pt for mouse and pen), and a drag doesn't count as a click.

With `focusable`, the widget is a focus stop, and pressing it focuses it. While it has focus, `focused` is set, `ring` says whether to draw the focus ring (focus moved by keyboard or gamepad, not by a press), and Enter, Space or gamepad A set `activated`. A `disabled` widget gets an empty `Interact`, loses focus if it had it, and is not a focus stop. Focus and key routing are covered in Input and focus.

`shown` marks states that `Ui.@show_state` fakes for galleries and screenshots (Testing): the widget draws them but must not act on them, which is why widgets set the cursor only when `!r.shown`.

6.2.3 State: `@state`

`self.@state(Type, id)` returns a pointer to a `Type` that belongs to `id`, zeroed the first time the id asks. Asking for a different type under the same id is a contract violation; where contracts are off, the old state is freed and the new type starts zeroed, so two kinds of widget swapping one id never read each other's memory. The state is freed after it goes unused for a while; State and memory covers the lifetimes.

6.2.4 Painting: `custom` and box paint

`custom(width, height, paint, data, id)` opens a Clay element holding `{paint, data}`. While building the draw list, the core calls `paint(DrawList*, Rect, void* data)` with the element's final rectangle. Given an id, the element's rectangle is kept for `interact`; so is any box's with an `.id`. The slider puts its id on the `custom` element; the toggles put it on the row box, so the label takes clicks too, and paint their control from the row box rather than a `custom` element of its own. A box opened with `Ui.@box_ex` can also take a paint function in its `BoxOpts` fields `.paint` and `.paint_data`, which draws its background under its children; that is how buttons and links paint.

A custom widget needs nothing special to work inside a floating layer (Layers). Kiln records the layer each id is built in, gives the widget the pointer only when the pointer is over that layer, gives it keys only when the layer is on top, and fades its paint with the layer.

Paint data is allocated from the `Ui`'s frame arena with `Ui.frame_new`, never from C3's global `tmem` (see State and memory).

6.3 Rules

1. Each widget has one state struct and one paint struct, named after it (`ButtonState`, `ButtonPaint`). A family of widgets drawn the same way shares them: the checkbox, switch and radio button all use `ToggleState` and `TogglePaint`.
2. Colors come from theme tokens, resolved with `self.c(token)` into the paint data. Text and icons take a `Token` (`color: TEXT_SECONDARY`); only paint functions see `Color`s. The one literal color is black, for shadows and for darkening `DANGER` on hover and press.
3. Options are named arguments with defaults; in an options struct the zero value is the default, which is why `ControlSize` lists `MD` first.
4. Every widget supports the keyboard, and its motion goes through springs, which jump straight to their targets when reduced motion is on (Motion).
5. Disabled means 45 % opacity (`ui::DISABLED_OPACITY`), carried into children through `opacity:`, and no focus stop. A disabled widget takes no input but its tooltip: it still holds the pointer, so a tooltip on it can say why it is off, and a press on it reaches nothing beneath, but it shows no hover or press and never clicks.
6. Paint functions only draw. They never change state.

6.4 Return values and binding

Most widgets return `bool`: buttons and links return true when clicked or activated, toggles when they changed your variable. A result struct is used only where one bool isn't enough, such as the slider reporting both "changed" and "released".

Widgets that edit a value take a pointer to your variable: `ui.toggle("Wifi", &on)`.

6.5 Options

Each widget's options are defined once, as a struct:

```
struct ButtonOpts
{
    Variant variant;
    ControlSize size;
    bool disabled;
    Icon icon;        // before the label
    Icon icon_end;    // after the label
    bool full;        // as wide as the parent allows
    bool loading;     // a spinner over it, keeping its size; it doesn't click
}
```

A widget has two entry points: one with named arguments for everyday use, and an `_ex` form that takes the struct, for presets, forwarding and options held in data. The named-argument version only fills in the struct and calls `_ex`, so behavior and defaults live in one place:

```
// The body of Ui.button(label, variant, size, disabled, icon, icon_end, full, loading):
return self.button_ex(label, { variant, size, disabled, icon, icon_end, full,
loading });
```

```
fn bool Ui.button_ex(&self, String label, ButtonOpts opts)
{
    return self.button_box(self.make_id(label), label_text(label), opts, false);
}
```

Presets compose by name with initializer splatting:

```
if (ui.button("Save", PRIMARY)) save();                    // named arguments
ButtonOpts toolbar = { .variant = GHOST, .size = SM };
if (ui.button_ex("Undo", { ...toolbar, .icon = ARROW_LEFT })) undo();
ButtonOpts danger_toolbar = { ...toolbar, .variant = DANGER };
```

Splatting a struct straight into a call (`ui.button("Save", ...toolbar)`) is not used. It matches fields by position, so reordering parameters could silently swap two fields of the same type, such as two `bool`s.

6.6 The widget set

Widget	Call	Notes
Heading	`Ui.heading(text, level = 1)`	Level 1 is `XXXL`, 2 `XXL`, 3 `XL`, 4 and above `LG`; SemiBold; wraps
Secondary text	`Ui.secondary(text)`	`SM` in `TEXT_SECONDARY`; wraps
Link	`Ui.link(text, disabled) -> bool`	`ACCENT` text with an underline that fades in on hover or focus; focusable. Disabled, it draws in `TEXT_DISABLED`
Button	`Ui.button(label, variant, size, disabled, icon, icon_end, full, loading) -> bool`	Variants `DEFAULT`, `PRIMARY`, `OUTLINE`, `GHOST`, `DANGER`; heights `SM` 28 pt, `MD` 36 pt (default), `LG` 44 pt; `full` grows to the parent's width; `loading` shows a spinner (Loading looks)
Icon button	`Ui.icon_button(icon, label, variant = GHOST, size = MD, disabled) -> bool`	Square, as wide as the button is tall; `label` is its id
Icon	`Ui.icon(icon, size = 20, color = TEXT, opacity = 1)`	A Lucide glyph in a square `size` points wide
Checkbox	`Ui.checkbox(label, &value, disabled, mixed) -> bool`	An 18 pt box; `mixed` shows a minus (some, not all, of what it stands for is on), and a click turns it on
Switch	`ui.toggle(label, &value, disabled) -> bool`	40 × 24 pt; `switch` is a C3 keyword, so the switch is `toggle`
Radio group	`Ui.radio_group(id, &selected, options, dir = COLUMN, disabled) -> bool`	`selected` is an `int` index into the `String[]` options. One focus group: Tab enters at the selected option, the arrows move and choose (WAI-ARIA)
Radio	`Ui.radio(label, selected, disabled) -> bool`	One option, for custom layouts inside `Ui.@focus_group`
Slider	`Ui.slider(label, &value, min = 0, max = 1, step = 0, format = "", disabled) -> SliderResult`	`step` 0 is continuous; a non-empty `format` shows the value after the track; `"##name"` hides the label

All `disabled` parameters default to `false`, as do `mixed`, `full` and `loading`; `icon` and `icon_end` default to `NONE`.

6.6.1 Slider

The slider is the widget that needs a result struct, `SliderResult`, because "it moved" and "the user let go" are different moments: update a preview on `changed`, save on `released`. `released` comes when the pointer lets go (after a drag or a tap) and after every keyboard or gamepad step.

While focused it takes the arrows (one `step`, or 1 % of the range when continuous), Page Up and Down (a tenth of the range, at least one step), Home and End through `Ui.key_pressed`, and the d-pad's left and right through `Ui.pad_pressed`. Both consume the key, so focus navigation never sees it; Tab and the d-pad's up and down still leave the slider. While held, the value follows the pointer, found with `Ui.pointer_of`, which gives the position of the pointer holding a widget, for any widget that drags.

A widget that drags calls `Ui.claim_pointer(id)` each frame it is held, as the slider does. Otherwise a touch or pen drag along its track that crosses the drag threshold would be taken by the scroll area around it, which lets go of the widget and moves the page instead (Scrolling).



Figure 7: The slider page: the thumb grows on hover and more while pressed, and the focus row rings the thumb rather than the track.

6.6.2 Toggles

Toggles (checkbox, switch, radio) return true when they changed your variable. The whole row, control and label, takes the pointer and focus; the focus ring circles the control. Radio options choose on arrival when the arrows or d-pad move focus (Focus).

A custom choice widget built inside a `Ui.@focus_group`, such as a segmented control or a row of `Ui.radio` options in your own layout, calls `Ui.set_group_entry(id)` with its chosen option's id each frame, so Tab enters the group at the chosen option rather than at the member focused last.

6.6.3 Variants

A variant says what a button means, not what color it is. `PRIMARY` fills with `ACCENT`, hovers to `ACCENT_HOVER` and presses to `ACCENT_PRESSED`, with `ON_ACCENT` text. `DANGER` fills with `DANGER` and darkens on hover and press. `DEFAULT` fills with `BACKGROUND`. `OUTLINE` (with a `BORDER` hairline) and `GHOST` are transparent until hovered. Hover, press and the focus ring are springs.

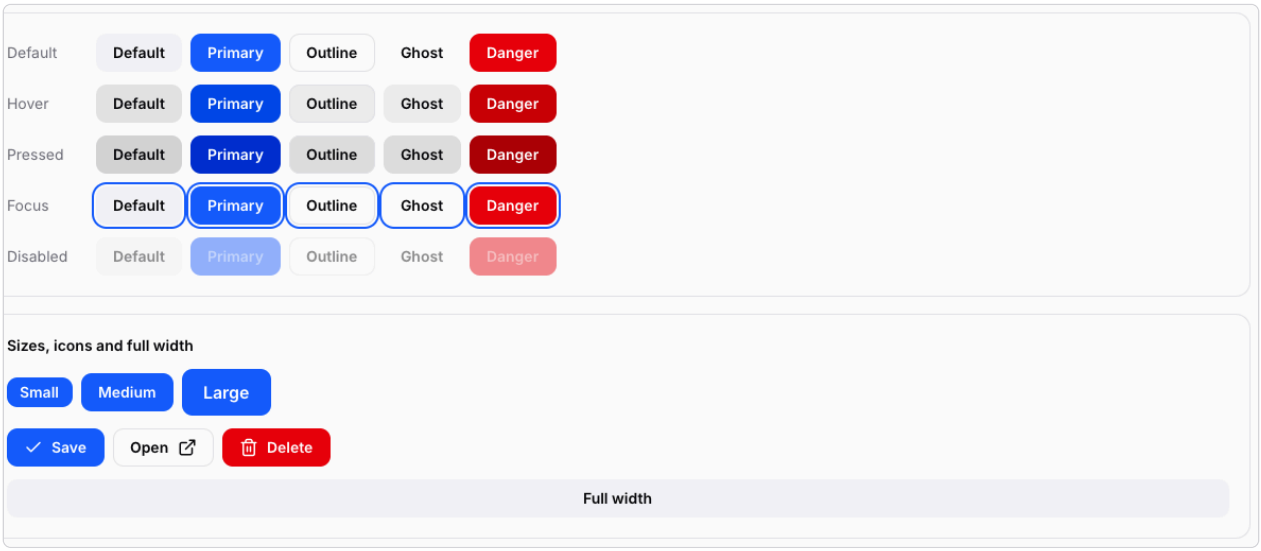


Figure 8: The buttons page. In the top panel each column is a variant and each row an interaction state: default, hover, pressed, focus (the ring) and disabled. Below it are the three sizes, buttons with leading and trailing icons, and a `full` button.

6.6.4 Icons

Icons are glyphs from the embedded Lucide icon font (ISC, with MIT for the icons from Feather), so they go through the glyph atlas like text (Text), scale crisply and take a color token. The `Icon` enum is generated from a curated list of Lucide names (`scripts/make_icons.sh`). Widgets that paint icons themselves, such as the checkbox's check mark, use `Ui.draw_icon(list, icon, rect, color)` from their paint function.

6.6.5 Cursor

`Ui.begin` resets `ui.cursor` to `ARROW`; hovered buttons (unless loading), links and toggles, and hovered or held sliders, set it to `POINTER`. `end()` tells the platform only when it changes.

6.7 Loading looks

Content that a job is still making (Background work) has two looks: a skeleton where the content will be, and a spinner that says something is under way. Both take their timing from looping motion (Motion), and both hold still under reduced motion.

6.7.1 Skeleton: `@loading`

`Ui.@loading(id, pending, width)` wraps the content that is pending:

```
ui.@loading("profile", pending: !profile.ready, width: ui::GROW)
{
    ui.heading(profile.name, 3);
    ui.secondary(profile.bio);
};
```

The body is built every frame, pending or not, so the box takes the size and layout the content gives it, and the page doesn't jump when the content arrives. `id` names the box, and `width` sizes it like any box: `ui::GROW` for content that wraps, which would otherwise have no width to wrap at.

While `pending`, a skeleton covers the box. It is a Clay floating element attached to the box, at the z order of the layer being built plus one, sized to the box's rectangle from last frame. A box built for the first time has no such rectangle, so the first frame shows the content uncovered and asks for another frame, which covers it. The skeleton is an opaque surface: `SURFACE` mixed slightly toward

`TEXT`, with a control's corner radius. It reads a little darker in the light theme and a little lighter in the dark one. The mix breathes in and out with `Loop.PULSE` (`Loop`).

A pending body is inert as well as hidden. The skeleton calls `interact` with its own id after the body, so it is the last widget under the pointer and takes it from the widgets beneath.

Keys are handled through a counter on the `Ui`, `inert`, which `@loading` raises around a pending body. While it is above zero, `interact` treats every widget as not focusable, so nothing under the skeleton is a focus stop and Tab passes over it. A widget that had focus when its content went pending stops being a stop, and focus moves on as when a widget goes away (Focus).

6.7.2 Spinner

`Ui.spinner(size = MD, color = TEXT_SECONDARY)` draws eight dots round a circle as big as a control's icon at that `ControlSize`. One dot is the brightest and the dots behind it fade; the bright one goes round once per `Loop.SPIN`. A paint function draws one anywhere with `ui::draw_spinner(list, center, size, phase, color)`, taking the phase from `Ui.loop_phase`, as buttons do.

6.7.3 Loading buttons

A button whose action is running takes `loading` (`ButtonOpts.loading`, or `ui.button(..., loading: true)`). Its content keeps its place but is drawn at zero opacity, so the button stays as wide as its label makes it, and its paint function draws a spinner in the middle, at the icon size, in the label's color. A loading button doesn't return true, either on a click or on activation, and doesn't set the pointer cursor; it keeps its focus, so focus isn't lost while the action runs.

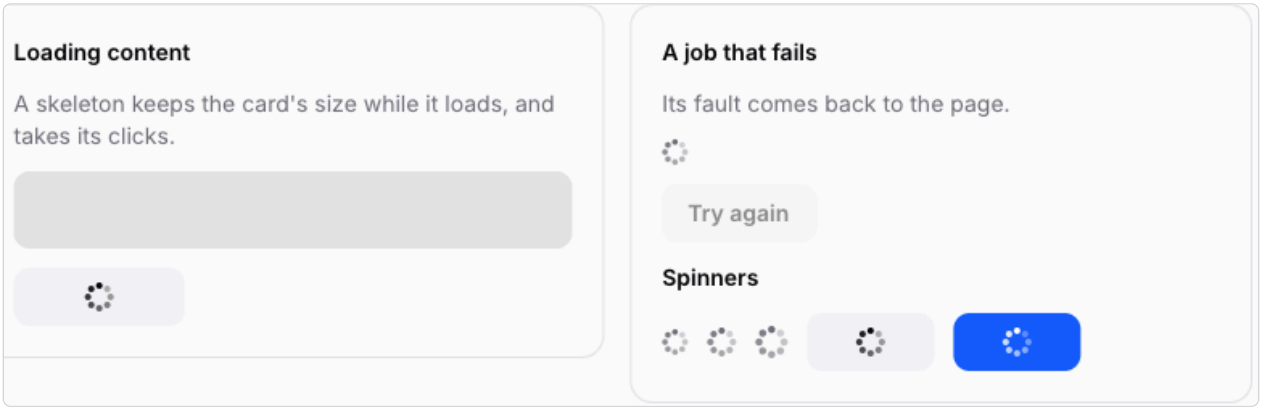


Figure 9: The gallery's background page. Left: a skeleton over a pending card, and a loading button below it. Right: spinners at the three sizes, then a loading `DEFAULT` and a loading `PRIMARY` button.

7 Input and focus

Kiln is built for the keyboard and the gamepad as much as for the pointer. Every focusable widget can be reached with Tab, the arrow keys or the d-pad, dialogs keep focus inside themselves, and shortcuts are asked for in the code that handles them. This chapter explains the input the core reads each frame, how focus is stored and moved, and how a key press is routed between widgets, shortcuts and navigation. How a widget asks about its own input, through `Ui.interact` and the `Interact` bits, belongs to Widgets.

7.1 The input of a frame

The core never polls a device. Each frame, a platform backend (or a test) fills one `Input` value and hands it to `Ui.begin` Backends describes who does that.

```
struct Input
{
    double time;          // seconds since some fixed start; 0 lets the Ui add up dt
    float dt;             // seconds since the last frame
    float[<2>] size;      // the window's size in logical points; 0 keeps the last size
    float scale;          // pixels per point; 0 keeps the last scale
    bool unfocused;       // the window doesn't have the keyboard focus (another app
does)
    bool paused;          // the platform has no visible surface; the app should not
build or draw a frame
    bool events;          // anything happened at all (an idle app may sleep when not)
    Pointer[MAX_POINTERS] pointers;
    usz pointer_count;
    float[<2>] wheel;     // wheel movement this frame, x and y
    KeySet keys_down;
    KeySet keys_pressed;  // went down this frame (including key repeat)
    Mods mods;
    Gamepad pad_down;     // the first gamepad
    Gamepad pad_pressed;  // went down this frame
    char[64] typed;       // UTF-8 the keyboard typed this frame; see `Input.text`
    usz typed_len;
    char[256] composing;  // UTF-8 an input method is composing at the caret, not typed
yet; see `Input.composition`
    usz composing_len;
    usz composing_cursor;  // where the caret is in it, in bytes
    usz[2] composing_clause;   // the part the input method is converting, as a byte
range in it; empty for none
    // Dropped on the window this frame: files from a file manager, text such as a link.
The
    // platform owns them until its next poll; an app that keeps them copies them.
    String[] dropped;     // the files' paths
    String dropped_text;
    float[<2>] drop_pos;  // where, in points
    bool dragging;        // something is being dragged over the window
    float[<2>] drag_pos;  // where, while dragging, where the platform reports it
    BackGesture back;     // the system's predictive back gesture, if any this frame
}
```

- **Pointers** are listed with their state this frame, up to `ui::MAX_POINTERS` (ten). Each has a stable `id` (a finger, the mouse), a `PointerKind` (`MOUSE`, `TOUCH` or `PEN`), a position, whether it is down (the primary button, or in contact), whether its `secondary` button is held (the right mouse button, or a

pen's barrel button), and whether its `middle` button is held, which on Linux and the BSDs pastes the primary selection (The primary selection). A lifted touch appears once more with `down` false, then disappears.

- **The wheel** is the wheel's movement this frame, in notches along x and y. Scroll areas take it (Scrolling).
- **Keys** are a `Key` enum: Tab, Enter, Space, Escape, Backspace, Delete, Insert, the four arrows, Home, End, Page Up and Page Down, the letters `A` to `Z`, the digits `D0` to `D9`, `F1` to `F12`, a few punctuation keys, and `MENU`, the Menu (application) key that opens a context menu (Context menus). `NONE` is no key: no platform reports it, and it stands for "no shortcut", as on a menu row without one (Menus, popovers and tooltips). A `KeySet` holds one bit per key. `keys_pressed` has the keys that went down this frame, including key repeat, so holding an arrow keeps moving focus; `keys_down` has the keys held.
- **Modifiers** are a `Mods` bitstruct: `shift`, `ctrl`, `alt` and `super`.
- **Typed text** is the text the keyboard produced this frame, as UTF-8 in `typed`, up to 64 bytes a frame (more is dropped); `Input.text` returns it. It is separate from the keys: a key press says which key went down, typed text says which characters came out once the keyboard layout and Shift are applied, so "é" arrives as one character whatever keys made it. A text field reads it (Text editing), and a select reads it to jump to an option by its first letters (Type-ahead); other widgets use the keys.
- **The composition** is what an input method is composing at the caret and hasn't committed yet, such as the "nihon" that becomes "　", as UTF-8 in `composing`, up to 256 bytes; `Input.composition` returns it, empty when nothing is being composed. `composing_cursor` is where the input method's caret is in it and `composing_clause` the part it is converting, both as byte offsets into it. The focused field shows it without typing it; once the input method commits, the result arrives as typed text (Input method composition).
- **The window's focus** is `unfocused`, true while the window doesn't have the keyboard focus because another app does. The platform reports it, and Kiln reads it to send a toast out as a system notification while the user is looking elsewhere (Notifications).
- **The gamepad** is the first one, as a `Gamepad` bitstruct with its buttons named by position, so `a` is the bottom face button whatever the controller prints on it:

```
bitstruct Gamepad : ushort
{
    bool a;
    bool b;
    bool x;
    bool y;
    bool lb;
    bool rb;
    bool start;
    bool back;
    bool up;
    bool down;
    bool left;
    bool right;
}
```

Tests build an `Input` by hand with `Input.pointer` (whose `secondary:` and `middle:` arguments hold the secondary and middle buttons), `Input.press`, `Input.press_pad` and `Input.add_text`, which appends text as far as it fits and never cuts a character in half. Backends that receive characters one at a time call `Input.add_codepoint`, which encodes one as UTF-8. Backends and tests set the composition with `Input.compose(text, cursor, clause)`, which cuts text past what fits at a

character's start. Reporting the same composition again isn't an event: the backend sets `events` only when it changes.

The raylib platform fills the mouse pointer's `secondary` from the right mouse button and its `middle` from the middle one, and reports `KB_MENU` as `MENU`. It reads the first gamepad: the four face buttons by position, the shoulder buttons (`lb`, `rb`), start and back, and the d-pad. The left stick counts as the d-pad: past half its range it holds the matching direction, and it reports a press when it crosses that threshold, once per push. On Android it reports the system's back button or gesture as an Escape press, so back closes the top layer as Escape does on a desktop. It sets `unfocused` from raylib's `IsWindowFocused`, and fills the typed text from `GetCharPressed`, one character at a time until its queue is empty.

7.2 Pointers and layers

A pointer goes to one layer first, and to a widget within it second. Floating layers (`Ui.@layer`, Layers) sit above the page, and each frame Kiln finds the layer each pointer is over from last frame's layers: the topmost modal layer wherever the pointer is, else the topmost layer containing it, else the page. Only widgets in that layer test the pointer against their rectangles, so a button on the page under an open menu never sees it. Within the layer, build order decides as before: the last widget containing the pointer takes it (The core calls).

`Ui.last_id` returns the id of the widget that last called `Ui.interact` this frame, such as the button just built, which is how a menu is anchored to the button that opens it without the caller knowing the button's id.

7.2.1 A long touch press

Each pointer remembers when it was last pressed, so a widget can tell a tap from a touch held in place. A tooltip uses it (Menus, popovers and tooltips): a touch has no hover, so a finger held on a widget for the tooltip delay shows the widget's tooltip, and the press then counts as a drag, so lifting the finger doesn't click. On a widget with a context menu, the menu wins: the touch asks for it after `ui::LONG_PRESS` instead, and the tooltip yields (Context menus).

7.2.2 Double and triple clicks

Each pointer also counts its clicks in a row. A press counts as the next click of a run when it comes within `ui.double_click_time` seconds of the previous press and lands within 4 points of it (12 for touch); otherwise it starts a new run of one. The count stops at three, and a fourth quick press starts again at one. `Ui.click_count(id)` returns the count of the pointer holding a widget: 1, 2 for a double-click, 3 for a triple; 0 when no pointer holds it. A text field selects a word on 2 and everything on 3 (Text editing).

`ui.double_click_time` comes from the OS, or 0.4 s where the OS has no such setting, and an app can pin it with `Ui.set_double_click_time` (Following the system).

7.2.3 Where a press landed

A widget learns of a press the frame after the press happens: the pointer is captured at `end()`, once the frame knows which widget is on top (The core calls). By then the pointer may have moved, so `Ui.pointer_of` gives where it is now, and `Ui.press_point_of(id)` gives where the pointer holding the widget went down: the start of a drag, or where a click landed. A text field places its caret at the press point, so a quick press-and-drag selects from where the press began.

7.2.4 A middle click

The middle button is not a primary button either: its press never captures a widget, never clicks and never focuses. `Ui.settle_pointers` records which widget was on top under it and where, and

widgets hear of it in the next frame, as they hear a context request (Context menus): `Ui.middle_clicked(id)` is true in the frame after a middle click landed on `id`, and `Ui.middle_point` says where. A text field or area pastes the primary selection there (The primary selection).

7.2.5 Holding a widget

`Ui.is_held(id)` says whether a pointer pressed on the widget is still down: true from the frame the press is reported until the pointer is released, and false in the release frame, where `Ui.pointer_of` still answers with the pointer's last position. A control that acts for as long as it is held uses it: the steppers of a number field step once on the press and then repeat while the pointer stays down (Inputs). Since nothing arrives while a pointer rests, such a control asks for its next repeat with `Ui.wake_after`.

7.2.6 A press on a selectable text

A selectable text (`Ui.text` with `selectable: true`, Text) is not a focus stop, yet once the user selects in it, it needs the keyboard to copy. The context keeps that text in `Ui.selected_text`, an `Id` separate from the focused widget. A press on a selectable text makes it the selected text and clears `focused`, so the keyboard leaves any text field that had it: typing and copying never go to two places at once. A drag or Shift+click extends the selection, a double-click selects a word and a triple-click everything.

The selection lasts until something else takes the keyboard or the pointer. In `Ui.settle_pointers`, any new press whose target isn't the selected text clears it, whether it lands on a button, on another text or on empty space. A secondary press does the same, so right-clicking elsewhere lets go of the selection, while one on the selected text keeps it for the text's own menu; presses inside that menu, while it is open, don't clear it either. `Ui.selected_focus` records what had focus when the text took the keyboard (nothing, since the press cleared it), and as soon as focus changes, by Tab, an arrow or a press on a focusable widget, the text drops its selection. Escape lets go too (Selecting in a text).

7.2.7 Waking for a deadline

Some input means something only after a wait: a pointer resting on a widget until its tooltip shows, or on a menu row until its submenu opens. Nothing arrives when the wait ends, so an App-style loop asleep for input would never see it. `Ui.wake_after(seconds)` asks for a frame that many seconds later. The earliest deadline still to come wins; one asked for in the frame an earlier deadline arrives replaces it. On a platform that can sleep until a deadline (SDL3), the loop sleeps until then; on one that can't (raylib), the `Ui` counts as animating until the deadline passes, so the loop keeps running frames instead of sleeping (App style: Kiln owns the loop). A touch held for a long press asks for its frame the same way.

7.3 Context menus

A context menu belongs to a widget and can be asked for in four ways, two by pointer, one by touch and one by keyboard. The core turns each into the same **context request**: a target widget and a point. `Ui.ask_context` records them in `settle_pointers` (or at `end()`, for the keys), and widgets hear the request in the next frame, while they are built: `Ui.context_requested(id)` is true for the target, `Ui.context_point` says where the menu opens, and `Ui.interact` sets the `context` bit of `Interact` on the target. A later request replaces an earlier one. `Ui.@context_menu` builds on this (Menus, popovers and tooltips), and so do the built-in menus of a text field (Text editing) and of a selectable text (Text).

A widget with no menu of its own falls back to the row it belongs to, when that row has one: a secondary press or a long touch on a button or link inside a list or table row opens the row's menu. The widget's own menu wins when it has one (One menu for every row).

7.3.1 A secondary press

The secondary button is not a second primary button. A secondary press never captures a widget, never clicks and never starts a drag; it only asks for a context menu, targeting the widget under the pointer at the pointer's position. Before that, it closes open menus and popovers as a primary press outside them does, so right-clicking elsewhere replaces an open context menu with a new one: unlike a primary press, the closing doesn't swallow the press, and the widget under it still gets its request. It also clears the selected text unless it lands on it (Pointers and layers).

With Mac keys (`ui.text_keys`), Ctrl+click is a secondary press too, as on macOS: the press isn't a press at all, so it neither clicks nor focuses, and it asks for the context menu of what is under the mouse.

7.3.2 A long touch

A touch has no secondary button, so it holds still instead. A touch resting on a widget for `ui::LONG_PRESS` (half a second) asks for that widget's context menu at the touch point, once per press, provided the widget has one. A widget registers its menu each frame with `Ui.has_context(id)`, and the long touch checks against last frame's registrations; `Ui.@context_menu` and the built-in menus do this for you. A widget inside a row falls back to the row's menu when it has none itself. The touch then counts as a drag, so lifting the finger doesn't click, and the tooltip, which otherwise shows on a long touch, stays hidden. A touch that moves past the drag threshold first is a drag, not a long touch.

A right-click or a long touch on a list row outside the selection selects that row before its context menu opens, as Finder, Explorer and GTK do, so the menu acts on the row the user pointed at (Lists).

7.3.3 The Menu key and Shift+F10

On the keyboard, the Menu key or Shift+F10 asks for a context menu. `Ui.context_keys` reads them at `end()`, before navigation, and only when no widget used the key during the build. The target is the selected text if one holds the keyboard, else the focused widget, and the menu opens at the bottom-left corner of its rectangle. With neither, the key goes on unused. `Ui.shortcut_label` spells the key "Menu".

7.4 Focus

7.4.1 Focus is an id

The context stores the focused widget as an `Id`, so focus survives from frame to frame while the widgets themselves are rebuilt. Code moves it with `Ui.focus` (`ui::NO_ID` clears it). A widget takes focus three ways:

- `interact(id, focusable: true)` focuses the widget when a pointer presses it.
- `interact(id, focusable: true, autofocus: true)` focuses it the first frame it appears (it wasn't a focus stop last frame), if it is in the top layer.
- Navigation moves focus to it: Tab, the arrows or the d-pad.

If the focused widget stops being called, focus moves to the stop that followed it in last frame's order within the same focus scope, else to the one before it, else nowhere. A widget that turns disabled while focused drops focus. A selectable text that holds a selection also takes keys without

being focused; it is tracked apart from focus, in `Ui.selected_text` (see "A press on a selectable text" above).

7.4.2 Tab order is build order

Every `interact(id, focusable: true)` registers a **focus stop**, so the stops of a frame are in call order, and that is the Tab order. A stop remembers the innermost focus scope and focus group around it. There is no numeric tab index; to change the order, change the order of the calls. Disabled widgets register no stop, and neither do widgets that are on screen but not live: those under a pending skeleton (Loading looks) and those in a disclosure that is closing (Navigation). `@loading` and `@disclosure` raise a counter on the `Ui`, `inert`, around such a body, and while it is above zero `interact` treats every widget as not focusable.

Tab and Shift+Tab, when no widget used them, move to the next and previous stop at `Ui.end`, wrapping around. With nothing focused, Tab goes to the first stop and Shift+Tab to the last.

7.4.3 Focus groups: roving focus

A focus group makes a set of related widgets (radio buttons, a segmented control, tabs, a menu) one focus stop:

```
ui.@focus_group("buttons", VERTICAL)
{
    if (ui.button("Resume", PRIMARY, LG, icon: PLAY, full: true)) open = false;
    if (ui.button("Settings", size: LG, icon: SETTINGS, full: true)) g.settings = true;
    if (ui.button("Main menu", GHOST, LG, icon: HOUSE, full: true)) g.screen = MENU;
};
```

Tab enters the group at the member that had focus last, or at its first member. The arrow keys along the group's axis (`HORIZONTAL`, `VERTICAL` or `BOTH`) move to the next or previous member in build order: the WAI-ARIA roving-focus pattern that web widgets use. Right and Down move forward, Left and Up move back. By default the arrows stop at the ends; with `wrap: true` they go round from the last member to the first and back, as in a menu, whose rows are a wrapping vertical group (Menus, popovers and tooltips). An arrow across the axis leaves the group the directional way described below. `Ui.radio_group` is a focus group whose remembered member is the chosen option, and a radio button chooses itself when the arrows move focus onto it.

A list or a table uses the other model, the one native list boxes use: its rows are one focus stop holding a **cursor row**, not a focus group of rows. A long list builds only the rows in view, and a row that isn't built can't hold focus, so the list keeps the cursor and answers the keys itself (Keys in a list, Lists). A table's header buttons are a horizontal focus group of their own, before its rows.

7.4.4 Focus scopes: layers

A focus scope marks a layer, such as a dialog or a menu:

```
if (confirm_open)
{
    ui.@focus_scope("confirm", &confirm_open)
    {
        ...
    };
}
```

`Ui.@focus_scope(id, open, trap)` takes an optional `bool*` and a `trap` flag that is on by default. The last trapping scope declared in a frame is the **top layer**. A floating layer (`Ui.@layer`, Layers) opens one around its body unless it takes no focus, so an open menu or dialog is the top layer:

- Tab and the arrows stay among its stops.
- Only its widgets are activated or read keys, and only shortcuts declared inside it fire.
- When it appears, focus moves into it: to its first stop, unless focus is already inside.
- Escape or gamepad B clears its `open` flag, unless its layer is kept open (`keep_open`, How it was dismissed); either way `Ui.dismissal` reports `ESCAPE` for it the next frame. Your code then stops declaring the scope.
- When it is no longer declared, focus returns to its **opener**, the widget that had focus when the scope appeared, unless focus is already on a widget that still exists.

With no top layer, every stop takes part. A scope with `trap: false` is not a layer for navigation, but it still bounds `local` shortcuts, and focus still returns to its opener when it goes.

The top layer is known only once the frame has declared all its scopes, so widgets and shortcuts check against the top layer as the last frame left it, while navigation at `end()` uses this frame's.

7.4.5 F6: to the toasts and back

A toast doesn't take focus when it appears, so typing on the page goes on undisturbed (Dialogs, sheets and toasts). F6 reaches the toasts from the keyboard: while toasts show, it moves focus to the newest toast's first button (its action, or else its dismiss button) and turns the focus ring on; pressed again from inside a toast, it moves focus back to the widget that had it before. While focus is in a toast, the toast's time pauses, so it doesn't leave under the user's hand.

7.4.6 Directional navigation

An arrow outside a group, or across a group's axis, and the d-pad move focus to the nearest stop in that direction within the layer. The search uses last frame's rectangles, the ones the user is looking at, and measures from center to center. A stop counts only if it lies more than half a point ahead. Among those, the lowest score wins:

$$\text{score} = \text{along} + 2 \cdot \text{across}$$

where **along** is the distance in the direction pressed and **across** the offset perpendicular to it, so a stop straight ahead beats a closer one off to the side. The candidates are the stops Tab can reach in the layer, so the arrows enter a group at its remembered member. With nothing focused, an arrow goes to the layer's first stop.

7.4.7 The gamepad

The d-pad (and the raylib left stick) navigates like the arrows. **A** activates the focused widget, like Enter and Space. **B** goes back, like Escape. Other buttons are yours to read with `Ui.pad_pressed`, which returns true once per press and uses the button up: a game can pause with `ui.pad_pressed({ .start = true })`. Like keys, a button goes only to the top layer: called outside it, `pad_pressed` returns false and leaves the button for the layer on top.

`Ui.tabs` reads two buttons itself (Navigation). The shoulders switch tabs: RB to the next, LB to the previous, through `pad_pressed`, so a tab bar on the page doesn't switch tabs under an open dialog. Ctrl+Page Down and Ctrl+Page Up (⌘ with Mac keys) do the same from the keyboard through `Ui.shortcut`. A focused text field doesn't take those chords (Shortcuts), so they switch tabs even while the user types. A second tab bar on the same page passes `keys: false` to leave them to the first.

7.5 Shortcuts

A shortcut is asked for in the code that handles it, every frame:

```
// Ctrl+S, or ⌘S on macOS
if (ui.shortcut(S, primary: true)) save();
// Only while focus is inside the scope or group around this call
if (ui.shortcut(F, primary: true, local: true)) find();
```

`Ui.shortcut(key, primary, shift, alt, local)` returns true once, in the frame its key goes down with exactly its modifiers, and then uses the key up. `primary` is Ctrl, or ⌘ on macOS; the other of the two must not be held. Shift and Alt must match too, so `ui.shortcut(S, primary: true)` stays quiet on Ctrl+Shift+S. A shortcut never fires on a key something already used, and under a top layer only shortcuts declared inside it fire. With `local: true`, it fires only while focus is inside the innermost focus group around the call, or if there is none, the innermost scope. While a text field has focus, a shortcut also stays quiet on the chords the field uses for typing and editing (Typing in a text field). While a selectable text holds a selection, it stays quiet on the chords the selection uses (Selecting in a text). A shortcut on a key that types (a letter, a digit or Space) with no Ctrl, ⌘ or Alt shares the key with a focused list's type-ahead (Type-ahead): while the list's search is running, the key is the search's and the shortcut stays quiet; otherwise the shortcut fires and the list doesn't move. On a focused song table, A adds to the queue, while typing "hallelujah" finds the song.

`Ui.shortcut_label(key, primary, shift, alt)` spells the chord for a menu, in the frame arena: "Ctrl+Shift+S" on Windows and Linux, "⇧⌘S" on macOS. Letters, digits and function keys use their names ("S", "1", "F5"); the arrows are drawn as arrows.

In safe builds, a chord asked for twice in one frame and scope counts in `duplicate_shortcuts`, and the first time Kiln prints a warning to stderr: only the first of the two can fire.

7.6 Key routing

A key press goes to one taker. Each key is used up once something takes it, and nothing after that sees it:

1. **A widget** takes it with `Ui.key_pressed` during the build. That is how a focused widget hears Enter and Space (and `ui.pad_pressed` for gamepad A) to activate, and how a slider takes the arrows. `Ui.interact` does the activating for the widget unless it passes `keys: false`, as a text field does: there Enter belongs to the field and Space types a space. Widgets outside the top layer never get keys or buttons.
2. **A shortcut** takes it when it fires.
3. **A context request** takes the Menu key or Shift+F10 at `end()`, if it is left and something has the keyboard (Context menus).
4. **Navigation** runs at `end()` with what is left: Escape and B, Tab, the arrows and the d-pad.

The first two run in build order, so call a shortcut after the widgets that should win its key. Because navigation runs last, a slider that takes the arrows keeps them, and the arrows move focus only when the focused widget has no use for them.

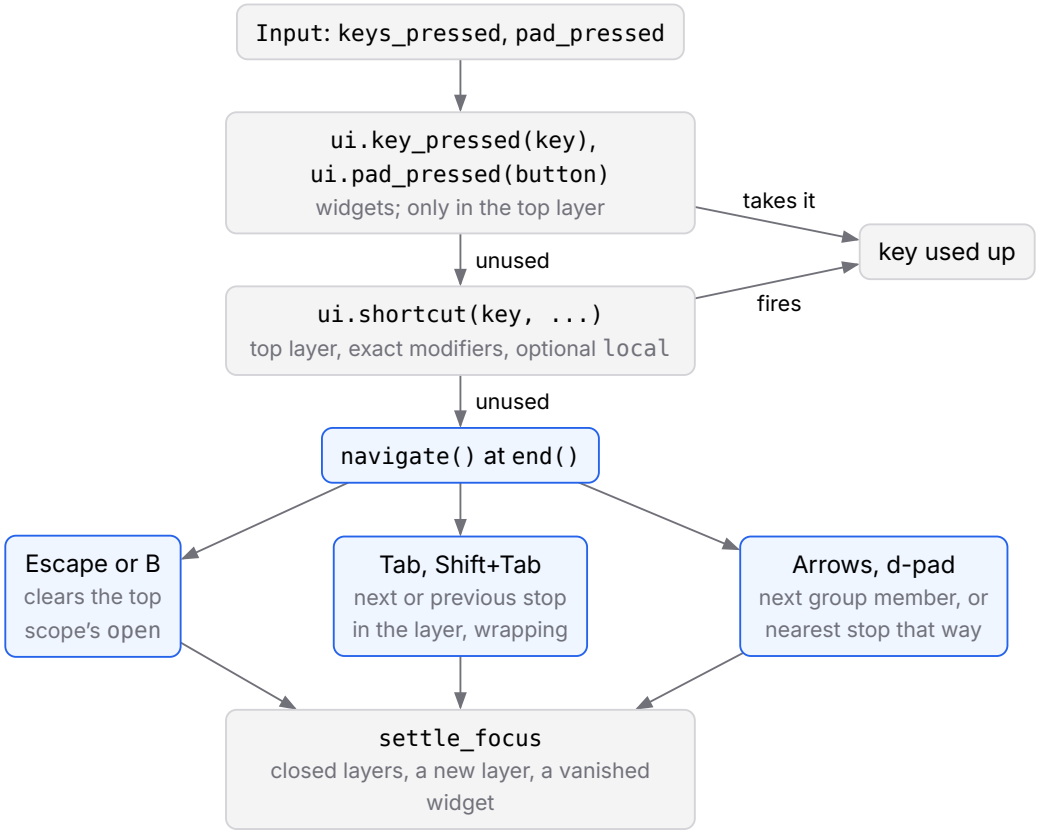


Figure 10: Where a key press goes. Widgets and shortcuts run during the build, in call order; navigation at `end()` sees only the keys nobody used, then focus settles.

`navigate()` handles the keys in that order within one `end()`, then settles focus: scopes that were not declared this frame give focus back to their opener, a layer that just appeared pulls focus in, and if the focused widget went away, focus moves to its neighbor.

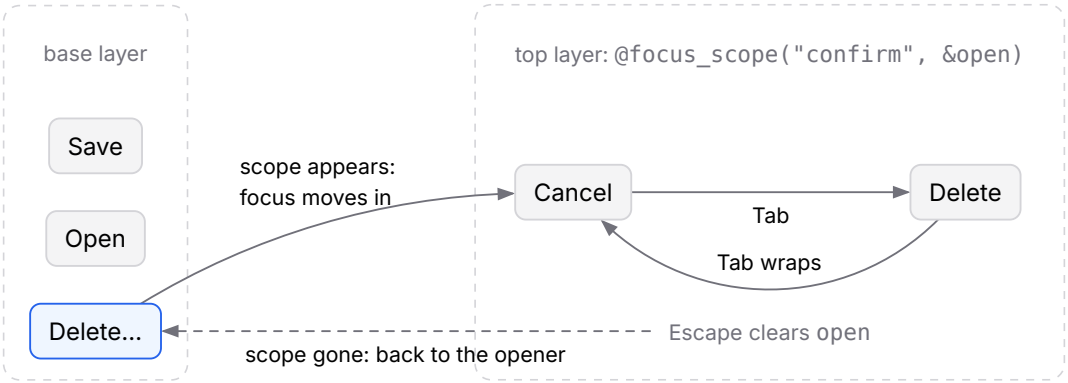


Figure 11: Layers. The confirmation dialog is a focus scope declared after the page, so it is the top layer: focus moves into it when it appears, Tab cycles among its stops, and when the scope is gone, focus returns to the button that opened it.

7.6.1 Typing in a text field

A focused text field needs most of the keyboard, and build order alone can't give it that: an app usually declares its shortcuts in a menu bar or toolbar built before the page, so `ui.shortcut(D)` would take the D key before the field is built. Kiln settles it in `ui.shortcut` itself. While a text field has focus, a shortcut stays quiet on any chord the field uses, wherever it is declared:

Chord	Why the field takes it
Letters, digits, Space and punctuation keys without the command modifier	Typing, Shift and Alt included; Ctrl+Alt is AltGr on Windows and Linux layouts, so it types too
Left, Right, Home, End, Backspace, Delete	Moving the caret, selecting, deleting by word; on PC keys, not with Alt
Enter without the command modifier or Alt	Submitting
The command modifier with A, C, V or X (no Shift), Z (with or without Shift), or Y (PC keys)	Select all, copy, paste, cut, undo, redo
The field’s own extra keys, without the command modifier or Alt	What that kind of field

Chord	Why the field takes it
	does with them: a search field holding text keeps Escape to clear it; a number field keeps Up, Down, Page Up and Page Down to step its value

The command modifier is Ctrl with PC keys and ⌘ with Mac keys, as `ui.text_keys` says; a chord with the other one (⌘ with PC keys, Ctrl with Mac keys) is never the field's. Other chords, such as Ctrl+S, Ctrl+Shift+C or F5, still reach their shortcuts, and so does Escape in a plain field or an empty search field. The text field registers itself while it is built, together with its extra keys (the `keys` of `FieldOpts`, passed to `Ui.field_ex` by the widget built on the field, which reads them after it), so the check uses the field that had focus last frame and still has it, with the extra keys it had then: in the first frame after focus moves into a field, the check doesn't apply yet, and a search field that has just been emptied keeps Escape for one more frame. The field takes only its editing keys with `Ui.key_pressed` it gets typed characters from `Input.text`, not from the letter keys. So Tab, Escape and the up and down arrows, which no shortcut takes while a field has focus, still reach navigation at `end()`, unless the field takes them as extra keys: Escape in a search field with text clears it instead of closing the dialog around it, and the arrows step a number field instead of moving focus.

While an input method composes in the focused field, every key is the input method's: `Ui.begin` marks every pressed key used and `Ui.typing_takes` takes every key, so no shortcut fires and neither Tab nor Escape reaches navigation (Input method composition).

The keys a field answers to follow the platform: `ui.text_keys` is `TextKeys.MAC` on macOS (Option moves by word, ⌘ by line and for commands) and `PC` elsewhere (Ctrl moves by word and runs commands). An app can pin it with `Ui.set_text_keys` (Following the system); Text editing lists the keys of each.

7.6.2 Selecting in a text

A selectable text holding a selection (`Ui.selected_text`) takes a few chords of its own, read with `Ui.key_pressed` while it is built:

Chord	What the selection does
The command modifier with A (no Shift)	Selects all the text
The command modifier with C (no Shift)	Copies the selection to the clipboard
Shift+Left, Shift+Right	Extends the selection by one grapheme; with the word modifier (Ctrl with PC keys, Option with Mac keys) by one word
Shift+Home, Shift+End	Extends the selection to the start or end of the text
Escape	Lets go of the selection, and of the keyboard

The command modifier follows `ui.text_keys` as in a text field: Ctrl with PC keys, where Ctrl+Alt is AltGr and so not a command, and ⌘ with Mac keys. With Mac keys, Shift+⌘ with an arrow is left alone.

As with a text field, build order alone would let a shortcut declared before the text take these keys first, so `ui.shortcut` asks before it fires. `Ui.typing_takes` first asks `Ui.selection_takes`, which answers for the chords above while a text holds the keyboard; only then does it check a focused text field. So a Ctrl+C shortcut declared in a toolbar before the text stays quiet while the text holds a selection, and Ctrl+C copies the selected words instead. Since the text uses these keys up, they don't reach navigation either: Escape lets go of the selection rather than closing the dialog around it. Copy and Escape are the text's only while it has something selected: after a plain click, which only places a point, Ctrl+C reaches its shortcut and Escape goes on to close the layer, while select all and Shift+arrows still work. A selected text that stops being built lets go at `end()`, and once the text lets go, every chord reaches its shortcut again. Text describes the selectable text itself, and Text editing the pieces it shares with the editor.

7.6.3 Keys in a list

A focused list or table (Collections) reads its keys with `Ui.key_pressed` while it is built, so it takes them only while it has focus, and the arrows it uses never reach navigation:

Key	What the list does
Up, Down, the d-pad	Moves the cursor row, and the selection with it
Home, End	Moves to the first or last row
Page Up, Page Down	Moves by a view in a long list, by ten rows in a plain list
Shift with a move	Extends a range from the anchor (multiple selection)
Ctrl with a move (⌘ with Mac keys)	Moves the cursor alone, leaving the selection (multiple selection)
Ctrl+Space (⌘ with Mac keys)	Toggles the cursor row in the selection (multiple selection); Space alone selects it

Key	What the list does
Ctrl+A (⌘A with Mac keys)	Selects all rows (multiple selection)
Enter, gamepad A	Activates the cursor row, as a double-click does
Menu, Shift+F10	Opens the cursor row's context menu under the row, selecting the row first if it is outside the selection
Alt+Up, Alt+Down	Moves the cursor row, or the selection it is in, one place (`reorder: true`, Reordering rows)

`Ui.list_activated` tells the app which row was activated. In a long list given a `text` callback, and in a table, typed letters move the cursor too, through type-ahead (below).

7.6.4 Keys in a chart

A chart (Charts) is one focus stop: Tab reaches the chart as a whole, not each point. While it has focus, it steps through its x positions, and the crosshair and tooltip follow, as they follow the pointer (Hover and keys):

Key	What the chart does
Left, Right, the d-pad	Moves to the previous or next x position; the first press starts at the first or last
Up, Down, the d-pad	In a horizontal bar chart (Bar charts), whose categories run down the side, moves to the previous or next category instead of Left and Right
Home, End	Moves to the first or last x position
Escape	Hides the crosshair and tooltip, until the next key
Ctrl/⌘ + and −	Zooms x in or out at the middle, on a zoomable chart
Ctrl/⌘ 0	Resets the zoom

With the pointer over the plot of a zoomable chart (`zoomable` in `ChartOpts`, Zoom and pan), Ctrl/⌘ and the wheel zoom at the pointer, Shift and the wheel or a sideways wheel pan, a drag pans, a pinch of two touches zooms, and a double-click resets. A plain wheel is left to the page, so a chart never traps scrolling (Scrolling).

While the pointer is over the plot, it decides which position is shown. `Ui.chart_keys` reads these keys with `Ui.key_pressed` while `Ui.chart` is built, so they reach the chart only while it has focus.

7.6.5 Type-ahead

In a list with focus, typing the start of an option's label jumps to it, as in native lists. `Ui.type_ahead(ta, labels, from)` reads this frame's `Input.text` and returns the index of the label it leads to, or −1 when nothing was typed or nothing matches. The widget keeps a `TypeAhead` in its state, holding the characters typed so far and when the last one came:

- Characters gather into one search while they come less than `ui::TYPE_AHEAD_PAUSE` (one second) apart; after a longer pause the next character starts a new search.
- A new search looks from the option after `from`, the current one, and goes round. A longer search may stay on the current option: with "Cambodia" and "Canada" in that order, "c" moves to "Cambodia", "ca" stays there, and "can" moves on to "Canada".
- The same letter typed again cycles: "c", "c", "c" visits each option starting with C in turn instead of searching for "ccc".
- Matching compares bytes and ignores ASCII case only.

The select uses it both closed, where typing changes the chosen option directly, and open, where typing moves focus among the rows (Inputs). `Ui.type_ahead_rows` is the same search over rows that aren't all built: it takes the row count and a `ListText` callback that returns a row's label, and searches in display order. A long list with a `text` callback and a table use it to move the cursor row; a table searches its `search_column`, else the sorted column, else the first (Tables). Type-ahead reads typed text, not keys, so it uses nothing up: the letter keys still go wherever they would have gone. It reads a frame's typing the frame after, once every shortcut has had its chance, so it shares letters with shortcuts wherever they are asked for: a search that is running keeps its letters, and a shortcut on a letter fires only between searches. A character a shortcut fired on is left out of the search, the rest of the frame's typing isn't, and so a search can't start with a shortcut's letter. A widget that uses typing itself, such as the Space that opens a select's list, says so with `Ui.use_typing`.

7.7 Scrolling

A scroll area, `Ui.@scroll`, takes the input that moves content: the wheel, touch and pen drags, its scrollbars, and focus moving by keyboard or gamepad. Areas nest, so each rule names which area a gesture goes to. Each area has a `ScrollAxis` (`VERTICAL`, `HORIZONTAL` or `BOTH`) and an `Overscroll` that says what it does with movement it can't use because it is already at its end: `AUTO` passes it to the enclosing area, `CONTAIN` keeps it, so nothing outside moves.

7.7.1 The wheel

A wheel gesture goes to the innermost area under the mouse that can still move that way, or that contains its overscroll; an area at its end with `AUTO` passes the gesture outward. Once chosen, the area keeps the gesture until the wheel pauses for `ui::WHEEL_LATCH_SECONDS` (0.3 s), so content that reaches its end mid-gesture doesn't hand the rest of the turn to the page around it. A vertical wheel turns a `HORIZONTAL` area sideways, and with Shift held the wheel scrolls sideways in any area.

One notch moves the content `Ui.wheel_step` points, `ui::WHEEL_STEP` (48) unless the OS says otherwise; `ui::WHEEL_PAGE` moves it one view per notch. On Windows the step follows the system's lines-per-notch setting at `ui::WHEEL_LINE` points a line. `Ui.set_wheel_step` fixes it for the app, and `Ui.follow_os` with `WHEEL_STEP` hands it back to the OS. The content eases to its new offset with the `SCROLL` motion.

A widget takes the wheel by clearing `ui.input.wheel` when it uses it: the scroll areas read it at the end of the frame, after every widget is built, so a widget that clears it first keeps the turn from the page. A zoomable chart clears it only for Ctrl/⌘ and the wheel, Shift and the wheel, or a sideways wheel over its plot, and leaves a plain wheel to scroll the page (Zoom and pan).

7.7.2 Touch and pen drags

A touch or pen drag that passes the drag threshold (`ui::DRAG_THRESHOLD_TOUCH` for touch, `ui::DRAG_THRESHOLD_MOUSE` for pen) scrolls the innermost area under the press that can move along the drag's main direction, or that contains its overscroll. The main direction is the larger of the horizontal and vertical movement, so a mostly vertical swipe across a horizontal strip scrolls the page around the strip. The area then lets go of the widget the drag began on, which doesn't click, and the content follows the finger. When the finger lifts, the content glides on at the drag's speed and slows with friction, losing about two thirds of its speed every `ui::FLING_DECAY_SECONDS` (0.325 s); it doesn't glide under reduced motion. A new touch on the area stops the glide.

A mouse drag never scrolls content: the mouse uses the wheel and the scrollbars.

A widget that drags, such as the slider, calls `Ui.claim_pointer` each frame it is held. A claimed pointer is never taken by a scroll area, so dragging the slider along its track moves the value, not the page. The scrollbars claim their pointer the same way.

7.7.3 Clipped widgets

A scroll area clips its content, as does a box with clipping on. The part of a widget that is clipped away doesn't take the pointer: a press, hover or wheel at a point outside the clip goes to whatever is visible there, not to the widget scrolled out of view under it.

7.7.4 Focus scrolls into view

When the keyboard or the gamepad moves focus (Tab, the arrows or the d-pad), each scroll area around the newly focused widget moves just enough to show it with a small margin, innermost area first. A click never scrolls this way: the widget a pointer pressed is already in view.

7.8 Drops

Files dragged from a file manager, and text such as a link dragged from a browser or a music app, can be let go over the window. The platform puts a drop in `Input` the frame it lands: `dropped` (the files' paths), `dropped_text` and `drop_pos`, and while something is dragged over the window, `dragging` and `drag_pos`. A box takes a drop that lands on it as the pointer finds widgets: on its rectangle from the frame before, and not where a scroll area clips it away, nor under another layer such as a dialog.

```
Id zone = ui.make_id("queue");
bool over = ui.drag_over(zone);   // for a highlight
ui.@box_ex({ .id = zone, .bg = over ? Token.ACCENT_SUBTLE : Token.SURFACE }) { ... };
foreach (path : ui.dropped_on(zone)) queue.add(path);
String link = ui.text_dropped_on(zone);
```

`Ui.dropped_files` and `Ui.dropped_text` give a drop wherever it landed. The paths and the text belong to the platform until its next poll, as typed text does, so an app that keeps them copies them. A drop wakes a sleeping App-style loop, as any event does. A Game-style app or a test puts its own drop in its `Input` with `Input.drop`.

What arrives depends on the platform. SDL3 reports all of it: the drag over the window and where, each file and the text. raylib on the desktop reports dropped files once they land, at the mouse, with no drag before and no text. A browser gives a page the drag and dropped text, but no paths for dropped files, which would need the in-memory file system the web builds leave out to stay small; there a dropped link arrives, and dropped files don't. `DropBuffer` is what a platform keeps of drops between its events and hands to `Input`.

7.9 Drag and drop inside the app

A widget becomes a drag source when `Ui.drag_source(kind, payload, label, count)` is called right after it, as a tooltip attaches to the widget just built (Widgets). Dragging it then carries `payload`, a pointer the app keeps, tagged with `kind`, a `DragKind` the app defines from 1 up. A box takes the drop as it takes files from the OS (Drops), through the same calls with a kind:

```
const DragKind SONGS = 1;

ui.@long_list("songs", count, 36, selection: &picked; int row)
{
    ui.text(titles[row]);
    bool many = picked.has(row) && picked.count > 1;
    ui.drag_source(SONGS, &picked, titles[row], many ? picked.count : 1);
};

Id queue = ui.make_id("queue");
```

```
ui.@box_ex({ .id = queue, .bg = ui.drag_over(queue, SONGS) ? Token.ACCENT_SUBTLE :
Token.SURFACE }) { ... };
if (try p = ui.dropped_item(queue, SONGS)) add_to_queue((Selection*)p);
```

- **Starting.** A mouse or pen drags once the press moves past the drag threshold, and the press then doesn't click. A touch rests for a long press first, so a quick swipe still scrolls; once lifted, its next move drags, and lifting without moving opens the widget's context menu if it has one, as on iOS.
- **The preview.** A small raised pill follows the pointer, showing `label` and, above 1, `count` in a badge. It is drawn above every layer, takes no pointer and never lags a frame.
- **Targets.** `Ui.drag_over(box, kind)` is true while a drag of that kind is over the box, for a highlight; `Ui.dropped_item(box, kind)` gives the payload on the frame it is let go there. A box is hit-tested as `interact` tests a widget: its rectangle from last frame, not where a clip hides it, and only in the layer the pointer is over, so a box under a dialog takes nothing. `Ui.drag_point` says where over the box the pointer is, for a target that shows where a drop would go, and `Ui.dragged_item` gives the payload anywhere while the drag is on.
- **Ending.** A release over no target of its kind, or Escape, ends the drag with nothing dropped. Escape is the drag's while it is on, so the menu or dialog under it stays open.
- **Scrolling.** While a drag is on, the innermost scroll area under the pointer scrolls when the pointer is within `ui::DRAG_SCROLL_EDGE` points of its edge, faster nearer the edge, up to `ui::DRAG_SCROLL_SPEED` points a second.

The drag belongs to the pointer, not the widget, so a row scrolled out of a long list, and no longer built, keeps its drag going. Kiln keeps only the pointer to the payload and copies the label; the payload must live until the drag ends. Dragging out of the app, to a file manager say, isn't possible: SDL3 reports drops but can't start a drag.

7.10 Who has the keyboard

`Ui.wants_keyboard` tells a Game-style loop whether the UI used a key, has visible focus, has a text field focused, even one focused by a click, or has a text holding a selection, so typing a name doesn't also move the player; Game style: your raylib loop describes it with `Ui.wants_pointer`.

7.11 The focus ring

The focus ring shows only while focus is being moved by keyboard or gamepad. The context keeps that in `ui.focus_visible`: navigation and activation set it, and any pointer press clears it, while focus stays where it is. So a mouse user who clicks a button sees no ring, and the moment they press Tab, the ring appears.

A widget learns it should draw the ring from `Interact.ring` (focused, and `focus_visible`). It animates a value toward 1 or 0 with the `FOCUS` motion (Motion) and draws `DrawList.focus_ring(rect, ui.c(FOCUS_RING), radius, t)`: a 2 pt outline 2 pt outside the rectangle, at opacity `t`.

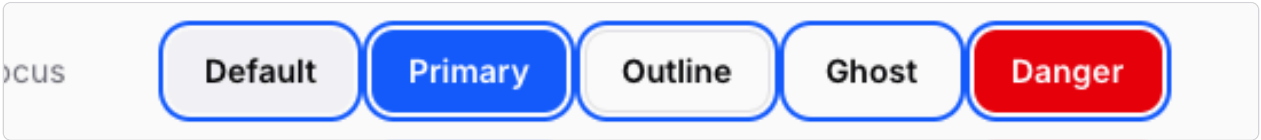


Figure 12: The Focus row: each variant's 2 pt `FOCUS_RING` outline sits 2 pt outside the button.

8 State and memory

An immediate-mode library rebuilds the whole interface every frame, so most of what it touches lives for one frame and then disappears. A few things have to last longer: what a widget remembers between frames, and the caches that keep text from being shaped and rasterized again. This chapter describes how long each kind of memory lives, where it comes from, and how Kiln keeps a steady frame from allocating at all.

8.1 Lifetimes

Kind	Lifetime	Where it lives
Paint data, formatted labels, scratch	one frame	The frame arena, reset at `Ui.begin`
Draw list (vertices, indices, commands, clips)	reused every frame	Lists cleared at the start of each layout and never shrunk; they grow when a frame needs more
Clay layout	one frame	Clay's arena, one block from the `Ui`'s allocator (see Layout for its size and overflow)
Last frame's rectangles	while the widget is laid out, plus `ui::STATE_RECT_FRAMES` frames	An id table
Widget state (springs, scroll, edits)	while `Ui.@state` asks for it, plus `ui::STATE_DATA_SECONDS`, or until released when kept	An id table pointing at one allocation per state struct
Shaped text	until unused for a number of frames	The shaped-text cache (see Text)
Rasterized glyphs	until the atlas fills, then the whole atlas is cleared	The glyph atlas, kept on the CPU and uploaded as textures (see Text)
Images' pixels and textures	until `Ui.image_destroy` or `Ui.destroy`	A table of images, one allocation per level, from the `Ui`'s allocator

An image's pixels are copied when `Ui.image_create` makes it, so the app can free its own. Each smaller level, halved on the CPU the first time the picture is drawn at half its size or less, is kept beside them: a picture drawn at a quarter of its size costs about a third more memory than its pixels alone. `Ui.image_destroy` frees an image's pixels, levels and textures, and `Ui.destroy` frees every image left. A level's texture is uploaded the first time it is drawn (Images); after the renderer loses its GPU context, `Ui.forget_image_textures` drops the handles and each level is uploaded again from its kept pixels the next time it is drawn.

The id tables are Kiln's own `IdMap`: open addressing with linear probing, and backward-shift deletion so a removal leaves no tombstone behind. Clearing a table keeps its slots. Once a table has grown to hold a frame's ids, inserting and removing ids allocates nothing, which the standard `HashMap` can't promise, since it allocates an entry per insert.

8.2 Widget state

Two things survive between frames for each id: the rectangle the widget had in the last layout, which `Ui.interact` tests input against (see Widgets), and the widget's state struct. A widget asks for its state with `ui.@state($Type, id)`:

```
macro Ui.@state(&self, $Type, Id id) => ($Type*)self.state_ptr(id, $Type::typeid,
$Type::size);

<*
 `Ui.@state` for state that owns memory: `drop` frees it before the state
 is collected or the `Ui` is destroyed.

 @require !self.has_state(id) || self.state_type(id) == $Type::typeid : "this id already
has state of another type"
*>
macro Ui.@state_owning(&self, $Type, Id id, StateDrop drop) =>
($Type*)self.state_ptr(id, $Type::typeid, $Type::size, drop);

fn void* Ui.state_ptr(&self, Id id, typeid type, usz size, StateDrop drop = null)
{
    if (try entry = self.states.get_ref(id))
    {
        if (entry.type == type)
        {
            entry.last_used = self.time;
            return entry.data;
        }
        // Another kind of widget had this id (the contract on `Ui.@state` reports
        // it where contracts are on): its state goes, rather than being read as this
type.
        if (entry.drop) entry.drop(entry.data);
        alloc::free(self.allocator, entry.data);
    }
    void* data = alloc::calloc(self.allocator, (sz)size);
    self.states.set(id, { .data = data, .type = type, .last_used = self.time, .drop =
drop });
    return data;
}
```

The first call for an id allocates the struct zeroed, from the `Ui`'s allocator; every later call returns the same memory and marks it used. An id keeps one state type: asking for a different type with the same id fails the macro's contract, and where contracts are off it frees the old state (running its drop) and starts the new one zeroed.

At `Ui.end` the `Ui` collects what has gone unused. The two timeouts are named constants:

```
<* A kept rect is dropped after this many frames without being laid out (half a second
at 60 fps). *>
const int STATE_RECT_FRAMES = 30;
<* Seconds a widget's state (springs, scroll, edits) survives without `Ui.@state` asking
for it, unless `Ui.keep_state` holds it. *>
const float STATE_DATA_SECONDS = 5.0;
```

Rectangles are counted in frames, so a widget that stops being laid out loses its rectangle about half a second later at 60 frames per second. State is counted in the `Ui`'s time, which comes from

the `Input` (its `time`, or the sum of its `dt` values), so a scroll position or a half-typed edit survives a widget being hidden for a few seconds, for example while you switch tabs and back. Collection frees the state struct and removes its entry.

Some state must outlive the widget's absence for longer, such as a document's undo history. `Ui.keep_state` exempts an id from collection until it's called again with `false`:

```
fn void Ui.keep_state(&self, Id id, bool keep = true)
{
    if (try entry = self.states.get_ref(id)) entry.keep = keep;
}
```

It does nothing for an id that has no state yet, so call it after `@state`:

```
History* h = ui.@state(History, id);
ui.keep_state(id);
```

Kept state is still freed by `Ui.destroy`.

8.3 Allocators

Everything the library allocates in C3 comes from one `Allocator`, passed when the `Ui` is created. It defaults to the heap (`mem`):

```
Ui* ui = ui::create(allocator: &my_allocator, theme: themes::DARK);
Ui* game_ui = raylib::create_ui(allocator: &my_allocator);
```

The `Ui` wraps that allocator in a counting allocator and gives the wrapper to everything it builds: the frame arena, the id tables, the draw list, Clay's arena, fonts, the shaped-text cache and the glyph atlas. Every allocation the core makes through its allocator passes through the counter, which is how `Ui.stats` can report a frame's allocations; the backends' platform and renderer, and the C font code's own mallocs, are not counted.

8.3.1 The frame arena

The frame arena is a dynamic arena of 64 KiB pages. `ui.begin()` resets it, and resetting keeps its pages for reuse, so after the first few frames it allocates nothing. Memory that only has to last until the frame is drawn comes from here:

```
<* Zeroed memory for one `$Type` from the frame arena; it lives until the next
`Ui.begin`. *>
macro Ui.frame_new(&self, $Type)
{
    return ($Type*)self.frame_bytes((sz)$Type::size);
}
```

`Ui.frame_new` and `Ui.frame_array` call the arena directly. Going through C3's `Allocator` interface, as `alloc::new(&ui.frame_arena, T)` does, costs a dynamic method lookup per call, which is more than the allocation itself when thousands of widgets each allocate their paint data.

Widgets put their paint data here (see Widgets), and formatted text goes here too: `Ui.textf`, a slider's value label and `Ui.shortcut_label` all format into `&ui.frame_arena`. For a formatted label you pass to another widget, do the same:

```
ui.textf("%d items", count);
String label = string::format(&ui.frame_arena, "Delete %d", count);
if (ui.button(label, DANGER)) delete_all();
```

Labels need no copying otherwise. `Ui.text` shapes its string during the call and keeps the shaped result, never the string, so you never manage the ownership of a string you pass to a widget.

8.3.2 Why not `tmem`

C3's temporary allocator, `tmem`, is global to the thread, and in Game style your own code owns the loop. It may reset `tmem` with `@pool` between `ui.begin()` and `ui.end()`, which would free paint data the draw list still points at. The frame arena belongs to the `Ui`, so only `ui.begin()` resets it. The core's one use of `tmem` is scanning font directories for fallback faces; the raylib backend also uses it briefly to make C strings (window title, paths, clipboard). Each use sits inside its own `@pool`, where nothing escapes the scope.

Your own code can still format labels with `tformat`. In App style, `app::run` wraps each frame in a `@pool`, so that text lives until the frame is drawn and is freed after it. In Game style, wrap the whole frame yourself, from `ui.begin()` through `ui.draw()`; a pool that ends earlier frees text the draw still reads. Without a pool, every label formatted with `tformat` stays in `tmem` for the life of the program, and memory grows with each frame drawn.

```
while (!rl::window_should_close())
{
    @pool()
    {
        ui.begin();
        build(ui);       // may use string::tformat
        ui.end();
        rl::begin_drawing();
        ui.draw();
        rl::end_drawing();
    };
}
```

8.4 No allocations per frame

After warm-up, a steady frame makes no heap allocations. Memory is allocated only when something new appears or something grows: an id seen for the first time, a table or list outgrowing its capacity, new text to shape, a new glyph to rasterize. Everything else reuses what earlier frames allocated.

Tests hold the library to this, under a tracking allocator and through `ui.stats()` on every gallery page (Frame statistics).

`ui.destroy()` frees everything the `Ui` allocated, including an attached platform and renderer.

9 Motion

Everything that animates in Kiln is a **spring**: a value chasing a target, stepped once per frame. A hover tint, a switch knob, a focus ring and a slider thumb are each one `Spring`, a struct of three floats that lives in the widget's state (Widget state):

```
struct Spring
{
    float value;      // where the spring is now; what you draw with
    float velocity;   // units per second
    float target;     // where it is heading
}
```

A widget sets the target every frame and reads `value` back, usually a number from 0 to 1 that its paint function uses to blend colors or scale a shape. Because a spring keeps its velocity, changing the target in mid-flight bends the motion instead of restarting it: a hover that ends halfway through its fade turns smoothly back. Motion is always interruptible.

9.1 Duration and bounce

Physical springs are usually described by stiffness and damping, which are hard to choose by feel. Kiln follows Apple's model instead (WWDC 2023, "Animate with springs"): you give a **duration** d, roughly how long the motion takes, and a **bounce** b, how far it overshoots. With unit mass, `Spring.step` turns them into

$$k = (2\pi/d)^2, \quad c = (4\pi(1-b))/d$$

for the stiffness k and the damping c. The damping ratio is $c/\left(2\sqrt{k}\right) = 1 - b$, so bounce 0 is critically damped and never overshoots, and a positive bounce gives an underdamped spring that swings past the target and settles. The bounce of 0.15 the tokens use is subtle: it overshoots by well under 1%, enough to make a change feel quicker and more physical without visibly wobbling.

Each frame, the spring integrates the force $-k\left(x - x_{\text{target}}\right) - cv$ with semi-implicit Euler: velocity first, then position from the new velocity. A frame of length Δt is split into $n = \max(1, \lceil \Delta t / h_{\max} \rceil)$ equal substeps with $h_{\max} = 1/240$ s (`SUBSTEP`), so a long frame stays as stable as a short one. One frame steps the spring by at most `ui::MAX_STEP` (0.1 s): the first frame after an app has slept has the whole sleep as its Δt, and a motion that starts on it then plays over the frames that follow instead of jumping to its end. Once the spring is within 0.0005 of its target and slower than 0.01 per second, it snaps there and stops; a settled spring costs nothing to step.

9.2 Springs find their frame

Widgets animate through `Spring.go`, which takes a motion token rather than a time step:

```
fn void Spring.go(&self, float target, Motion motion)
{
    self.target = target;
    Ui* ui = current;
    if (!ui || ui.reduced_motion)
    {
        self.value = target;
        self.velocity = 0;
        return;
    }
```

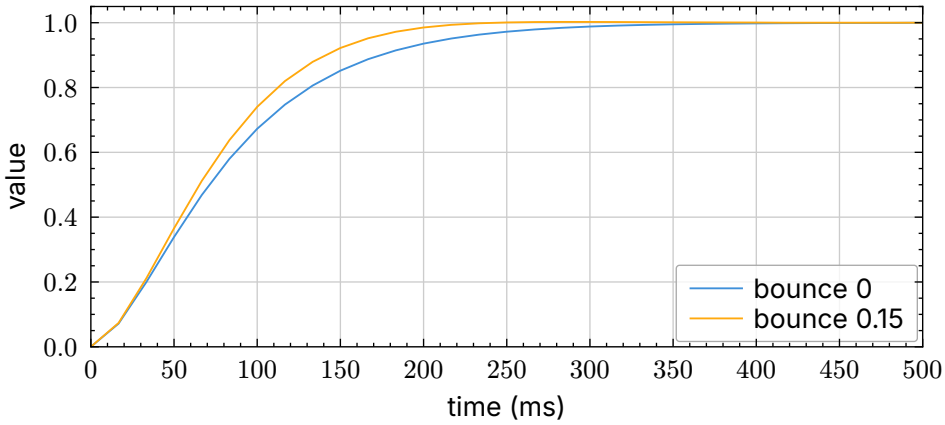


Figure 13: A spring with a duration of 280 ms going from 0 to 1, at 60 frames per second. The curves are computed with the same formulas, substeps and settle rule as `Spring.step` in `src/ui/spring.c3`. Bounce 0 approaches the target without passing it; bounce 0.15 gets there sooner, peaks at 1.002 and settles.

```
    self.step(ui.dt, motion.duration, motion.bounce);
    if (!self.settled()) ui.animating = true;
}
```

The time step comes from the `Ui` itself. Between `Ui.begin` and `Ui.end`, the `Ui` stores a pointer to itself in a thread-local variable (`tlocal Ui* current` in `src/ui/ui.c3`), much as Clay keeps its current context. `go` reads `ui.dt` through that pointer, so a widget needs no extra argument to animate, and each thread that draws UI has its own current `Ui`. Outside a frame there is no current `Ui`, and `go` jumps straight to the target.

While a spring has not settled, `go` sets `ui.animating`. `begin()` clears the flag, so after `end()` it says whether any spring moved this frame.

A custom widget uses springs the same way the built-in ones do. The sidebar row in `examples/hello/main.c3` keeps three springs in its state and hands their values to its paint function:

```
RowState* st = ui.@state(RowState, id);
st.hover.go(r.hovered ? 1 : 0, HOVER);
st.selected.go(demo.selected == i ? 1 : 0, ROW);
st.ring.go(r.ring ? 1 : 0, FOCUS);
```

9.3 Motion tokens

Durations and bounces are not chosen per widget. Like colors and spacing (see Theme), they come from a small set of named tokens, the `Motion` enum, so the same kind of change moves the same way everywhere:

```
enum Motion : char (float duration, float bounce)
{
    ROW        { 0.100, 0 },       // a selected row, a key step, a press
    HOVER      { 0.140, 0 },       // hover color, edges
    FOCUS      { 0.140, 0 },       // focus ring appearing
    STATE      { 0.180, 0.15 },    // check, selection, switch
    DISCLOSURE { 0.320, 0 },       // content opening, such as a section expanding
    POPOVER    { 0.280, 0.15 },    // a small floating layer appearing
    DIALOG     { 0.340, 0.15 },    // a modal layer appearing
    EXIT       { 0.160, 0 },       // any floating layer leaving
    SCROLL     { 0.200, 0 },       // a scroll area settling after a wheel step or a
```

```
jump
}
```

Small, frequent feedback is fast and flat: `ROW` for a selected row, a slider thumb moved by keys, and the press of a button or slider (a held thumb follows the pointer exactly, with no spring), `HOVER` for hover colors and edges, `FOCUS` for a focus ring appearing. A change of state the user caused, such as a check mark or a switch knob, uses `STATE`, with a little bounce to make it feel physical. The remaining tokens are for larger movements, which take longer:

- `DISCLOSURE` for content opening and closing. `Ui.@disclosure` slides its body's height with it (Navigation), and a scroll area's scrollbar fades out with it when the area has been still for a moment with the pointer away, though it comes back at `HOVER` speed.
- `POPOVER` and `DIALOG`, both with bounce, for a layer appearing (Layers): a modal layer appears with `DIALOG`, any other with `POPOVER`. The spring fades the layer in and slides it a few points into place, and dims a modal's backdrop.
- `EXIT`, short and flat, for a layer leaving, which should get out of the way. The layer stays on screen, fading, until the spring is nearly at 0.

9.4 Colors

Most springs drive colors. A spring going from 0 to 1 blends between two colors resolved from theme tokens with `Color.mix`, which interpolates in OKLab so the midpoint looks halfway. A button, for example, mixes its fill toward its hover fill by the hover spring, then toward its pressed fill by the press spring:

```
Color fill = p.fill.mix(p.fill_hover, p.hover).mix(p.fill_press, p.press);
```

The paint function receives both the colors (from `Ui.c`) and the spring values; see Rendering for how paint functions run.

9.5 Looping motion

A spring settles; some motion should not. While something waits, a spinner turns and a skeleton breathes, for as long as the wait lasts (Loading looks). Such motion is a `Loop`, a token with a period, like the motion tokens:

```
enum Loop : char (float period)
{
    SPIN       { 0.9 },        // a spinner's turn
    PULSE      { 1.6 },        // a skeleton's breath
    SLIDE      { 1.4 },        // an indeterminate progress bar's pass
    PING       { 1.5 },        // a notification dot's ring, growing and fading
}
```

`SPIN` is a spinner's turn, 0.9 s; `PULSE` is a skeleton's breath, 1.6 s; `SLIDE` is an indeterminate progress bar's pass, 1.4 s; `PING` is a notification dot's ring, growing and fading, 1.5 s (Status). `Ui.loop_phase(loop)` says where the loop is this frame, from 0 to 1 through its period, computed from `ui.time`, so every spinner on screen turns in step and a loop needs no state of its own. Each call sets `ui.animating`, so frames keep coming exactly as long as a widget asks for the phase; a spinner that is no longer built lets the loop sleep again.

9.6 Idle

An app that is not changing should not use the CPU. After `end()`, `ui.animating` says whether any spring moved or loop was asked for; the App-style loop sleeps on frames where it is false and no input arrived (App style: Kiln owns the loop). While anything moves, it draws every frame.

9.7 Reduced motion

Setting `ui.reduced_motion` makes every spring jump straight to its target, with no velocity and without marking the `Ui` as animating. Feedback stays: hover, selection and focus still change color, only at once instead of over time. Loops hold still: `loop_phase` returns a fixed phase, the start of a turn for `SPIN`, the middle of a breath for `PULSE` and 0 for `SLIDE` and `PING`, and asks for no frames, so a spinner shows a still mark and a skeleton a steady tint. A slide or a ring frozen partway would look like a fault, so widgets drop them instead: an indeterminate bar becomes a still wash of the fill color at quarter strength, and a notification dot stays without its ring (Status).

10 Theme

A theme decides every color, spacing step, corner radius and text size a `Ui` draws with; widgets name roles, and the theme supplies values.

Tailwind CSS shapes how Kiln's tokens and scales are **organized and named**, and its OKLCH palettes supply the primitive colors. Tailwind's class strings (`"p-4 bg-blue-500"`) are not used: they would need parsing at run time and would lose type checking, while named arguments and option structs give the same convenience with the compiler's help. `Ui.@column` with `pad: 4, gap: 2, bg: SURFACE` reads like a class list, but every name in it is checked when the program compiles.

10.1 Tokens

Colors come in two layers.

- **Primitives** are palettes: `themes::ZINC_50` ... `themes::ZINC_950`, and the same steps of Tailwind's other 21 palettes, from `SLATE_*` to `ROSE_*`, in `kiln::themes` (`src/themes/palette.c3`). The built-in themes use zinc, blue, red, green and amber; an app may pick from any of them. They are Tailwind CSS v4's values, transcribed as `Oklch` constants.
- **Semantic tokens** name a role, not a color. Widgets use only these, and a theme decides which primitive each one gets.

```
constdef Token : ushort
{
    SURFACE,           // window and page background
    SURFACE_RAISED,    // cards, panels, popovers
    BACKGROUND,        // controls' resting fill (fields, buttons' neutral variant)
    TEXT,              // body text
    TEXT_SECONDARY,    // quieter supporting text
    TEXT_DISABLED,     // text of disabled controls
    BORDER,            // outlines and separators
    TRACK,             // the unfilled part of switches, sliders, progress bars
    ACCENT,            // the brand color of primary and selected controls
    ACCENT_HOVER,      // ACCENT under the pointer
    ACCENT_PRESSED,    // ACCENT while pressed
    ACCENT_SUBTLE,     // tinted backgrounds: selected rows, soft buttons
    ON_ACCENT,         // text and icons drawn on ACCENT
    FOCUS_RING,        // the ring around the focused control
    DANGER,            // destructive actions and errors
    SUCCESS,
    WARNING,
    BACKDROP,          // dims the page behind a modal layer, at the theme's
backdrop_opacity
    SELECTION,         // the background of selected text; derived from ACCENT
    SUCCESS_SUBTLE,    // tinted backgrounds of success alerts and badges; derived from
SUCCESS
    WARNING_SUBTLE,    // the same for WARNING
    DANGER_SUBTLE,     // the same for DANGER
    SERIES_1,          // chart series, in order: eight hues apart enough to tell side
by side
    SERIES_2,
    SERIES_3,
    SERIES_4,
    SERIES_5,
    SERIES_6,
```

```
    SERIES_7,
    SERIES_8,
}
```

`ACCENT` comes with six **companions**: `ACCENT_HOVER`, `ACCENT_PRESSED`, `ACCENT_SUBTLE`, `ON_ACCENT`, `FOCUS_RING` and `SELECTION`, the background a text field paints behind selected text (Text editing). A theme can set them, but usually leaves them zero and lets Kiln derive them from the accent (see Colors in OKLCH), so one brand color restyles every accented control.

The status colors `SUCCESS`, `WARNING` and `DANGER` each have a soft **tint** in the same way: `SUCCESS_SUBTLE`, `WARNING_SUBTLE` and `DANGER_SUBTLE`, the backgrounds of alerts and badges, derived from their status color when the theme leaves them zero (Status tints).

`BACKDROP` is the color that dims the page behind a modal layer (Layers). It is drawn at the strength `Theme` holds in `backdrop_opacity`, from 0 for no dimming to 1 for an opaque cover, rising and falling as the layer fades in and out. The built-in themes dim with near-black `ZINC_950` at 0.4 in light and black at 0.6 in dark.

`SERIES_1` to `SERIES_8` color the series of a chart, in order (Chart series colors).

A `Theme` is a plain struct: whether it is dark, one `Oklch` per built-in token (`ThemeColors`, which a union also exposes as an array indexed by `Token` the eight series colors are its `series` array), the spacing base and the scales. The built-in light theme shows all of it:

```
const Theme LIGHT = {
    .dark = false,
    .colors = {
        .surface = ZINC_50,
        .surface_raised = ui::WHITE,
        .background = ZINC_100,
        .text = ZINC_950,
        .text_secondary = ZINC_500,
        .text_disabled = ZINC_400,
        .border = ZINC_200,
        .track = ZINC_300,
        .accent = BLUE_600,
        .danger = RED_600,
        .success = GREEN_600,
        .warning = AMBER_500,
        .backdrop = ZINC_950,
    },
    .backdrop_opacity = 0.4,
    .spacing_base = 4,
    .radius = { .control = 10, .panel = 14, .surface = 18 },
    .radii = { 2, 4, 6, 8, 12, 16, 24 },
    .text = { { 12, 16 }, { 14, 20 }, { 16, 24 }, { 18, 28 }, { 20, 28 }, { 24, 32 }, { 30, 36 }, { 36, 40 } },
    .breakpoints = { 0, 640, 768, 1024, 1280 },
};
```

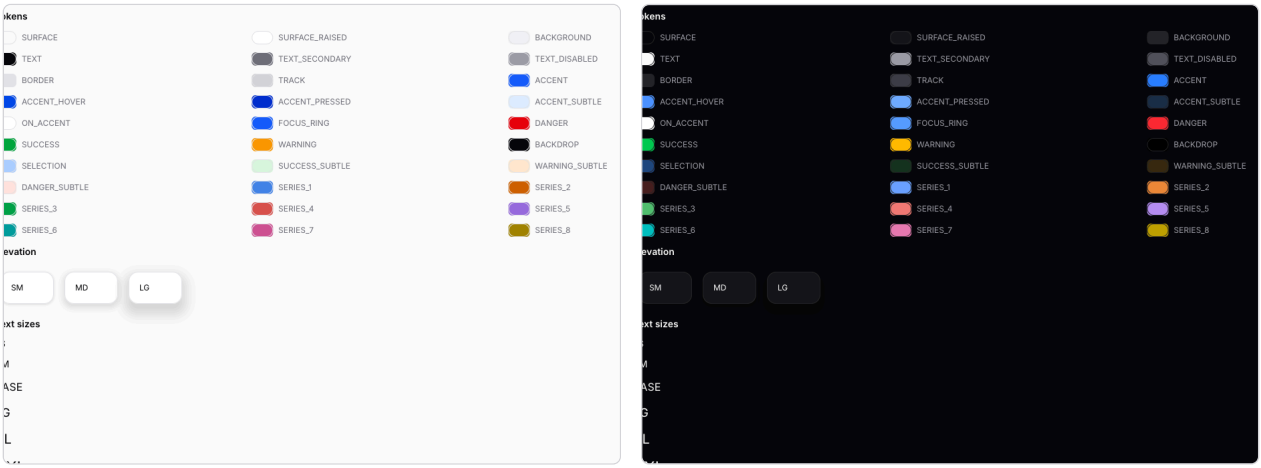


Figure 14: The gallery's theme page in the built-in light and dark themes. Each swatch is one semantic token as `Ui.c` resolves it; the six companions are derived from `ACCENT`: hover and pressed are darker in the light theme and lighter in the dark one, the subtle tint and the selection are pale in light and deep in dark, `ON_ACCENT` is white on both blues, and `FOCUS_RING` is the accent itself in light and a step lighter in dark. The last eight swatches are the chart series colors, a step lighter in dark. Below are the three elevations and the text scale from XS to XXXXL.

10.2 Colors in OKLCH

Themes are written in OKLCH: lightness from 0 to 1, chroma from 0 to about 0.4, and hue in degrees. It is a perceptual space, where equal steps look equal, so a shade derived from any brand color keeps its hue and its relative contrast. The conversions in `src/ui/color.c3` follow Björn Ottosson's OKLab definition.

The companions are derived in OKLCH from the accent's value, with steps that point the other way in dark themes:

Derived token	Light theme	Dark theme
`ACCENT_HOVER`	lightness −0.06	lightness +0.06
`ACCENT_PRESSED`	lightness −0.12	lightness +0.12
`ACCENT_SUBTLE`	lightness 0.95, a quarter of the chroma, same hue	lightness 0.30, a quarter of the chroma, same hue
`ON_ACCENT`	near-black (lightness 0.141) in the accent's hue when the accent's lightness is above 0.68, otherwise white	
`FOCUS_RING`	the accent	lightness +0.08
`SELECTION`	lightness 0.85, half the chroma, same hue	lightness 0.40, half the chroma, same hue
`SUCCESS_SUBTLE`, `WARNING_SUBTLE`, `DANGER_SUBTLE`	as `ACCENT_SUBTLE`, from `SUCCESS`, `WARNING` and `DANGER`	

A theme that sets a derived token keeps it: only a zero one is derived. `ui::DERIVED` lists every derived token, and `ui::derived_from` names the token each is made from: `ACCENT` for the companions, the status color for a tint.

Widgets draw with `Color`, 8-bit sRGB with alpha. Some OKLCH values, such as the most saturated palette steps, lie outside the sRGB gamut. Kiln maps them the way CSS Color 4 specifies, which is

also how browsers produce Tailwind's published hex values: it lowers the chroma, by binary search, until clipping what is left changes the color by less than a just-noticeable difference (0.02 in OKLab). A palette value therefore gives the same pixels in Kiln as in a browser.

10.3 Status tints

Alerts, badges and other status widgets (Status) pair a strong color with a soft one: the text, icon or border in `SUCCESS`, `WARNING` or `DANGER`, and the background in its tint, `SUCCESS_SUBTLE`, `WARNING_SUBTLE` or `DANGER_SUBTLE`. A `Tone` names the pair (`INFO` uses `ACCENT` and `ACCENT_SUBTLE`, `NEUTRAL` quieter text on a raised surface), and `Tone.strong` and `Tone.subtle` give its two tokens.

A tint is derived exactly as `ACCENT_SUBTLE` is from the accent: lightness 0.95 in light themes and 0.30 in dark ones, a quarter of the chroma, the same hue. The built-in themes leave the tints zero, so a theme that changes a status color gets a matching background for free, and a theme that sets a tint keeps it. A scoped override follows the same rule as the accent's: `ui.push_token(DANGER, …)` derives `DANGER_SUBTLE` again for the scope, and `pop_token` puts the previous tint back.

Some users ask the operating system for more contrast, and a soft tint behind colored text is exactly what they find hard to read. `Ui.high_contrast` holds that request as a flag widgets read while the frame is built. While the theme is `themes::SYSTEM` it follows the OS setting (Following the system); `Ui.set_high_contrast` pins it, and tests keep it off so screenshots don't depend on the machine. The status widgets read it to outline themselves at full strength in place of a faint border or none, and badges switch to the body text color (Status); other widgets, built-in or custom, can read it the same way.

10.4 Chart series colors

A chart (Charts) colors its series, in order, with `SERIES_1` to `SERIES_8`, and `Ui.series_color(i)` gives the color of series `i`, going round after the eighth. Kiln's palette spaces eight hues around the OKLCH circle, `ui::SERIES_HUES`: blue, orange, green, red, purple, teal, pink and yellow. All eight share one lightness and one chroma, 0.62 and 0.17 in light themes and 0.72 and 0.15 in dark ones, so no series looks more important than another; only the hue tells them apart. `ui::default_series(i, dark)` returns one of them.

The built-in themes leave `series` zero, and setting a theme fills each series it leaves zero with Kiln's palette for light or dark, so a theme can set all eight colors, some of them, or none. As with any token, `ui.push_token(SERIES_1, …)` recolors one series for a scope.

10.5 Resolution

Converting OKLCH to sRGB costs cube roots, powers and possibly a gamut search, far too much for every lookup. The `Ui` converts each token once, when the theme or an override changes, and keeps two parallel tables: the `Oklch` value of every token, and its resolved `Color`. The built-in tokens come first; custom tokens follow them, so a custom `Token` is just the next index.

`ui.c(token)` returns the resolved `Color`; it is an inline function that indexes the table. Color methods derive variants from it without touching the theme: `ui.c(ACCENT).alpha(0.5)` scales the opacity, and `ui.c(ACCENT).mix(ui.c(TEXT), 0.2)` blends a fifth of the way toward the text color, in OKLab so the midpoint looks halfway. `Oklch.lighten` steps lightness at the same hue and chroma, which is how the hover, pressed and focus companions are made.

10.6 Scales

Sizes are named steps too, and the theme holds their values.

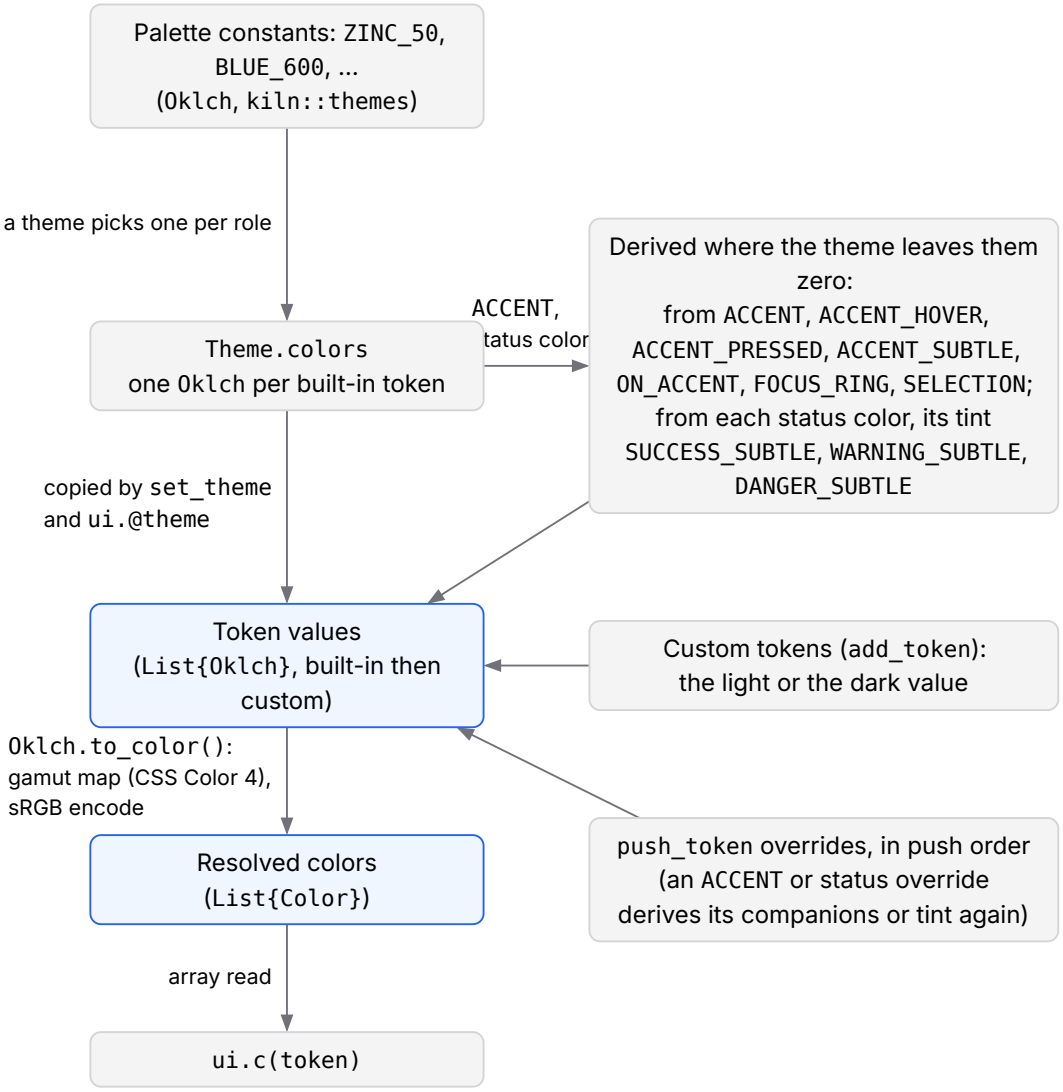


Figure 15: How a token gets its color. `Ui.set_theme` and both ends of a `Ui.@theme` scope resolve every token; `Ui.push_token` and `Ui.pop_token` change one token (plus the companions, for `ACCENT`, or the tint, for a status color); `Ui.add_token` adds one. Everything else, `ui.c(token)` included, only reads the table.

- **Spacing** has one base: `Ui.space(n)` is `n × spacing_base` points, 4 in the built-in themes (`ui.space(4)` is 16 pt). Containers' `pad` and `gap` count spacing steps, so changing the base makes every layout denser or looser at once.
- **Radii** come in two forms. `Radii` (`theme.radius`) holds three named radii: `control` (buttons, and boxes with `radius: CONTROL`), `panel` (cards and `PANEL` boxes) and `surface` (`SURFACE` boxes); switches are pills and checkboxes use a fixed radius. Boxes pick one with `Corners`: `NONE`, `CONTROL`, `PANEL`, `SURFACE` or `FULL` (a pill). The `Radius` scale (`theme.radii`, read with `Ui.radius`) is a general scale for custom drawing.
- **Text sizes** (`TextSize`) each pair a size with a line height (`TextStyle`), read with `Ui.text_style`.
- **Elevations** (`Elevation`: `NONE`, `SM`, `MD`, `LG`) select a shadow for a box or card; their blur, offset and opacity are fixed in the drawing code (Shadows and elevations).
- **Breakpoints** (`Breakpoint`: `BASE`, `SM`, `MD`, `LG`, `XL`) are the layout widths at which a responsive layout changes; Layout shows how layouts use them.

Scale	Steps	Values in the built-in themes
`Radius`	`XS` … `XXXL`	2, 4, 6, 8, 12, 16, 24 pt
`Radii`	`control`, `panel`, `surface`	10, 14, 18 pt
`TextSize`	`XS`, `SM`, `BASE`, `LG`, `XL`, `XXL`, `XXXL`, `XXXXL`	12/16, 14/20, 16/24, 18/28, 20/28, 24/32, 30/36, 36/40 pt (size/line height)
`Breakpoint`	`BASE`, `SM`, `MD`, `LG`, `XL`	0, 640, 768, 1024, 1280 pt

10.7 Presets

`kiln::themes` ships `themes::LIGHT` and `themes::DARK`: zinc surfaces with a blue accent. They are plain C3 constants, so a theme of your own is a value that extends one with initializer splatting. `DARK` itself is `{ ...LIGHT, .dark = true, .colors = { ... } }`, and a brand variant changes only what differs:

```
Theme brand = { ...themes::LIGHT, .colors.accent = themes::GREEN_600 };
ui.set_theme(brand);
```

The companions follow the new accent because `LIGHT` leaves them zero. `ui::create` takes the starting theme as its `theme:` argument (`LIGHT` by default), and `ui.set_theme` switches it later.

A third preset, `themes::SYSTEM`, is not a theme but a request to follow the operating system. A `Ui` given it applies `themes::from_os`, which is `DARK` or `LIGHT` to match the user's setting with the system accent in place of the blue (all six companions cleared so they are derived from it), and keeps following the setting as it changes (Following the system). On macOS it also sets `backdrop_opacity` to 0, as macOS doesn't dim a window behind its sheets and alerts. It is the default of `app::run` and `raylib::create_ui`; passing any other theme, or calling `ui.set_theme` with one, stops the colors following, but not the OS's settings (Following the system). `ui.set_themes` follows the OS with an app's own light and dark palettes. The built-in demo's theme editor builds a theme the same way and shows its accent, spacing and radii as `Theme` fields, ready to paste (The built-in demo).

10.8 Customizing

Controls never take raw colors. There are four ways to customize, from most to least preferred.

10.8.1 Variants

A variant names a meaning, not a color: `Ui.button` with `variant: DANGER`. The button draws with `DANGER`, so the theme still decides what danger looks like.

10.8.2 Scoped token overrides

`push_token` overrides one token until the matching `pop_token`. Everything drawn in between follows, including hover, pressed and focus shades: overriding `ACCENT` derives its six companions again from the new value, even ones the theme had set explicitly, and `pop_token` restores them. Overriding `SUCCESS`, `WARNING` or `DANGER` derives its tint again the same way.

```
ui.push_token(ACCENT, brand_purple);
defer ui.pop_token();
```

Overrides form a stack. `set_theme` clears it, so set the theme outside any push.

A whole theme can be scoped as well. `ui.@theme(themes::DARK) { ... }` resolves every token from the given theme for the body and puts the outer theme back afterwards, replaying the overrides pushed around it. Inside the body, the scoped theme's own values apply, not the outer overrides,

and any token pushed inside must be popped inside. Because widgets read their colors while the frame is built, a single frame can show light and dark side by side, such as a preview of the other theme in a settings page.

10.8.3 Custom tokens

A color that means something in your application, but not in Kiln, becomes a token of its own. `add_token` registers it once, with a value for light themes and one for dark themes, and returns the `Token` to use. `add_token` takes a name for readability at the call site; Kiln doesn't use it.

```
Token now_playing = ui.add_token("now_playing",
    { 0.656, 0.241, 354.3 }, { 0.718, 0.202, 349.8 });
```

A custom token resolves like a built-in one: switching between a light and a dark theme picks its other value, and `push_token` can override it.

10.8.4 The escape hatch

Where the color is the content, the call takes a `color:` argument: text and icons take a `Token` there, built-in or custom.

```
ui.text(song.title, color: now_playing);
ui.icon(HEART, color: DANGER);
```

Custom paint functions draw with `Color` values, so they read them with `ui.c(token)` while the frame is built and keep them in their paint data (Rules).

Not allowed: a background or border color on a control, such as `ui.button("Play", bg: ...)`. Controls have no such argument. Override `ACCENT` for the scope instead, and the button's hover, pressed and focus shades follow.

11 Layout

Clay does the layout math. Each `Ui` owns a Clay context: between `Ui.begin` and `Ui.end`, containers and widgets declare Clay elements, and `end()` runs the layout, keeps the rectangle of every element that has an id (where the next frame's `Ui.interact` finds it) and turns Clay's render commands into the draw list (Rendering). Kiln adds names and helpers on top, borrowing Tailwind's vocabulary.

11.1 From Tailwind to options

Tailwind	Kiln
`w-full`, `grow`	`width: ui::GROW`
`w-fit`	`width: ui::FIT` (the default)
`w-64` / `w-[137px]`	`width: ui::px(256)` / `ui::px(137)`
`w-1/2`	`width: ui::pct(50)`
`min-w-*`, `max-w-*`	`width: ui::grow(min: 200, max: 640)`, or `ui::fit(min:, max:)`
`p-4`, `px-2`, `py-1`, `gap-2`	`pad: 4, pad_x: 2, pad_y: 1, gap: 2` (spacing steps)
`items-center`, `justify-end`	`align: CENTER, justify: END`
`justify-between`	`Ui.spacer` between the children (Clay has no space-between)
`flex-col md:flex-row`	`Ui.@stack(Ui.at(MD, Dir.ROW, Dir.COLUMN))`
`@container`, `@sm:flex-row`	`ui.width_at(320, Dir.ROW, Dir.COLUMN)` inside a box with an id (Container queries)
`container mx-auto`, `max-w-prose`	`ui.@container(max: ui::CONTAINER_XL)`, `ui.@container(max: ui::PROSE)`

Tailwind	Kiln
`bg-*`, `rounded-*`, `border`	`bg: SURFACE_RAISED, radius: PANEL, border: true` (tokens only)
`bg-linear-to-b from-… via-… to-…`	`.gradient = &g` in `BoxOpts`, with `g = ui::gradient3(ui::TO_BOTTOM, from, via, to)` (Gradients)
`shadow-*`	`Ui.@card(shadow: MD)`, or `.shadow` in `BoxOpts`
`overflow-hidden`	`clip: true`
`overflow-x-clip`	`.clip_x = true` in `BoxOpts`: the box keeps its content's height, as `@sidebar` does
`overflow-y-auto`, `overflow-x-auto`	`Ui.@scroll(id)`, `ui.@scroll(id, axis: HORIZONTAL)` (Scrolling)
`object-fill`, `object-contain`, `object-cover`, `object-none`, `object-scale-down`	`Ui.image(..., fit: FILL)`, `fit: CONTAIN`, `COVER`, `NONE`, `SCALE_DOWN` (`ImageFit` `FILL` is the default)
`object-center`, `object-top`, `object-left-bottom`, …	`position: CENTER, TOP, LEFT_BOTTOM, …` (`ImagePosition` `CENTER` is the default)
`aspect-square`, `aspect-video`, `aspect-[3/2]`	`aspect: 1`, `aspect: 16.0 / 9`, `aspect: 1.5` (on `ui.image`; width divided by height)

Spacing options count steps of the theme's spacing base, as Tailwind's do (Scales). Sizes are points: `px(n)` is exactly `n`, `pct(n)` is `n` percent of the parent less its padding and gaps, and `grow` shares the room left with growing siblings. Backgrounds, corners and borders take theme tokens, never raw colors (Theme). A gradient's stops are colors instead, so one can start from a picture's, such as an album cover's, and end in a token's, `ui.c(BACKGROUND)`.

`align` places children across the container's direction and `justify` along it: in a row, `justify` is horizontal and `align` vertical; in a column, the other way around.

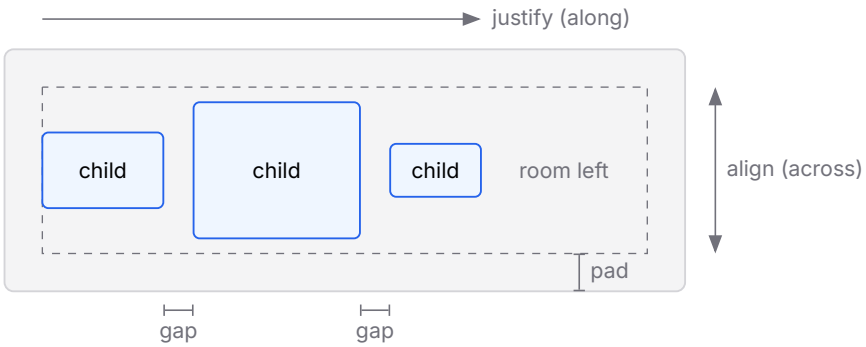


Figure 16: A `Ui.@row(pad: 2, gap: 2, align: CENTER)`: padding on every side, gaps between children, children centered across the row and placed from the start along it (`justify: START`). The room left stays empty unless a `ui.spacer()` takes it.

11.2 Containers

Containers are method macros with a trailing body: `ui.@row(...)`, `Ui.@column(...)`, `ui.@stack(dir, ...)` (a row or a column chosen by `dir`) and `ui.@card(...)` (a raised panel: `SURFACE_RAISED`, panel corners, a hairline border and a small shadow, laid out as a column with some padding and gap by default):

```
ui.@column(pad: 8, gap: 4)
{
    ui.@card(width: ui::GROW) { ... };
};
```

Each container declares its options as named parameters, with defaults:

```
macro void Ui.@row(&self, String id = "", Size width = FIT, Size height = FIT,
    float pad = 0, float pad_x = 0, float pad_y = 0, float gap = 0,
    Align align = START, Align justify = START, Token bg = NO_TOKEN,
    Corners radius = NONE, bool border = false, bool clip = false; @body())
```

C3 requires named arguments in declaration order, so a call lists them in the order of the declaration: `id, width, height, pad, pad_x, pad_y, gap, align, justify, bg, radius, border, clip` for `@row` and `@column`. `@stack` takes only `dir, id, width, height, pad, gap, align, justify`; `@card` takes `id, width, height, dir, pad, gap, align, justify, shadow`.

Every container fills in a `BoxOpts` and passes it to `Ui.@box_ex(opts) { ... };`. C3 can't forward named arguments through a variadic macro, so this struct is how the containers share their options, and it is everything a box can be:

```
struct BoxOpts
{
    Id id;              // NO_ID: anonymous, no rect is kept
    Size width;
    Size height;
    Dir dir;
    float pad;          // spacing steps on every side
    float pad_x;        // overrides pad left and right when not 0
    float pad_y;        // overrides pad top and bottom when not 0
    float lead;         // points added to the left padding, for what the box's paint
```

```
draws there (a toggle's control)
    float gap;         // spacing steps between children
    Align align;       // across the direction
    Align justify;     // along the direction
    Token bg;          // NO_TOKEN: nothing drawn
    Gradient* gradient; // a linear gradient in place of `bg` (Tailwind's `bg-linear-
*`); copied when the box opens
    Corners radius;
    Elevation shadow;  // drawn only when bg or gradient is set
    bool border;       // a 1 pt BORDER outline
    bool clip;         // hide what overflows
    bool clip_x;       // hide only what overflows left and right, so the box keeps its
content's height
    PaintFn paint;     // draws under its children, over its own bg or gradient
    void* paint_data;
}
```

Widgets build a `BoxOpts` directly and call `@box_ex`, which reaches the fields the containers leave out. A field left out of the initializer is zero, and token zero is `SURFACE`, so a box that draws no background says `.bg = ui::NO_TOKEN`. With an `id`, the box's rectangle is kept for `interact()`.

A box can draw its own background with `.paint = &fn, .paint_data = p`, leaving `bg` at `NO_TOKEN`; with a `bg` or `gradient` too, the paint draws over that background and under the children (From Clay to the draw list).

Two helpers fill the space between children:

- `ui.spacer()` takes the room left along the parent's direction (it takes a `Size`, `GROW` by default). Between two children it pushes them to the two ends, as `justify-between` does.
- `Ui.separator` draws a 1 pt `BORDER` hairline across the parent's direction: a vertical line in a row, a horizontal one in a column.

`Size` covers fit, grow, fixed and percent sizes; fit and grow take a minimum and a maximum. Clay never wraps: children that overflow a row stay on one line. A row that may not fit becomes a column at narrow widths (Responsive layout); children that should wrap onto new rows go in a `Ui.@grid`, a Tailwind-style grid that places its cells itself: a set number of columns (`cols:`) or as many as fit (`min_cell:`), cells spanning columns and rows, and gaps between them (Grids).

11.3 The element budget

Clay works in a fixed arena sized for a maximum number of elements. `ui::create(max_elements: ...)` sets that budget (8192 by default), and the arena comes from the `Ui`'s allocator (State and memory).

Clay doesn't reliably report running out: it stops taking elements, and at most calls its error handler (capacity exceeded, or unbalanced opens and closes once it has dropped some), which sets an overflow flag. So the `Ui` also counts every element it opens, boxes and custom elements alike (text and icons are custom elements). At the end of a frame whose count reaches one less than the budget, or after Clay reported a capacity error, `end()` skips the rest of the layout: no rectangles are kept and the draw list stays empty, so that frame draws nothing.

The next `begin()` destroys the Clay context, doubles the budget until it exceeds one and a half times the number of elements the last frame opened, and creates Clay again at that size. In safe builds it prints `kiln: layout overflowed; Clay now holds N elements` to standard error; release builds stay silent. Clay's capacity and unbalanced-open errors only set the flag; its other errors are

printed, in safe builds only. The budget never shrinks, so one large frame is enough to size it for the rest of the run. `Ui.stats` reports each frame's element count (Frame statistics).

11.4 Responsive layout

Responsive layout falls out of immediate mode: the layout code runs every frame, so it can ask how wide the window is. `Ui.bp` is the widest breakpoint the layout width reaches: `BASE` below `SM`, then `SM`, `MD`, `LG` and `XL`, whose widths the theme sets (Scales). `ui.at(bp, wide, narrow)` picks `wide` at `bp` and above, `narrow` below it:

```
ui.@stack(ui.at(MD, Dir.ROW, Dir.COLUMN), width: ui::GROW, gap: 4)
{
    ui.@card(width: ui.at(MD, ui::pct(33), ui::GROW)) { ... };
    ui.@card(width: ui::GROW) { ... };
};
if (ui.bp() >= LG) sidebar(ui);
```

`at` is a macro over untyped values, so its arguments name their type (`Dir.ROW`, `Align.CENTER`) where a bare enum value would be ambiguous. Any option can change with the width: a direction, a size, an alignment, or whether a part is drawn at all.

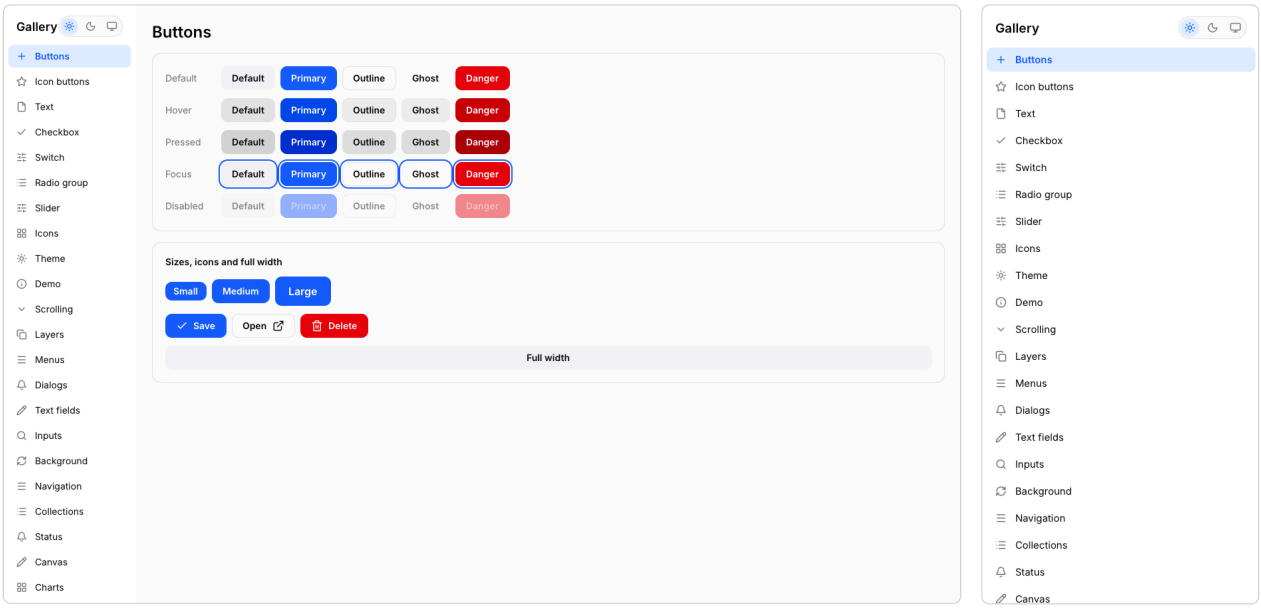


Figure 17: The gallery's Buttons page at 1440 × 900 and at 390 × 844. Below `MD`, the sidebar moves above the page; below `SM`, each state's five buttons stack too, because they don't fit a phone's width on one line.

11.5 Container queries

The window's breakpoint is the wrong question for a component that can sit anywhere: a card in a 240 pt sidebar and the same card across a wide page see the same window. Container queries ask the room the component itself has, as CSS's `@container` does. `Ui.width` and `Ui.height` return the size of the nearest open box that has an id, as it was laid out last frame. The `Ui` keeps a stack of the open boxes' ids, `box_ids`, beside `dirs`: opening a box pushes its id (or `NO_ID`), closing it pops, and the query walks the stack down from the top, skipping anonymous boxes. When no open box has an id, or the nearest one wasn't laid out last frame (its first frame), the answer is the window's size. `Ui.width_at(min, wide, narrow)` is `at()` for the container: `wide` when the container is at least `min` points wide, `narrow` below.

The size comes from last frame's rectangles, the ones `interact()` uses, because the current frame's layout runs only at `end()`. Like text wrapping, a container query lags one frame after a resize and settles on the next.

The gallery's `song_card` (in `examples/gallery/pages.c3`) lays itself out from its own width. Its card has an id, so `width_at` reads the card, not the window:

```
fn void song_card(Ui* ui, String id, Size width)
{
    ui.@card(id, width: width, pad: 4, gap: 2)
    {
        bool wide = ui.width_at(320, true, false);
        ui.@stack(wide ? Dir.ROW : Dir.COLUMN, width: ui::GROW, gap: 3,
                  align: wide ? Align.CENTER : Align.START)
        {
            ui.icon(MUSIC, wide ? 40 : 28, TEXT_SECONDARY);
            ui.@column(width: wide ? ui::GROW : ui::FIT, gap: 1)
            {
                ui.text("Clair de lune", weight: SEMIBOLD);
                ui.secondary(wide ? "Claude Debussy · Suite bergamasque" : "Debussy");
            };
        };
    };
}
```

The gallery draws it twice side by side, at `ui::px(180)` and at `GROW`: the narrow one stacks a small icon over a short line, the wide one sets a larger icon beside the full credit.

`Ui.@container(id, max, gap)` is the box to measure when a layout needs one: a column that takes the room it is given up to `max` points, centred in its parent, as Tailwind's `container mx-auto` does. It always has an id, from `id` or a counted `"container"` id when `id` is empty, so `ui.width()` inside reads its own width. `ui::PROSE` (640, about 65 characters of body text) bounds running text; `ui::CONTAINER_SM`, `ui::CONTAINER_MD`, `ui::CONTAINER_LG` and `ui::CONTAINER_XL` (640, 768, 1024 and 1280) follow Tailwind's container steps. The hello example wraps its cards in `ui.@container("content", max: ui::CONTAINER_XL, gap: 4)`, so a very wide window keeps them at a readable width, centred, and lays them out inside it with `ui.@grid("top cards", min_cell: 260, gap: 4)`: three side by side, then two, then one as the container narrows.

A grid's column count follows the same queries. `cols: ui.at(MD, 3, 1)` follows the window, `cols: ui.width_at(600, 3, 1)` the container around the grid, and `min_cell:` the grid's own width. Like a container query, the grid reads last frame's width, so its columns settle one frame after a resize.

The two kinds of query split the work. `bp()` and `at()` decide the window's layout, the shell: whether the sidebar shows, whether the main columns sit side by side. `width()` and `width_at()` belong to a component that may land in a narrow column or a wide one and should fit either without its caller passing a flag.

11.6 Scrolling

A scroll area is a box that clips its content and moves it by an offset. `Ui.@scroll(id, axis, overscroll, ...)` takes the same layout options as `@row` and `@column` after its two own ones (`width, height, dir, pad, pad_x, pad_y, gap, align, justify, bg, radius, border`), and `Ui.@scroll_ex(id, opts)` takes them in a `ScrollOpts`: the axis, the overscroll behavior and a `BoxOpts` for the area's box. The id is required, because the offset lives in the area's state (State and memory) and has to survive from frame to frame.

```
ui.@scroll("messages", gap: 2, pad: 3)
{
    foreach (i, m : messages) ui.@id(i) { message_row(ui, m); };
};
```

`ScrollAxis` says which way the area scrolls:

- `VERTICAL`, the default, scrolls up and down. Across, the content keeps the area's width, so text wraps to it as it would in a column.
- `HORIZONTAL` scrolls sideways, such as a strip of cards in a `dir: ROW` area.
- `BOTH` scrolls both ways, and the content keeps its natural width.

An area grows to fill its parent by default (`width` and `height` are `GROW`). With `FIT` along its axis it would always be as large as its content, and so never scroll; give it `GROW`, a fixed size or a percentage, or a `max` on `fit`.

11.6.1 On top of Clay

Clay has the pieces: a clipping element with a child offset. Kiln keeps the offset itself and only uses Clay to place and measure.

- **Clipping along the scrolling axes only.** An area declares Clay's clip on the axes it scrolls, and only those. Clay lays out the children of a clipping element at their natural size along a clipped axis instead of compressing them into the box, so the content can be longer than the view there; across, the content is still fitted, which is what makes text wrap in a vertical area.
- **The offset.** The area's state holds the offset in points, as a spring that eases toward a target (Motion), with 0 at the start. Each frame, `open_box` passes it to Clay as the element's `childOffset`, rounded to whole device pixels so text and edges stay sharp while the content moves.
- **Content and view sizes.** After the layout, `Ui.end` reads each area's content size (padding included) and view size back with Clay's `getScrollContainerData`. The range is the content less the view on each scrolling axis, never negative. The offset is then clamped to the range, so content that shrinks, such as a list losing rows, never leaves the area showing past its end.
- **Hit testing.** The area joins the stack of clipping boxes the `Ui` keeps while building, so each widget inside it records the visible part of its rectangle: a button scrolled out of view can't be clicked through the area's edge.



Figure 18: A vertical scroll area. The content is laid out at its natural height and moved up by the offset; only the part inside the view shows. The range is the content's height less the view's, and the thumb's length and position follow the view and the offset as fractions of the content.

11.6.2 Overscroll

`Overscroll` is CSS's `overscroll-behavior`: what an area does with scrolling it can't use because it is already at its end. With `AUTO`, the default, the scrolling passes to the enclosing scroll area, so a list inside a scrolling page hands the wheel back to the page at its last row. With `CONTAIN`, the area keeps it and nothing outside moves, which suits a list in a dialog or a panel with its own scrolling. The choice is made once per gesture: a wheel gesture or a touch drag goes to the innermost area under the pointer that can still move that way (or that contains its overscroll) and stays with it until the gesture ends, so a page doesn't start moving halfway through a flick that reached the end of an inner list. Input and focus describes the wheel, touch and pen drags, the glide after a flick, and how keyboard focus brings a widget into view.

11.6.3 The scrollbar

Each axis whose content overflows gets a thin thumb along the area's trailing edge, over the content: 6 pt thick, widening to 10 pt under the pointer, in the `TEXT` color made translucent. Its length is the view's share of the content, with a minimum so it stays easy to grab. The bar fades in while the area scrolls or while the pointer is over it, and fades out about a second after the area last moved. With `ui.always_show_scrollbars` set, the bar stays as long as the content overflows. That flag follows the operating system's own scrollbar setting unless the app sets it with `Ui.set_always_show_scrollbars` (The operating system).

Dragging the thumb moves the content with it; a press on the track beside the thumb pages by one view toward the press. The bar's hit area is the full 10 pt track, so a thin thumb is still easy to catch.

The thumb is drawn at the end of the area's own clip, when the render commands close its scissor. So it sits over the content, which was drawn inside that clip before it, and inside every enclosing clip, so a bar of an area scrolled partly out of a page is cut by the page's edge like the rest of the area.

11.6.4 Reading where an area stands

`Ui.scroll_info(id)` returns a `ScrollInfo` from the last frame: the `offset` and the `range` on both axes, in points, and `bars`, the scrollbars' opacity from 0 to 1. It takes the area's `Id`, which `Ui.make_id_quiet` gives for the same name in the same scope without counting it as used twice. An id that isn't a scroll area gives all zeros. It is enough to show something that depends on the position, such as a "back to top" button or a hint that there is more below:

```
ui.@scroll("log", gap: 1) { log_rows(ui); };
ScrollInfo s = ui.scroll_info(ui.make_id_quiet("log"));
if (s.offset.y < s.range.y) ui.secondary("More below");
```

11.7 Conventions

- **No margins.** Spacing comes from the parent's padding and gaps; a box never pushes its siblings away.
- **Logical directions.** Options say start and end, never left and right: `align: START`, `justify: END`, a button's `icon_end`.

12 Layers

A layer is a box that floats above the page: a menu under its button, a popover beside an icon, a hint above a switch, a dialog in the middle of the window. It is declared where the code that opens it runs, inside whatever row or card holds its button, and Kiln lifts it out of that place: it draws above everything declared before it, isn't cut by the boxes around it, goes where it fits in the window, and takes the pointer before the page does. This chapter describes how.

12.1 Declaring a layer

`Ui.@layer(id, open, ...)` runs its body in a layer while `*open` is true. The flag belongs to the app, as a checkbox's value does (Return values and binding): the app sets it to open the layer, and the layer clears it when the user dismisses it. A menu is a button that flips the flag, followed by the layer anchored to that button:

```
if (ui.button("Options", icon_end: CHEVRON_DOWN)) g.menu = !g.menu;
ui.@menu("menu", &g.menu, anchor: ui.last_id())
{
    ui.menu_item("Rename", icon: PENCIL);
    ui.menu_item("Duplicate", icon: COPY);
    ui.menu_separator();
    ui.menu_item("Delete", icon: TRASH_2);
};
```

`Ui.@menu` is a layer with a menu's rows and keys, which clears the flag when a row is chosen (Menus, popovers and tooltips). `Ui.last_id` is the id of the widget that last called `Ui.interact`, here the button just built, so the menu is anchored to it without the caller naming the button's id. Without an anchor, a layer sits in the middle of the window, which is where a dialog goes. `modal: true` makes it block and dim the page:

```
if (ui.button("Delete the playlist", DANGER, icon: TRASH_2)) g.dialog = true;
...
ui.@layer("dialog", &g.dialog, modal: true, width: ui::px(380), pad: 6, gap: 3)
{
    ui.heading("Delete the playlist?", 3);
    ui.@row(width: ui::GROW, gap: 2, justify: END)
    {
        if (ui.button("Cancel")) g.dialog = false;
        if (ui.button("Delete", DANGER)) g.dialog = false;
    };
};
```

`Ui.@dialog` builds this shape for the app: a modal layer with a title, the app's body and a button row in the platform's order, which also reports how it was answered (Dialogs, sheets and toasts). The gallery's Layers page opens its dialog with it.

`@layer` takes its options as named parameters: where it goes (`anchor`, `side`, `align`, `distance`), how it behaves (`modal`, `dismiss`, `focus`, `backdrop`), and the layout of its box (`width`, `height`, `dir`, `pad`, `gap`, `bg`, `radius`, `shadow`, `border`). By default the box is a raised panel: `SURFACE_RAISED`, panel corners, a hairline border and a large shadow. `Ui.@layer_ex(id, open, opts)` takes the same options in a `LayerOpts`, whose `box` is a full `BoxOpts`, for the fields `@layer` leaves out. The zero `LayerOpts` is a non-modal layer in the middle of the window that takes focus and closes on a press outside.

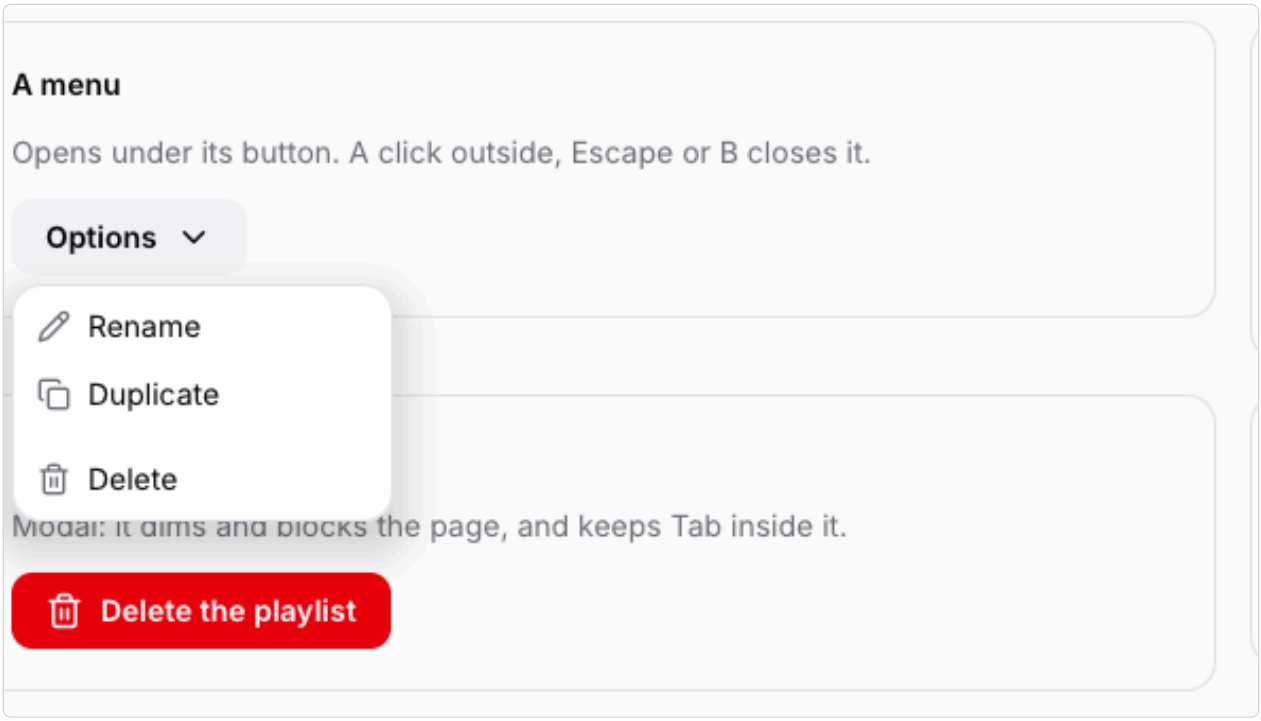


Figure 19: The gallery's Layers page with the menu open. The menu is declared inside the first card, after its button, yet it draws over the card below and isn't cut by the card it was declared in.

A layer's body runs in an id scope named after the layer, as inside `Ui.@id`, so its widgets never share an id with the page's: a toolbar's "Save" button and the "Save" row of a menu are two widgets with their own state, focus and input.

12.2 Z-order

Each layer's box is a Clay floating element attached to the root, not to its parent. Clay lays out a floating element separately from the flow it is declared in, so a layer takes no room in the row that holds its button, and Clay sorts the render commands of floating roots by their z-index after the page's.

Kiln gives the layers z-indices in the order they are declared in the frame: the first layer gets 2, the next 4, and so on, so a later layer draws above an earlier one. A modal layer's backdrop is a second floating root, one below its layer: it covers the page and every layer declared before the modal, but not the modal itself. A layer declared inside another layer's body, such as a color menu in a dialog, comes later in the frame, so it stacks above the layer it was opened from.

`Ui.current_layer` is the layer being built, or `ui::NO_ID` on the page.

12.2.1 Never clipped where it is declared

While building, the `Ui` keeps a stack of the clipping boxes around the code that runs, for hit testing (Scrolling). A layer pushes "no clip" onto that stack for its body and pops it at the end. Clay already draws a floating root outside its parent's scissor, so the two agree: a menu declared in a scroll area or a card with `clip: true` is neither drawn cut nor hit-tested as cut by that box. A layer whose own box clips still has its shadow: a clipping box draws its shadow when its clip starts, just before it, since Clay begins the clip before the background and the clip would cut a shadow drawn with it.

12.3 Placement

`ui::place_layer(anchor, size, side, align, distance, bounds)` decides where an anchored layer goes, and returns a `Placement`: the position and the side it took.

1. **Main axis.** The layer goes on `side` of the anchor (`BELOW` by default; `Side` also has `ABOVE`, `START`, `END` and `OVER`, described below), `distance` points away from it. If it doesn't fit there, it flips to the other side when that side fits, or when that side has more room even though neither fits.
2. **Cross axis.** `align` lines it up with the anchor's start, middle or end (`Align`): a menu under a button at the end of a row usually wants `align: END`, so it grows back toward the row.
3. **Shift.** The position is then clamped on both axes to stay inside the bounds. A layer larger than the bounds starts at their start.

`OVER` places the layer on its anchor instead of beside it, and never flips: the layer's top sits `distance` points above the anchor's top, `align` lines it up along the anchor as above, and the shift keeps it inside the bounds. A select uses it to open its list with the chosen row over the button, so the choice stays where the user's eye already is (Inputs). `Placement` reports `BELOW` as the side taken, so everything that follows the side taken treats an `OVER` layer as one below its anchor: it slides down into place from above, and an arrow goes on its top edge.

The bounds are the window, less the platform's safe area (`Platform`'s `insets()`: cutouts, gesture bars, an on-screen keyboard), less `ui::LAYER_MARGIN` on every side. A layer without an anchor is centered in the same bounds, unless it sets `at_edge`.

Placing needs the layer's size, which Clay only knows after the layout. So the first frame a layer is open, it is laid out unseen and takes no input: Kiln reads its size after the layout and requests another frame, and from the next frame on it is placed with the size from the frame before. `Ui.layer_live` says whether a layer takes input this frame: open, and past that first unseen frame. Its anchor's rectangle also comes from the last layout, as every widget's does (The frame), so a layer follows a moving anchor one frame behind.

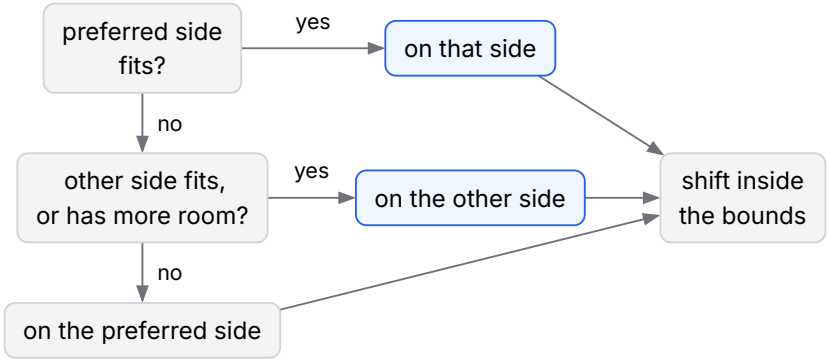


Figure 20: How `ui::place_layer` chooses the side, then keeps the layer inside the bounds. The gallery's popover that prefers `END` beside an icon at the window's right edge has no room there and more before the icon, so it opens on `START`.

12.3.1 Next to a point

A layer with `at_point` set in its `LayerOpts` goes next to `point`, a position in the window, instead of next to a widget: a context menu opens where the user asked for it. `ui::place_layer` places it against a rect of no size at that point, on `side` of it and `distance` away, so it flips and shifts at the window's edges as any anchored layer does.

12.3.2 Against a window edge

A layer with `at_edge` set in its `LayerOpts` and no anchor goes against an edge of the bounds instead: `side` names the edge (`BELOW` is the bottom, `ABOVE` the top, `START` and `END` the sides), `align` places it along that edge, and it sits `distance` plus `edge_offset` points in from it. It slides in from that edge as it appears. A toast is such a layer (Dialogs, sheets and toasts): `Ui.toasts` places each one at the bottom or top edge, and gives each older toast an `edge_offset` of the height of the newer ones plus a gap, so they stack inward from the edge, the newest nearest it.

12.3.3 An arrow toward the anchor

A layer with `arrow` set in its `LayerOpts` points at its anchor with a small triangle on the edge facing it, as a popover with `arrow: true` does (Menus, popovers and tooltips). The triangle is `ui::LAYER_ARROW_BASE` wide along that edge and `ui::LAYER_ARROW_HEIGHT` tall, in the layer's background color, outlined in the border color when the layer has a border; its base reaches 1 pt under the layer's edge, so it covers the border where they meet. It goes on the side the layer took after flipping, over the middle of the anchor, and slides along the edge only as far as the layer's rounded corners allow. Kiln draws it after the layer's box, as a small floating box of its own at the layer's z-index, with `DrawList.triangle` (Drawing calls). A layer without an anchor or without a background has no arrow.

12.4 Hit testing by layer

Without layers, the last widget built under the pointer takes it (The core calls). That rule alone fails for floating boxes: a menu declared in the first card is built before the second card's widgets, so the button it covers would win. Kiln therefore decides in two steps: first the layer, then the widget within it.

- **Each rectangle records its layer.** When the layout keeps an element's rectangle, it keeps the layer it was declared in too, or `ui::NO_ID` for the page.
- **Each pointer finds its layer.** At the start of a frame, every pointer looks through last frame's layers from the top down, skipping those that aren't live (being measured, or closing). The first modal layer takes it wherever it is; otherwise the first layer whose rectangle contains it does; otherwise it is on the page.
- **Widgets test only their own layer.** `Ui.interact` only lets a widget take a pointer that is over the widget's layer. The layer's box and a modal's backdrop call `interact` too, so a press on an empty part of a menu or on a dimmed page lands on the layer and reaches nothing beneath it.

A layer with `passthrough` set in its `LayerOpts` takes no pointer at all: pointers skip it when they look for their layer, and its box doesn't call `interact`, so the pointer reaches whatever is beneath it. A tooltip is such a layer (Menus, popovers and tooltips): a pointer that moves onto it still hovers and presses the widgets it covers.

Build order still decides between overlapping widgets of the same layer, but no longer between a page and a layer. The same test applies to hovering scroll areas, to the wheel and to touch drags: the wheel over a dialog doesn't scroll the page behind it, and a drag that starts in a menu moves only scroll areas in that menu.

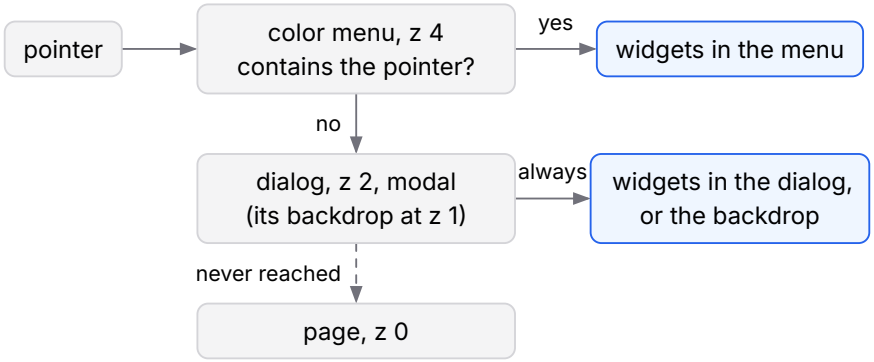


Figure 21: The layers a pointer is tested against, from the top down, with a color menu open from a modal dialog. A pointer over the menu goes to it; anywhere else, the modal dialog takes it, backdrop included, so nothing on the page sees it.

12.5 Dismissing

12.5.1 A press outside

A press closes every live, dismissible layer above the layer the pointer is over, and then does nothing else: Kiln drops the widget the press would have gone to, so the press only closes. A click outside an open menu closes the menu without also pressing the button underneath it, as native menus do on every platform. A press inside a menu that has a submenu open closes only the submenu.

A secondary press closes dismissible layers as a primary press outside them does, yet unlike it still reaches what is under it: it asks for that widget's context menu (Input and focus), so a right-click outside an open menu closes that menu and opens the menu of what it lands on.

`Dismiss` says which layers a press outside closes:

Value	A press outside
`AUTO`	closes a non-modal layer; a modal one stays (the default)
`OUTSIDE`	closes it, a modal layer too when the press lands on its backdrop
`NEVER`	doesn't close it; only the app, Escape or gamepad B do

12.5.2 Escape, gamepad B and the back button

A layer that takes focus opens a focus scope with its `open` flag (Focus), so it is the top layer for the keyboard and the gamepad while it is live. Tab and the arrows stay inside it, only its widgets read keys and gamepad buttons, focus moves into it when it appears, and Escape or gamepad B clear its flag, closing the top layer only. On Android, the raylib backend reports the system back button or gesture as Escape, so it closes the top layer the same way. When a layer closes, its scope goes, and focus returns to the widget that had it when the layer opened, usually the button that opened it.

A layer that should never take focus, such as a hint or a tooltip, passes `focus: false` (`no_focus` in `LayerOpts`): it opens no scope, so it doesn't trap Tab, Escape doesn't reach it, and the page keeps the keyboard. A hint usually pairs it with `dismiss: NEVER` and lets the app close it; a tooltip also passes the pointer through.

12.5.3 How it was dismissed

`Ui.dismissal(id)` says how the user dismissed the layer or focus scope `id`, as a `Dismissal`:

Value	The user
`ESCAPE`	pressed Escape or gamepad B
`OUTSIDE`	pressed outside it, or on a modal's backdrop that dismisses
`BACK`	used the system back gesture or button
`SWIPE`	dragged a sheet down past its threshold (Bottom sheets)
`NONE`	did none of these this frame

Each is reported once, on the first frame the app's code sees the layer closed. Escape and presses outside are handled at `end()`, after the app's code has run, so they are reported the next frame, when `*open` is first false; back and swipes are handled before the layer's body or inside it, and are reported in the same frame. A press outside that closes a menu and its submenu reports both. The app closing the layer itself, a menu row being chosen and a dialog's buttons aren't dismissals, and report nothing: the app already knows about them.

`keep_open: true` (in `LayerOpts` too) refuses the four: `*open` stays true, and `Ui.dismissal` still reports the attempt. It is for a layer holding work that would be lost, which asks before it goes:

```
ui.@dialog("rename", &renaming, "Rename the playlist", keep_open: edited)
{
    if (ui.text_field("Name", &name)) edited = true;
};
if (ui.dismissal("rename") != NONE) asking_discard = true;   // "Discard the new name?"
```

A refused press outside still does nothing else, and a refused back gesture still belongs to the layer, not the app: the layer stays where it is instead of following the gesture and springing back. With `dismiss: NEVER` a press outside isn't an attempt at all and reports nothing.

12.6 Presence and motion

Each layer keeps a spring in its state (Motion) from 0, hidden, to 1, shown. It eases in with `POPOVER` for a non-modal layer and `DIALOG` for a modal one, and out with `EXIT`, which is shorter and doesn't bounce. When `*open` turns false, the layer stays present while the spring falls: its body keeps running each frame, but the layer is not live, so it takes no pointer and opens no focus scope. Once the spring is near 0, the layer stops running its body.

So a layer's body must be able to run for a few frames after its `open` flag turned false. A dialog that shows an item the app has just deleted has to keep what it shows until the dialog is gone, or check for it; it must not assume that a running body means an open layer.

While the spring moves, two things follow it:

- **Fade.** The whole layer fades as one. Before painting each render command, Kiln sets the draw list's fade from the z-index of the floating root the command belongs to (The draw list), and the draw list scales the alpha of every shape and glyph by it. The page, whose z-index is 0, and backdrops, which have their own alpha, don't fade this way. Each shape fades on its own, so where shapes inside a layer overlap, the lower one shows faintly through while the layer is half faded.
- **Slide.** The layer moves `ui::LAYER_SLIDE` (6 pt) toward its anchor while hidden: a menu below its button comes down from it, one above it rises toward it. A layer without an anchor rises into place, and one against a window edge slides in from that edge.

With reduced motion on (Reduced motion), springs jump to their target, so a layer appears and disappears at once, without a fade or a slide.

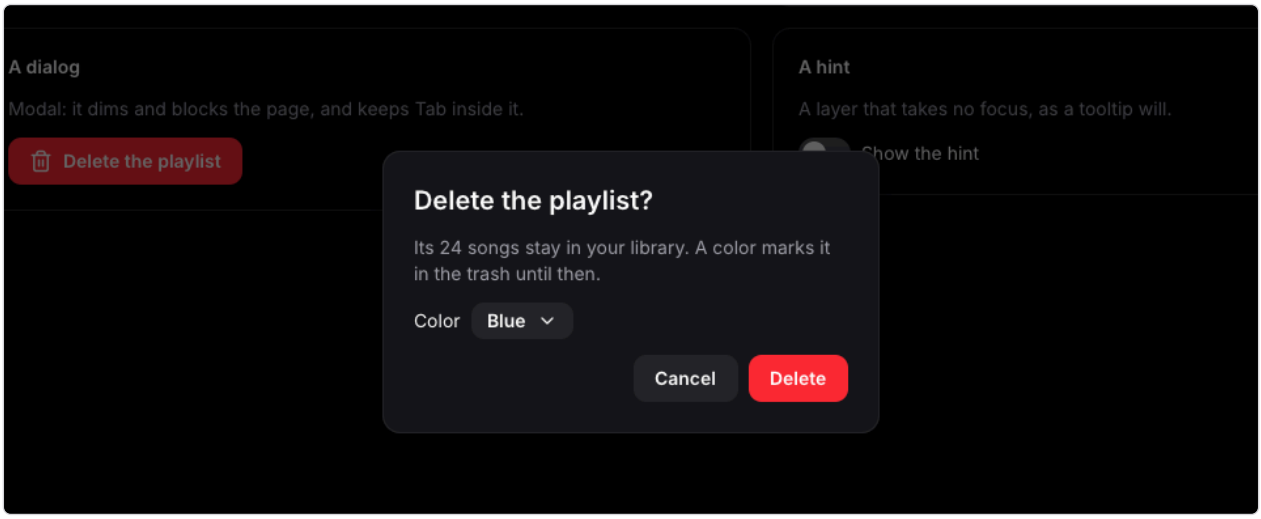


Figure 22: A modal dialog in the dark theme. The backdrop dims the page at 0.6; the dialog's color button opens a menu that stacks above the dialog.

12.7 The backdrop

A modal layer, unless it passes `backdrop: false` (`no_backdrop`), draws a backdrop: a floating root as large as the window, painted in the theme's `BACKDROP` color at `Theme.backdrop_opacity` times the layer's spring, so it fades in and out with the layer. The `LIGHT` preset dims with near-black zinc at 0.4, the `DARK` preset with black at 0.6, since a dark page needs more to recede (Presets).

The system theme (`themes::from_os`, The system theme) sets `backdrop_opacity` to 0 on macOS: macOS doesn't dim a window behind its sheets and alerts, while Windows, Linux desktops and Android dim behind their modal dialogs. A modal layer still blocks the page there; only the dim goes. A designer's own theme sets the opacity it wants on every platform, and overrides the system's choice.

A modal layer without a backdrop still takes the pointer everywhere, as a modal layer with one does.

12.8 Across platforms

	A press outside	Closes the top layer	Modal dim
Windows	closes, consumed	Escape	yes
macOS	closes, consumed	Escape	no (system theme)
Linux	closes, consumed	Escape	yes
Android	closes, consumed	back button or gesture	yes
Web	closes, consumed	Escape	yes

Gamepad B closes the top layer on every platform. The modal dim is the system theme's default; a designer's theme decides for itself. Layers stay `ui::LAYER_MARGIN` inside the window, less the safe area the platform reports through `insets()`; the raylib platform reports none.

13 Menus, popovers and tooltips

Menus, popovers and tooltips are layers (Layers) with a purpose: a menu is a column of rows to choose from, a popover is any content beside a widget, and a tooltip is a short line of text that names a widget. Each is a macro or a function over `Ui.@layer_ex`, so they float, flip and dismiss as every layer does. This chapter describes what each adds on top.

13.1 Two ways to write a menu

`Ui.@menu(id, open, anchor, side, align, width, distance, at_point, point)` runs its body as a menu in a layer while `*open` is true. It opens beside `anchor`, or, with `at_point: true`, beside `point` instead, as a context menu does (Context menus). As with any layer, the open flag belongs to the app: it opens the menu by setting the flag, and the menu clears it when the user chooses a row or dismisses it.

```
if (ui.button("File", icon_end: CHEVRON_DOWN)) g.file = !g.file;
ui.@menu("file", &g.file, anchor: ui.last_id())
{
    if (ui.menu_item("New", icon: PLUS, key: N, primary: true)) new_file();
    if (ui.menu_item("Open", icon: FOLDER, key: O, primary: true)) open_file();
};
```

Keeping the flag in the app is what a menu opened by anything other than its own button needs: a right-click on a list row, a shortcut that opens a menu, a menu bar where moving from one title to the next closes one menu and opens another. The app sets the flag from wherever it decides, and anchors the menu wherever it belongs.

For the common case, a button with a menu below it, `Ui.@menu_button(label, icon, variant, size, disabled, align)` and `Ui.@menu_icon_button(icon, label, ...)` keep the flag in Kiln's state instead, keyed by the button's id (The core calls). The button flips the flag when pressed and anchors the menu to itself; the app writes only the rows:

```
ui.@menu_button("Sort", icon: LIST)
{
    if (ui.menu_item("Name")) sort_by_name();
    if (ui.menu_item("Date")) sort_by_date();
};
ui.@menu_icon_button(ELLIPSIS_VERTICAL, "more")
{
    if (ui.menu_item("Rename", icon: PENCIL)) rename();
    if (ui.menu_item("Delete", icon: TRASH_2)) delete();
};
```

`@menu_button` aligns its menu with the button's start, and `@menu_icon_button` with its end, since an icon button with a menu usually sits at the end of a row. A menu is at least `ui::MENU_MIN_WIDTH` wide and grows to fit its widest row; it opens `distance` points from its anchor, 4 by default, on a raised panel with a border and a large shadow. With `side: OVER`, `distance` is how far the menu's top sits above the anchor's top instead (Placement), which is how a select lines its chosen row up with its button (Inputs).

13.2 Rows

`Ui.menu_item(label, icon, key, primary, shift, alt, disabled, checked)` is a row `ui::MENU_ROW_HEIGHT` tall. From start to end it holds:

- **A check mark column**, when `checked` points at a `bool`: a check mark while the value is true, and an empty space of the same width while it is false, so the labels of checkable rows line up. Choosing the row flips the value.
- **An icon**, when `icon` isn't `NONE`, in the secondary text color.
- **The label.**
- **The shortcut's label**, at the end, when `key` isn't `Key.NONE`: `Ui.shortcut_label` writes the chord the way the platform does, "Ctrl+Shift+S" on Windows and Linux, "⇧⌘S" on macOS, where `primary` is Ctrl or ⌘.

The shortcut label only shows the shortcut. A menu's body runs only while the menu is open, so the row can't be what handles the key: the app still calls `Ui.shortcut` with the same chord where the action lives (Shortcuts), and passes the same chord to the row so that the menu tells the user about it.

`menu_item` returns true once, in the frame the row is chosen. `disabled: true` draws the row in the disabled text color and makes it inert: it never returns true, the pointer doesn't highlight it, and the arrows skip it. `Ui.menu_separator` draws a hairline between groups of rows.

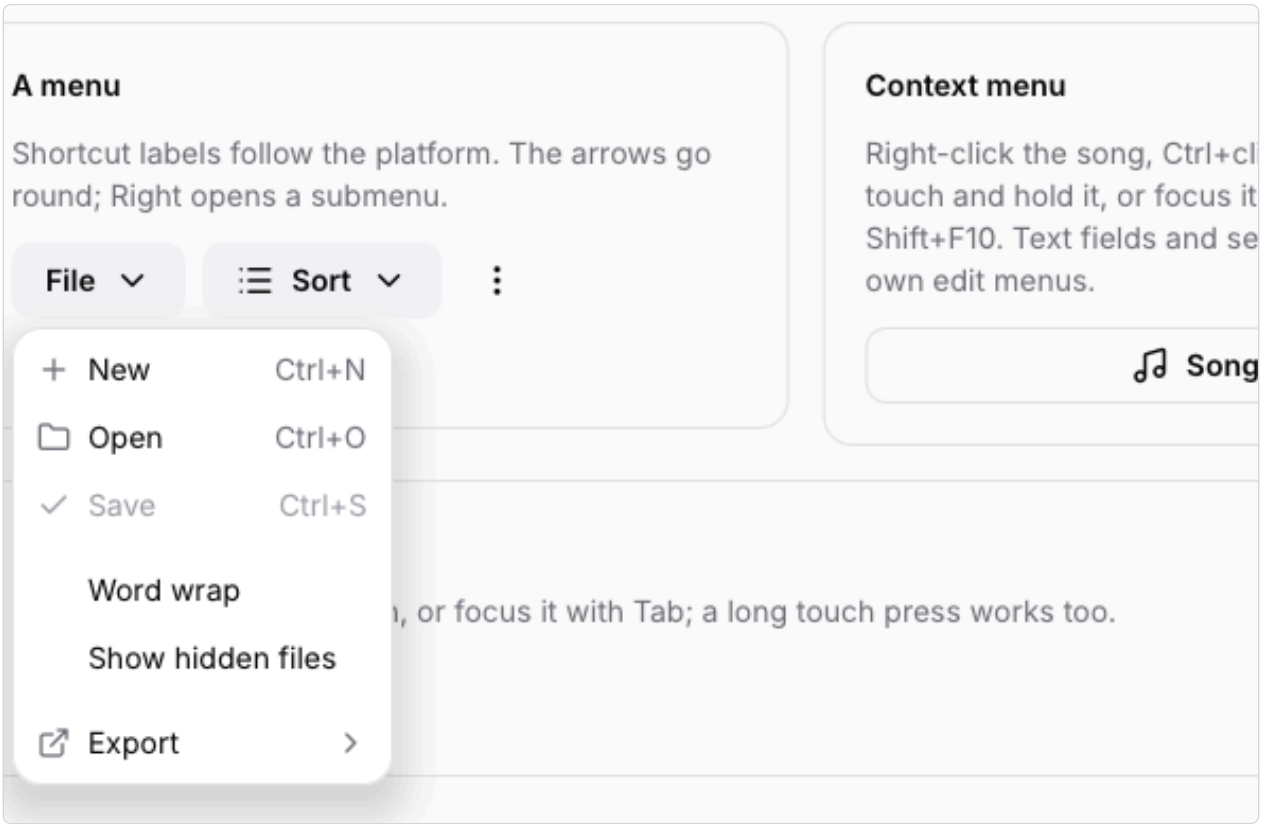


Figure 23: The gallery's File menu. Rows with icons and shortcut labels, a disabled Save, a separator, two checkable rows (both unchecked, so their check column is empty), and a row that opens a submenu.

13.2.1 Choosing closes the chain

While a menu's body runs, Kiln keeps the menu's open flag on a stack of the menus being built, with the menus it was opened from below it. When a row is chosen, it clears every flag on that stack, so choosing "SVG" in File › Export › More formats closes all three menus at once, as native menus do. A dismissal closes less: a press outside closes the menus above the layer it lands on, and Escape closes only the top menu.

13.3 Keyboard

A menu's rows are in a vertical `Ui.@focus_group` with `wrap: true` (Focus): Up and Down move between rows and go round from the last row to the first and back, skipping disabled rows, since they aren't focusable. Enter, Space or gamepad A choose the focused row. Escape or gamepad B close the top menu, as they close any layer that takes focus (Dismissing), and focus returns to the widget that opened it.

13.3.1 The highlight

A row is highlighted while the pointer is over it, or while it has focus and the focus is visible. The highlight is the menu's only cursor: rows draw no focus ring. When the pointer moves onto a row, the row also takes focus, so the arrows continue from where the pointer is rather than from a row the user left behind.

Opening a menu moves focus to its first row, but focus is only visible after keyboard use (The focus ring). So a menu opened with a click highlights nothing until the pointer reaches a row, while a menu opened with Enter or Space highlights its first row, ready for the arrows.

13.4 Submenus

`Ui.@submenu(label, icon, disabled)` is a row with a chevron at its end whose body is a menu of its own, anchored to the row on its `END` side. The submenu opens:

- when the pointer rests on the row for the submenu delay (Timing), so a pointer passing over the row on its way to another doesn't open it;
- at once with Right, Enter or Space while the row has focus, or with a click.

Left or Escape close only the submenu, and focus returns to its row in the parent menu. Resting on another row of the parent menu for the same delay closes it too, so the pointer can cut a corner of the parent menu on its way into the submenu without closing it. Submenus nest: a submenu may hold another `@submenu`.

A submenu is a layer like any other, so its placement follows Placement: it opens on the row's `END` side, and when the window has no room there, it flips to `START`. Since it is declared inside its parent's body, it stacks above its parent (Z-order).

13.5 Context menus

`Ui.@context_menu(id; body)` gives the widget built just before it a context menu. Like a tooltip, it needs no flag and no anchor: it attaches to `Ui.last_id`, keeps its open flag and its point in its own state under that widget's id, and its body is a menu's rows.

```
if (ui.button_ex("Song.mp3", { .variant = OUTLINE, .icon = MUSIC, .full = true }))
g.last_action = "Play";
ui.@context_menu("song menu")
{
    if (ui.menu_item("Rename", icon: PENCIL)) g.last_action = "Rename";
    if (ui.menu_item("Duplicate", icon: COPY)) g.last_action = "Duplicate";
    ui.menu_separator();
    if (ui.menu_item("Delete", icon: TRASH_2, key: DELETE)) g.last_action = "Delete";
};
```

The menu opens on any of the ways a platform asks for one:

- **A secondary press**, the right mouse button, opens it at the pointer.
- **Ctrl+click** with Mac keys opens it at the pointer, as on macOS; the click doesn't press the widget.

- **A long touch**, a touch held still for `ui::LONG_PRESS` (0.5 s), opens it at the finger, and the touch then counts as a drag, so lifting it doesn't click the widget.
- **The Menu key or Shift+F10**, while the widget has focus, opens it at the widget's bottom-left corner.

Each of these is a request that the widget hears in the next frame through `Ui.context_requested`, with `Ui.context_point` saying where (Input and focus). `@context_menu` opens on a request, and another request while the menu is open moves it to the new point.

The menu is `Ui.@menu` at a point: `at_point: true`, the request's `point`, and a `distance` of 2. The point is a rectangle of no size, so the menu flips and shifts to stay in the window as a menu at a widget does (Placement), and rows, separators, submenus, the keyboard and closing all work as they do in any menu. A press outside closes it, and so does a right-click elsewhere; that right-click still counts as a request, so it opens the context menu of whatever lies under the pointer, if that has one.

`@context_menu` also registers the widget with `Ui.has_context`. A long touch on a registered widget opens the menu rather than the widget's tooltip, so a widget can have both (Tooltips).

A list, long list or table declares one menu for every row instead: `Ui.@row_menu(id; int row)` after it, called once each frame. Its body gets the app's row, and the open state and point belong to the list, so it survives scrolling and sorting. See One menu for every row.

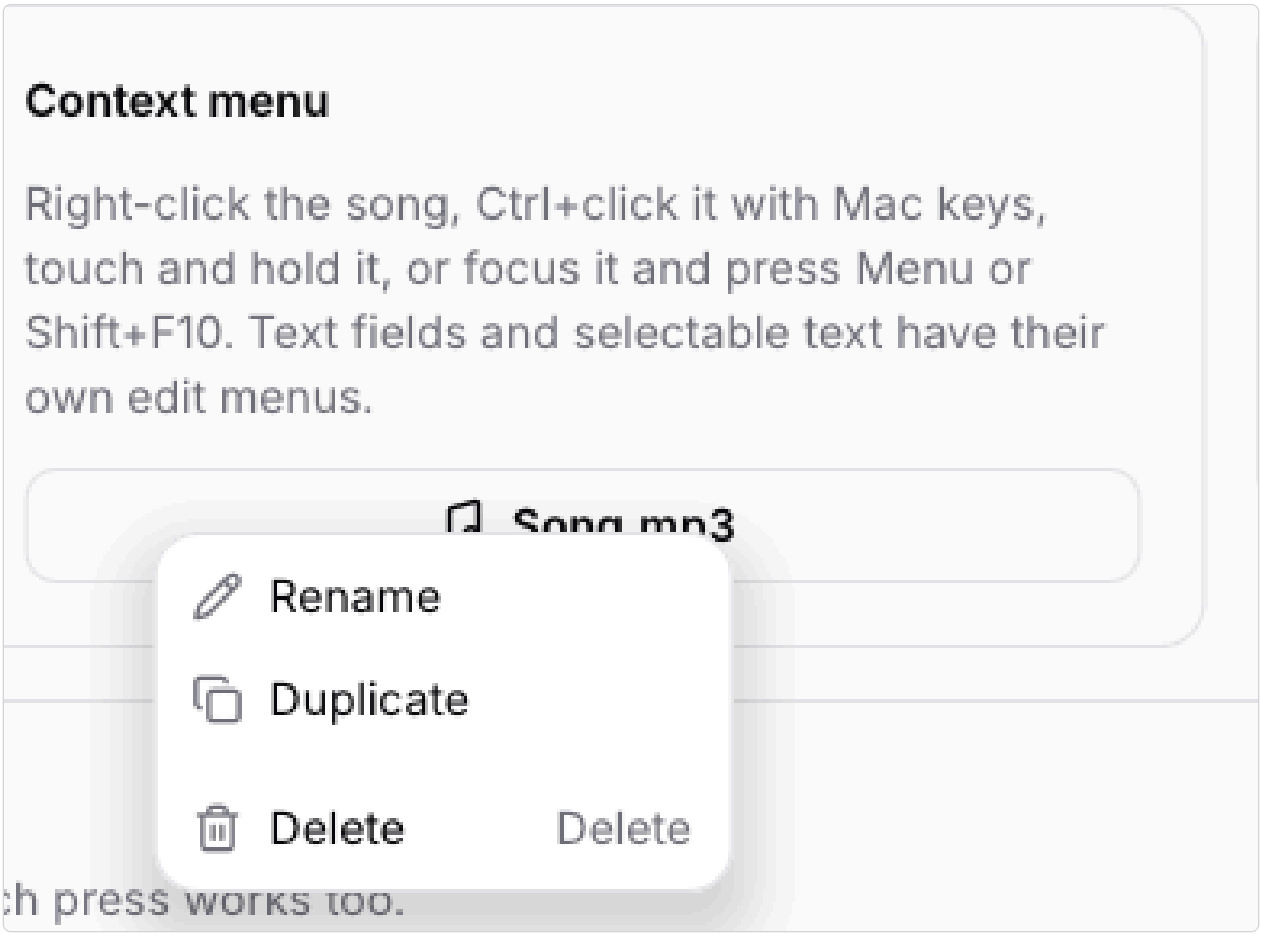


Figure 24: The gallery's context menu, opened with a right-click on the song. The menu opens at the pointer, over the button it belongs to.

Text fields and selectable text don't need `@context_menu`: they have built-in edit menus, opened the same ways (Text editing, Text).

13.6 Popovers

`Ui.@popover(id, open, anchor, side, align, arrow, width, height, pad, gap)` runs any content on a raised card beside `anchor`, while `*open` is true. Unlike a menu, it centers on its anchor by default, and holds whatever the app puts in it: a volume slider, a color picker, a short form. It dismisses as a layer does.

```
if (ui.button("With an arrow")) g.popover = !g.popover;
ui.@popover("volume", &g.popover, anchor: ui.last_id(), arrow: true, width: ui::px(220))
{
    ui.text("Volume", SM, SEMIBOLD);
    ui.slider("Volume", &g.volume, 0, 100);
};
```

13.6.1 The arrow

`arrow: true` (`LayerOpts`'s `arrow`) adds a small triangle, `ui::LAYER_ARROW_BASE` wide and `ui::LAYER_ARROW_HEIGHT` tall, on the edge of the layer that faces the anchor. The popover then opens a little further from its anchor, to make room for it.

The arrow is placed after the layer, from the side the layer took, not the side it asked for: a popover that prefers `BELOW` but flips `ABOVE` carries its arrow on its bottom edge, pointing down. Along that edge, the arrow sits over the middle of the anchor, clamped so it stays clear of the layer's rounded corners; when the layer has been shifted far from its anchor to stay in the window, the arrow stops at the corner nearest the anchor rather than leaving the layer.

The arrow is drawn after the layer's box, in the box's background color, with a `DrawList.triangle`. Its base overlaps the layer's edge by a point, covering the border there, so the arrow and the box read as one shape. When the layer has a border, a second triangle a hairline larger is drawn beneath the fill in the border color, so the outline continues around the arrow. A layer without an anchor or without a background draws no arrow.

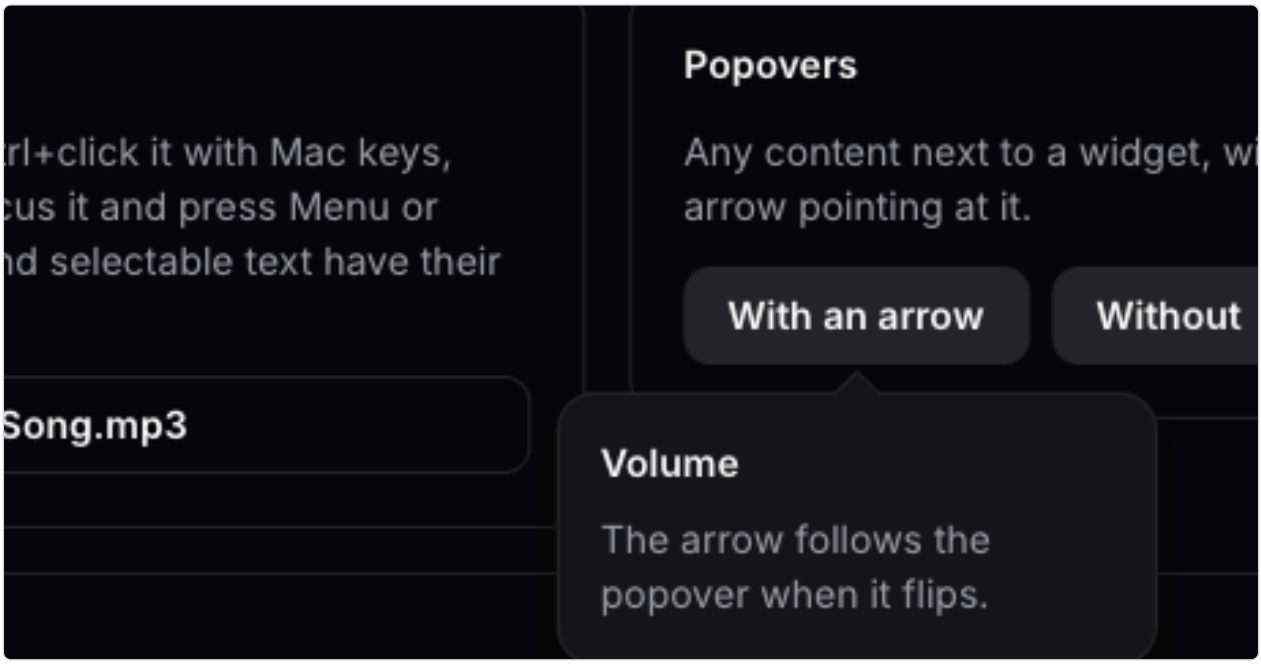


Figure 25: A popover with an arrow in the dark theme. The arrow points up at the middle of its button, and the layer's border runs around it.

13.7 Tooltips

`Ui.tooltip(text)` gives the widget built just before it a tooltip: the text on a dark chip above the widget, flipping below when there is no room. It needs no id and no flag: it attaches to `Ui.last_id`, the widget that last called `Ui.interact`, and keeps its own state under that widget's id.

```
ui.icon_button(DOWNLOAD, "download", OUTLINE);
ui.tooltip("Download for offline listening");
```

A widget wants its tooltip while the pointer is over it, or while it has visible keyboard focus. Once it has wanted it for the tooltip delay (Timing), the tooltip shows. On touch, where nothing hovers, a press held for the same delay shows the tooltip at once, and the press then counts as a drag, so releasing it doesn't click the widget. On a widget with a context menu, a long touch opens the menu instead of the tooltip (Context menus).

The tooltip hides when the widget stops wanting it, when the widget is pressed, and on Escape. After a press or Escape it stays hidden until the pointer or the focus leaves the widget, so a tooltip the user dismissed doesn't come back while they keep working with the same button.

A tooltip's layer is passthrough (`LayerOpts`'s `passthrough`): pointers never find it when they look for their layer (Hit testing by layer), so the widget beneath it keeps the pointer and the tooltip can't hold itself open. It also takes no focus and never dismisses itself on a press outside: the page keeps the keyboard, and Escape reaches the page.

`tooltip` returns true while the tooltip shows, for an app that wants to show something else at the same time.

13.8 Timing

Submenus and tooltips wait for the pointer to rest. Kiln reads two delays, in seconds, from the `Ui`: `tooltip_delay` and `submenu_delay`.

Waiting needs a frame when the delay runs out, even if nothing moves. An App-style loop (App style and Game style) sleeps until input arrives when nothing animates (Idle). So a widget that waits calls `Ui.wake_after(seconds)`, and the `Ui` keeps the earliest deadline still to come (Pointers and layers). On a platform that can sleep until a deadline (SDL3), the loop sleeps until the delay runs out and draws the frame that opens the tooltip or submenu. On one that can't (raylib), the `Ui` counts as animating until the deadline has passed, so the loop keeps running frames until then, and goes back to sleep after that frame.

13.9 Across platforms

The delays come from the system, as the theme does (Following the system), and fall back to `ui::DEFAULT_TOOLTIP_DELAY` (0.5 s) and `ui::DEFAULT_SUBMENU_DELAY` (0.2 s) where it has no setting.

	Tooltip delay	Submenu delay	Shortcut labels
Windows	the mouse hover time	the menu show delay	Ctrl+Shift+S
macOS	`NSInitialToolTipDelay`, when set; else 0.5 s	0.2 s	⇧⌘S
Linux	0.5 s (GNOME has no setting)	0.2 s	Ctrl+Shift+S
Android	0.5 s	0.2 s	Ctrl+Shift+S
Web	0.5 s	0.2 s	Ctrl+Shift+S

On Windows, the hover time and the menu show delay are the ones `SystemParametersInfo` reports for `SPI_GETMOUSEHOVERTIME` and `SPI_GETMENUSHOWDELAY`, the delays native tooltips and menus use. An app can pin either delay with `Ui.set_tooltip_delay` or `Ui.set_submenu_delay`, which the system then no longer changes, until `Ui.follow_os` with `OsSetting`'s `TOOLTIP_DELAY` or `SUBMENU_DELAY` hands it back (Overriding a setting).

14 Dialogs, sheets and toasts

A dialog asks a question and waits for the answer; a sheet offers a list of actions from the bottom edge of a phone; a toast says something and goes away on its own. Each is a layer (Layers): a dialog is a modal layer in the middle of the window, a sheet is a modal layer on the window's bottom edge, and each toast is a small layer against one of its edges. This chapter describes what each adds on top.

14.1 Dialogs

`Ui.@dialog(id, open, title, confirm, cancel, destructive, width, focus, enter_confirms)` runs its body in a modal layer while `*open` is true. The layer holds three things, top to bottom: `title` as a heading, the body, and a row of buttons at the end, `confirm` (`"OK"` by default) and `cancel` (`"Cancel"`). It is `width` points wide, 400 by default, on a raised panel with a border and a large shadow, and dims the page behind it as every modal layer does (The backdrop).

As with any layer, the open flag belongs to the app. The macro returns a `DialogResult`, which says how the dialog was answered in this frame:

Value	In this frame
`NONE`	the dialog wasn't answered: it is still open, or it was already closed
`CONFIRMED`	the confirming button was pressed, or Enter confirmed
`CANCELLED`	the cancelling button was pressed, or the dialog closed some other way

A dialog closes itself when it is answered: in the frame it returns `CONFIRMED` or `CANCELLED`, it has set `*open` to false. The app acts on the answer and has nothing to clean up:

```
if (ui.button("Delete", DANGER)) g.remove = true;
DialogResult r = ui.@dialog("remove", &g.remove, "Delete the playlist?", confirm:
"Delete", destructive: true)
{
    ui.text("This can't be undone. Its songs stay in your library.", SM, color:
TEXT_SECONDARY, wrap: true);
};
if (r == CONFIRMED) delete_playlist();
```

14.1.1 Every way out is an answer

A dialog can also close without a button: Escape or gamepad B close it as they close any layer that takes focus (Dismissing), and the app can clear the flag itself. The dialog notices: it keeps whether it was open in the last frame, and when it was open then and isn't now, without a button having answered, it returns `CANCELLED`, once. An app that only checks for `CONFIRMED` and `CANCELLED` therefore sees every dialog end exactly once, however it ended.

A dialog holding unsaved work passes `keep_open: true`: Escape, B, back and presses outside then leave it open, and `Ui.dismissal` reports the attempt, so the app can ask whether to discard the work (How it was dismissed). The dialog returns `NONE` for a refused attempt, and its buttons still answer it.

A press on the backdrop doesn't close a dialog: a modal layer stays open on a press outside unless it asks for `Dismiss`'s `OUTSIDE`, and the dialog keeps the default. A dialog that the app builds itself from `Ui.@layer_ex`, `Ui.dialog_buttons` and `Ui.dialog_answer` may allow it, and a press on its backdrop then reports `CANCELLED` like Escape.

14.1.2 A single button

`cancel: ""` leaves out the cancelling button, for an alert that only informs:

```
ui.@dialog("about", &g.about, "Kiln gallery", cancel: "")
{
    ui.text("Every widget in every state.", SM, color: TEXT_SECONDARY, wrap: true);
};
```

Its one button confirms; Escape and B still close it, and report `CANCELLED`.

14.2 Keys and focus

Enter confirms a dialog, unless a widget in the dialog used it first: Enter on a focused button presses that button, and Enter on a focused switch flips the switch. After the body and the buttons have run, the dialog checks whether Enter was pressed and is still unused (`Ui.key_pressed`), and only then answers `CONFIRMED`. Enter in a single-line text field confirms too: the field reports it as submitted and the dialog confirms on it, so “Name the playlist” takes a name and Enter. A dialog whose field means something else by Enter, such as a search inside a picker, passes `enter_confirms: false`. Escape and gamepad B cancel.

When the dialog opens, focus moves to the widget named by `focus`, by its label, such as a field the user types into first. Without one, focus the app put inside the dialog itself stays where it is; otherwise it moves to the default button:

- the confirming button, in an ordinary dialog, so Enter or Space answer the common case;
- the cancelling button, in a `destructive` dialog, whose confirming button is also drawn in the `DANGER` variant.

A destructive dialog also never confirms on an unused Enter. Together these mean that no Enter, whether pressed on purpose or left over from typing, confirms a risky action: the user has to move to the confirming button, or click it.

Focus moves only once the layer is live (`Ui.layer_live`). A layer’s first frame is laid out unseen, to measure it (Placement), and its focus scope notes the widget that had focus, the opener, only when it goes live. Moving focus in that first frame would make the dialog’s own button look like the opener. Waiting a frame keeps the opener, so when the dialog closes, focus returns to the button that opened it, as it does from every layer that takes focus. For the same reason an app names the field to start on with `focus` rather than calling `Ui.focus` the frame it opens the dialog, which would make the field the opener.

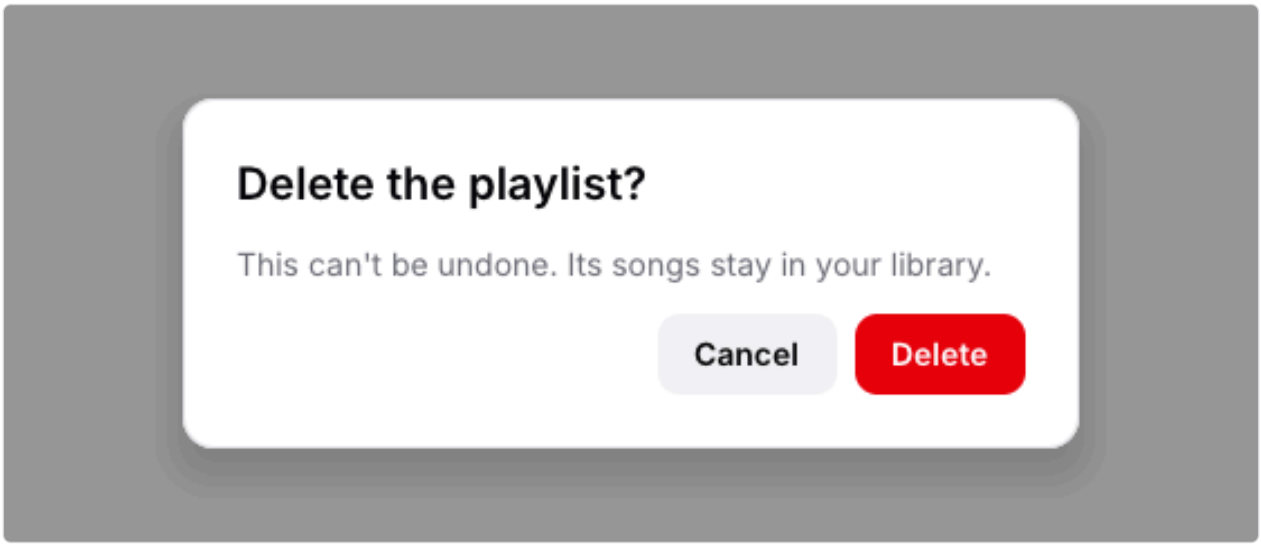


Figure 26: A destructive dialog in the light theme, with Cancel before Delete, the order everywhere but Windows. Delete is drawn in the `DANGER` variant, and focus starts on Cancel; focus is visible only after keyboard use, so the screenshot shows no ring.

14.3 Button order

Platforms disagree about where the confirming button goes. Windows and KDE put it first, "OK" then "Cancel"; macOS, GNOME and most other Linux desktops, Android and the web put it last, nearest the end of the row. Both orders right-align the row, so only the order of the buttons changes.

Kiln follows the platform. `Ui.dialog_confirm_first` is true on Windows, and on Linux when `XDG_CURRENT_DESKTOP` names KDE, and false elsewhere; `Ui.dialog_buttons` lays out the row in that order. The value comes from `ui::platform_confirm_first`: the order is a fixed convention of each platform or desktop, not a setting a user can change, so it is read from the build target and, on Linux, from the desktop the app runs in.

An app whose design wants one order everywhere pins it with `Ui.set_dialog_confirm_first(first)`. The system then no longer changes it, until `Ui.follow_os` with `OsSetting`'s `DIALOG_ORDER` hands it back (Overriding a setting).

14.4 Bottom sheets

`Ui.@sheet(id, open, wide, anchor, ...)` runs its body in a modal layer on the window's bottom edge while `*open` is true. On a phone it is as wide as the window, with a grab handle at its top, the page dimmed behind it, and the platform's bottom safe-area inset kept clear under its content. It is the shape native apps use for a song's actions, a device picker or an add-to-playlist list: the actions are within reach of the thumb, the page stays visible above them, and a downward drag puts them away. `Ui.@sheet_ex` takes the same options in a `SheetOpts`, for an app forwarding its own choices.

The body is ordinary widgets, laid out in a column. `Ui.sheet_item(label, icon, disabled, checked, tone)` is the usual row: a full-width action with an icon and a label, a check at its end when `checked`, and the `DANGER` tone for a destructive action. It returns true when it is chosen, by a click or by Enter, Space or gamepad A while focused.

```
if (ui.button("Song")) g.song = true;
ui.@sheet("song", &g.song)
{
    ui.heading("Midnight City", 3);
```

```
    ui.secondary("M83");
    if (ui.sheet_item("Add to queue", icon: LIST)) add_to_queue();
    if (ui.sheet_item("Add to playlist", icon: LIST_MUSIC)) open_playlists();
    if (ui.sheet_item("Cancel")) g.song = false;
};
```

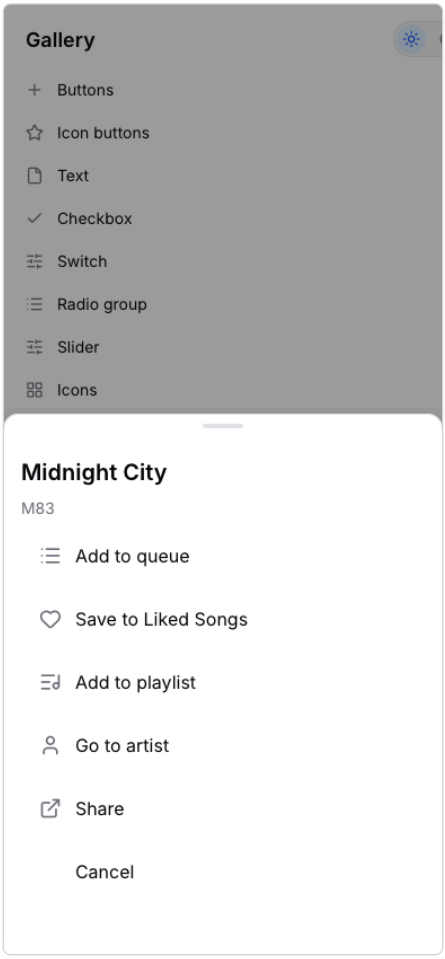


Figure 27: The gallery's song sheet on a phone at 390 × 844: the grab handle at its top, the list of actions, and the platform's bottom inset under the Cancel row. The page behind is dimmed and takes no input.

14.4.1 The handle and the scrim

The handle is a 22-point row at the top of the sheet with a 36 × 4 pill, which grows to a 44-point target under a finger. Dragging it down moves the sheet with the pointer; a drag shorter than about a third of the sheet's height springs back, and a longer one closes the sheet, a `SWIPE` dismissal (How it was dismissed). With `keep_open: true` the longer drag springs back too, and is reported. The handle is the only drag target, so a press on a row is the row's, and a list inside the body scrolls as it usually does.

A press on the scrim closes the sheet and does nothing else, as a press outside a menu does (Dismissing). Escape and gamepad B close it too, as they close any layer that takes focus. Choosing a row is the app's to handle: it sets `*open` to false, and the sheet slides away.

14.4.2 Focus

When the sheet opens, focus moves to its first action; Tab stays inside it, and when it closes focus returns to the widget that opened it, as for every layer that takes focus. The handle takes no focus: it is a pointer affordance, and Escape, B or a Cancel row are the keyboard's ways out.

14.4.3 The safe area

The sheet's background runs to the window's bottom edge, under a phone's gesture bar or home indicator, while a spacer of the platform's `Insets`'s `bottom` keeps the content above it. The inset is zero on the desktop and in the browser, and the height of the gesture bar on Android. A sheet is capped at the window less the top inset and `ui::LAYER_MARGIN`, so it never runs under the status bar or a cutout. A body longer than the cap is the app's to scroll, in a `Ui.@scroll` of its own.

14.4.4 Wide windows

Below `MD` the body is always a sheet. At `MD` and above it is a centred modal card with the page dimmed behind it, or with `wide: POPOVER` and an `anchor` a popover beside that widget; the default is the dialog. `SheetWide` names the two. Both use the same body, rows and actions, without the grab handle, so an app writes the list once and gets the shape its window calls for; the popover does not dim or block the page, as a popover never does (Menus, popovers and tooltips).

14.5 Toasts

A toast is a short message, such as "Saved" or "Playlist deleted · Undo", that appears at an edge of the window and leaves on its own. Two calls make them:

- `Ui.toast(text, action, id, seconds, notify_if_unfocused)` raises one, anywhere in the frame, usually where the event happens. It shows for `seconds`, 4 by default.
- `Ui.toasts(place)` draws the toasts raised so far, once a frame, after the page, so that they sit above it.

```
if (ui.button("Delete"))
{
    delete_playlist();
    ui.toast("Playlist deleted", action: "Undo", id: "undo", seconds: 6);
}
if (ui.toast_action("undo")) restore_playlist();
// ... the rest of the page ...
ui.toasts();
```

A toast has a dismiss button, an X, at its end. With an `action`, it also has a ghost button with that label before the X. Pressing the action closes the toast, and `Ui.toast_action(id)` returns true once afterwards, for the toast raised with the same `id`. `Ui.toast_count` says how many toasts are showing, not counting ones fading out.

A toast fits its text, up to `ui::TOAST_WIDTH` points wide; longer text wraps onto more lines.

14.5.1 One layer each

Each toast is a layer of its own, with `LayerOpts`'s `at_edge`: without an anchor, the layer sits against the window's edge on its `side`, at its `align` along that edge, `distance` plus `edge_offset` points in from it (Against a window edge). `Ui.toasts` draws the newest toast nearest the edge and stacks the older ones inward, each one's `edge_offset` the height of those before it plus 8 points.

Being separate layers is what lets toasts behave independently:

- Each one fades and slides on its own spring (Presence and motion). A layer at an edge slides in from that edge, `ui::LAYER_SLIDE` points, so a toast at the bottom rises into place and sinks as it leaves; once it has gone, the toasts behind it move in.
- Each takes the pointer only where it is. A layer catches the pointer only over its own box (Hit testing by layer), so a click beside a toast reaches the page, even between two toasts. A toast also never dismisses on a press outside (`Dismiss.NEVER`): the user working on the page doesn't sweep the toasts away.

- A toast takes no focus when it appears (`no_focus`): the keyboard stays where the user was typing.

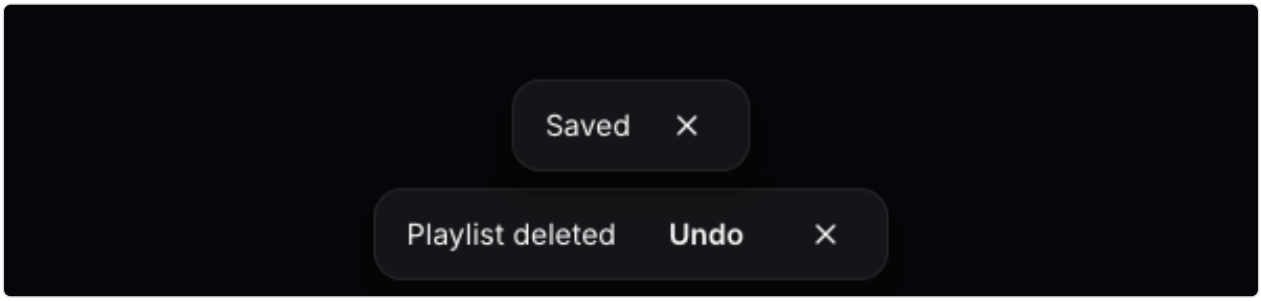


Figure 28: Two toasts at the bottom center in the dark theme. “Playlist deleted”, the newest, sits against the edge with its Undo action; the older “Saved” has moved up above it.

14.5.2 Time

A toast’s time counts down with each frame’s `dt`. It pauses while the pointer rests on the toast, so a user reading a long one or reaching for its action doesn’t lose it, and while focus is in it. A touch doesn’t pause it: a finger that lifts leaves no pointer resting anywhere. When the time runs out, the toast closes and fades.

While a toast counts down, it asks for a frame when its time ends with `Ui.wake_after` (Timing), so an App-style loop that sleeps while nothing moves still wakes to take it away. On a platform that can sleep until a deadline (SDL3), a toast draws no frames while it waits: the loop sleeps until its time ends, and the `dt` of the frame that wakes covers the sleep. On one that can’t (raylib), frames keep coming until it ends.

At most `ui::TOAST_LIMIT` toasts show at once. Raising one more sends the oldest away first, so a burst of events leaves the newest messages on screen.

14.5.3 No pointers kept

A toast raised in one frame shows in many, so it can’t keep the `String` the app passed: that may live in the temp allocator, as `string::tformat` results do, and be gone by the next frame. `Ui.toast` copies the text into a fixed pool inside the `Ui` instead: one slot per toast, `ui::TOAST_LIMIT` plus one so the oldest can fade while a new one appears, each with room for `ui::TOAST_TEXT` bytes of text and `ui::TOAST_ACTION` of action label. Longer text is cut at a character boundary, never inside a UTF-8 sequence.

So a toast holds no pointer into the app, and frames that raise no toast allocate nothing for them: the pool is part of the `Ui`, and its slots are reused.

14.5.4 From the keyboard

A toast that takes no focus still has to be reachable without a pointer. F6 moves focus to the newest toast, on its action or else its dismiss button, and shows the focus ring. F6 again, while focus is in a toast, sends it back to where it was. While focus is in a toast, the toast’s time pauses, so it stays while the user decides.

14.5.5 When the window isn’t in front

A toast is drawn inside the window, so nobody sees it while another app is in front. The platform reports that through `Input.unfocused` (`Input`); the raylib platform sets it from `IsWindowFocused`.

`notify_if_unfocused: true` asks for a toast that matters even then: raised while the window is unfocused, it also goes out as a system notification through `os::notify` (Notifications), with the toast’s text as its title. The toast still shows, for when the user comes back. The `hello` example raises its “Saved” toast this way.

14.5.6 Placement

`ToastPlace` says where `Ui.toasts` puts them: `BOTTOM_CENTER` by default, where GNOME, Android and Material on the web put theirs, or `BOTTOM_END`, `TOP_CENTER` and `TOP_END`. At the top, the newest toast sits against the top edge, and the others stack downward.

14.5.7 Toasts on Android

Android has a native toast, but Kiln draws its own there too. The native toast has no action button, so it can't offer "Undo", and it doesn't stack: a new one waits until the last one has gone. Its successor, the snackbar, belongs to Android's own view toolkit. Kiln's toasts look and behave the same as on the other platforms, at the same place Android puts its own.

14.6 Across platforms

	Button order	Modal dim	Unfocused toast also notifies	F6
Windows	confirm first	yes	a tray balloon, shown as a system toast	yes
macOS	confirm last	no (system theme)	`UNUserNotificationCenter`, in an app bundle	yes
Linux	confirm last; first on KDE	yes	`org.freedesktop.Notifications`	yes
Android	confirm last	yes	no: `os::notify` is unsupported, the toast alone shows	with a keyboard
Web	confirm last	yes	no: `os::notify` is unsupported, the toast alone shows	when the browser passes it on

The button order is the build target's, until the app pins it. The modal dim is the system theme's default (The backdrop); a designer's theme decides for itself. Toasts sit at the bottom center on every platform unless the app passes another `ToastPlace`.

15 Text editing

A text field looks simple and carries more than any other widget: a caret and a selection that move by characters a reader would recognize, undo that groups typing the way people expect, keys that differ between macOS and the other platforms, the mouse's double and triple clicks, the clipboard, an edit menu, and a caret that blinks at the user's rate. Kiln splits this in two. `TextEdit` is the editor: it changes the app's string and draws nothing. `Ui.text_field` puts one on screen and connects it to the pointer, the keys and the platform; `Ui.text_area` does the same over several wrapped lines (The text area).

15.1 The editor

A `TextEdit` edits a `DString` that the app owns. It keeps no copy of the text, only positions in it and the history of changes:

- `caret`, the byte offset where typing goes;
- `anchor`, the other end of the selection, equal to `caret` when nothing is selected. `TextEdit.selection` returns the two as a range, start first, and `TextEdit.selected` the selected text, a view into the string;
- the grapheme and word boundaries of the text, a `Breaks` (Graphemes and words);
- the undo steps.

Since the app owns the string, it may change it between frames: clear it after a search, fill it from a file. `TextEdit.sync` brings the editor up to date. It recomputes the boundaries if the text changed, moves a position past the end to the end, and moves a position that falls inside a grapheme to that grapheme's start. Every operation syncs first, so the editor never acts on stale offsets.

The operations are the ones a field needs:

Call	What it does
`TextEdit.move(text, m, extend)`	Moves the caret by an `EditMove`: a grapheme left or right, to the start of the word before, to the start of the next word (Windows, Linux), to the end of the word (macOS), or to the start or end of the text. With `extend`, the anchor stays, growing the selection. A plain left or right move from a selection lands on the selection's edge, as in every native field.
`TextEdit.place(text, pos, extend)`	Puts the caret at a byte offset, a click, or extends the selection there, a Shift+click or a drag.
`TextEdit.select_word(text, pos)`	Selects the word around `pos`, or the run of spaces or punctuation it is in.
`TextEdit.select_all(text)`	Selects the whole text.
`TextEdit.type(text, typed, time)`	Replaces the selection with typed text.
`TextEdit.replace_selection(text, with, time)`	Replaces the selection as one step of its own: a paste, or a cut with `with` empty.

Call	What it does
`TextEdit.delete_back(text, unit, time, line_start)`	Deletes the selection, or else an `EditUnit` before the caret: a grapheme, back to the start of a word, or back to `line_start`. A field leaves `line_start` at 0, the start of the text; a text area passes the start of the line on screen.
`TextEdit.delete_forward(text, unit, time)`	Deletes the selection, or else a grapheme or a word after the caret. `TO_START` here deletes to the end.
`TextEdit.undo`, `TextEdit.redo`	Undo or redo one step; false when there is none.

15.1.1 Undo steps

Each change is a step that records where it happened, the text it removed and the text it inserted, and the caret and anchor before it. Undo puts the removed text back and restores the caret and selection; redo applies the step again and puts the caret after the inserted text. A new change after an undo drops the steps that could have been redone.

Typing a character at a time must not make one step per character, so typing merges. `TextEdit.type` extends the last step instead of starting a new one when all of these hold:

- the last step was made by typing, and nothing has sealed it since;
- the user typed again within `ui::UNDO_PAUSE` seconds (1 s) of the last keystroke;
- the caret is where that step's text ends, and nothing is selected;
- the new text doesn't start a word: a space typed after a non-space ends the step, and so does a line break.

So "hello world", typed in one go, undoes in two steps: " world", then "hello". A move, a click, a selection, Enter in a field, or an undo seals the last step (`TextEdit.seal`), so typing after it starts a new one. Deletes, cuts and pastes are always steps of their own.

The editor keeps at most `ui::UNDO_LIMIT` steps (100) and drops the oldest past that.

15.1.2 Memory

The editor allocates in the allocator given to `TextEdit.init`: the boundary flags and the removed and inserted text of each step. The boundaries reuse their memory, and `Breaks.update` skips the work when the text's hash and length haven't changed, so a frame in which nothing is typed allocates nothing. `TextEdit.free` frees the history.

A text field or area keeps its editor in widget state (Widget state) with `Ui.@state_owning`, which takes a `StateDrop`: a function that frees what the state owns before the state itself is freed. The field's drop frees the editor's boundaries and undo history. State is collected when its widget hasn't been built for `ui::STATE_DATA_SECONDS`, so a field that leaves the screen for longer than that, on a page the user switched away from, loses its undo history; the text itself is the app's and stays. An app that wants the history kept holds the state with `Ui.keep_state`, with the field's id from `Ui.last_id` right after the field.

15.2 Graphemes and words

The caret steps by grapheme, not by byte or code point. A grapheme is what a reader sees as one character: a letter with combining accents, a flag (two regional indicators), an emoji built from

several pictographs and zero-width joiners such as , or a Devanagari, Thai or Khmer cluster of a consonant with its vowel signs. Moving by code point would put the caret inside these, where typing breaks them and Backspace removes half a family or half a flag. `TextEdit.sync` also moves a position that the app left inside a grapheme to its start, so the caret never sits between the parts of one.

`Breaks` supplies the boundaries, from kb_text_shape's implementation of Unicode segmentation, with one correction for emoji: kb_text_shape joins only the first pair of a chain of pictographs and joiners, so `Breaks.update` removes the boundaries after each later joiner. Word moves, word deletes and double-clicks use its word boundaries. Grapheme and word boundaries describes both, and what counts as part of a word.

15.3 The text field

`Ui.text_field(label, value, placeholder, icon, size, disabled, password, width, context_menu, read_only, kind)` edits `value`, the app's `DString`. If the string isn't initialized yet, the field initializes it with `mem`. The `label` names the field, and its id comes from it (The core calls). It is a wrapper around `Ui.field_ex` (Fields with more options), which takes all of a field's options and is what the search and number fields build on.

```
if (ui.text_field("Name", &demo.name, placeholder: "Your name", icon: USER).submitted)
{
    String who = demo.name.str_view().trim();
    ui.toast(who.len ? string::tformat("Hello, %s", who) : "Type a name first");
}
```

It returns a `TextFieldResult` for this frame:

Field	True when
`changed`	the text changed, by typing, deleting, pasting, cutting, undo or redo
`submitted`	Enter was pressed in the field
`focused`	the field has the keyboard focus
`left`	the focus left the field in this frame: the moment to check or reformat what the user typed
`cleared`	a search field's clear button or Escape emptied it (Inputs); a plain field never sets it

15.3.1 Look

The field is one line, a control's height for its `ControlSize` (`MD` by default), and `GROW` wide unless `width` says otherwise. It has a border, the theme's `BACKGROUND` behind the text (`SURFACE` when it is read-only, Read-only fields and areas), and the control corner radius. An `icon` sits before the text, in the secondary text color. The `placeholder` shows in the secondary text color while the text is empty and the field doesn't have focus; it goes when the field takes focus, so the caret starts on an empty line.

The selection is painted in the theme's `SELECTION` color (Tokens) behind the selected glyphs, and only while the field has focus. The caret is a 1.5-point bar in the text color, the line's height, hidden while something is selected.

A field shows its focus ring however it got focus. A button shows the ring only after keyboard navigation (The focus ring), since a mouse user knows what they clicked; a field marks where typing will go, which matters just as much after a click.

A `disabled` field takes no focus and no pointer: its text is drawn in `TEXT_DISABLED`, its icon faded, and the pointer keeps its arrow.

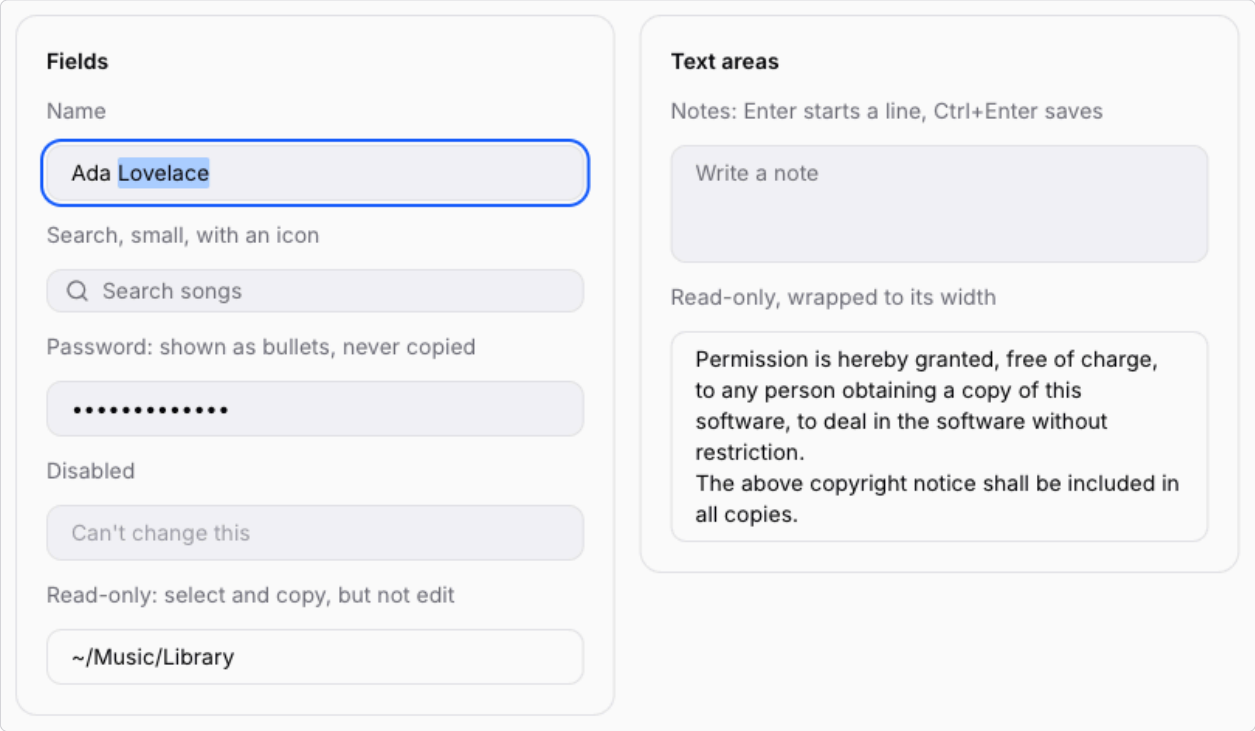


Figure 29: The gallery's text fields and areas in the light theme. The focused Name field shows its ring and a double-clicked word in the `SELECTION` color; below it a small field with an icon and a placeholder, a password field, a disabled one and a read-only one. On the right, an empty text area with its placeholder, and a read-only area whose text wraps at its width.

15.3.2 Passwords

A `password` field shows one bullet (•) per grapheme of the text, not per byte, so a password with an accented letter or an emoji shows as many bullets as the user typed characters. The field keeps a second `Breaks` for the bullets and maps each offset between the text and the bullets, so the caret, the selection and clicks land between the right bullets.

The text never leaves the field through it: copy and cut do nothing, the edit menu shows them disabled (The edit menu), and a selection in it never becomes the primary selection (The primary selection). A double-click selects the whole text, as a triple-click does, rather than a word whose length would show where the password has spaces. Paste works, for passwords that come from a password manager.

15.3.3 Scrolling sideways

Text longer than the field scrolls sideways. After every frame's edits, the field moves its scroll offset just enough to keep the caret in view, 2 points from the edge, and never past the end of the text, so deleting at the end pulls the text back in. The glyphs are clipped to the text area.

15.3.4 Where the caret goes

The caret's x comes from the shaped glyphs (Text), not from measuring substrings, because shaped text isn't a sum of its characters: a right-to-left run reverses the order of glyphs, and a ligature such as "fi" or a Devanagari conjunct draws several characters as one glyph.

The field shapes the text it shows and keeps its glyphs in visual order, each with its x and the byte range of text it draws. A glyph's text starts at its own byte offset and ends where the next glyph's text starts, in text order. For a caret at a byte offset, the field finds the glyph whose text holds it and

puts the caret at its leading edge: the left edge in a left-to-right run, the right edge in a right-to-left one. Inside a glyph that draws more than one grapheme, a ligature, the caret goes part of the way across, the glyph's advance split evenly among its graphemes. The end of the text is after the glyph that draws its last character, which is on the left in a right-to-left run.

A click goes the other way: the field takes the grapheme boundary whose caret x is nearest the pointer.

The field keeps these glyphs in a private `FieldGlyphs`, built for one line, and paints its selection as one span from the selection's start to its end.

Read-only text can be selected too. `Ui.text` with `selectable: true` reuses the editor's pieces without editing: `Breaks` for graphemes and words, `Ui.click_count` for double and triple clicks, the `SELECTION` color, and the clipboard for copy (Text). Its text can wrap, so it maps the pointer and paints the selection with `TextLines`, which knows each line's glyphs and their byte ranges. `Ui.selection_rects` returns the rectangles for a range: one or more per line, since in mixed-direction text the selected bytes can sit in several separate spans of the same line.

15.3.5 Fields with more options

`Ui.field_ex(label, value, opts)` is the whole field, with its options in a `FieldOpts`. `Ui.text_field` fills one from its parameters and calls it; a widget that needs a field with more behavior builds its own. The search field and the number field are built this way (Inputs).

Option	What it does
`placeholder`, `icon`, `size`, `disabled`, `password`, `read_only`	As for `Ui.text_field`.
`multiline`	Set by the text area (The text area): a paste keeps its line breaks (Edit actions).
`kind`	An `InputKind`: what the field holds, `TEXT` by default, or `NAME`, `EMAIL`, `USERNAME` or `NUMBER`, so the platform offers the right on-screen keyboard and input method (The platform and key routing). The number field passes `NUMBER`.
`no_context_menu`	No edit menu (The edit menu). `Ui.text_field`'s `context_menu: false` sets it; a zeroed `FieldOpts` has the menu.
`width`	The field's width. Pass `ui::GROW` for the usual field: a `FieldOpts` left zeroed has a zero `Size`, which is `FIT`.
`filter`, `filter_data`	A `CharFilter`, `fn bool(void* data, uint codepoint)`, called for each code point typed or pasted, with `filter_data`. A rejected code point is dropped and the rest goes in, so a number field can turn away letters while a paste of "12 kg" still brings in the digits. Null accepts every character but the control characters.

Option	What it does
`trailing`, `trailing_data`	A `FieldTrailing`, `fn void(Ui* ui, void* data)`, that builds widgets inside the field's box, after the text: the search field's clear button, the number field's steppers. It runs inside the field's row, after the text area, so what it builds sits within the border and the text area grows only into the space left.
`keys`	A `KeySet` of extra keys the field takes while it has focus. The field itself does nothing with them; the widget around it reads them after the field returns, and the set keeps them from shortcuts in the meantime (The platform and key routing).

A button that a trailing function builds belongs to the control, not to the focus order: `Ui.button_box` takes `focusable: false` for it. The pointer presses it like any button, and Tab goes from the field straight to the next widget, as it does past the clear button of a native search field.

15.4 The mouse

The pointer shows the I-beam text cursor (`Cursor.TEXT`) over a field that isn't disabled. A press focuses the field and acts by how many clicks the press makes in a row:

Press	Effect
Click	puts the caret at the nearest grapheme boundary
Shift+click	extends the selection from the anchor to the click
Drag	selects from where the press went down to the pointer
Double-click	selects the word, or run of spaces, under the pointer; the whole text in a password field
Triple-click	selects the whole text

`Ui.click_count(id)` counts the clicks. The pointer tracks count a press as another click of the same series when it comes within `Ui.double_click_time` of the last one and within 4 points of it, or 12 for a touch; the count goes up to 3 and then starts again. The double-click time is the user's, from the OS (Following the system).

A widget learns of a press in the frame after it happened, once layout has placed it, and the pointer may have moved by then. So a click places the caret where the pointer went down, `Ui.press_point_of`, and a drag that is already under way extends the selection to where the pointer is now.

A middle click is not one of these presses: on Linux and the BSDs it pastes the primary selection where it lands (The primary selection).

15.5 Keys

The field reads the keys itself: it passes `keys: false` to `Ui.interact`, so Enter and Space don't activate it as they would a button. Which keys do what depends on `TextKeys`, `ui.text_keys`: `MAC` on

macOS and `PC` elsewhere, or what the app pinned with `Ui.set_text_keys`. Shift with any of the moves extends the selection.

Action	PC	MAC
Grapheme left, right	Left, Right	Left, Right
Word left	Ctrl+Left: start of the word before	Option+Left: start of the word before
Word right	Ctrl+Right: start of the next word	Option+Right: end of the word
Start, end of the text	Home, End	Home, End, ⌘Left, ⌘Right
Delete back	Backspace; Ctrl+Backspace, to the word's start	Backspace; Option+Backspace, to the word's start; ⌘Backspace, to the start
Delete forward	Delete; Ctrl+Delete, to the word's end	Delete; Option+Delete, to the word's end
Select all	Ctrl+A	⌘A
Copy, cut, paste	Ctrl+C, Ctrl+X, Ctrl+V	⌘C, ⌘X, ⌘V
Undo	Ctrl+Z	⌘Z
Redo	Ctrl+Shift+Z, Ctrl+Y	⌘Shift+Z
Submit	Enter	Return

The command chords for select all, the clipboard, undo and redo don't act on the editor directly: each calls `Ui.field_do` with its `FieldAction`, as the edit menu's rows do (Edit actions), so a key and its menu row can't drift apart. The keys a field and a text area share, Left and Right, Backspace and Delete, the command chords and typed text, are read by one function, `Ui.field_edit_keys`; each widget reads its own Home, End, vertical keys and Enter.

Each key is taken with `Ui.key_pressed`, which counts key repeat, so holding an arrow or Backspace keeps moving or deleting, and which uses the key, so nothing later in the frame acts on it again.

Typed characters come from `Input.text`, not from key presses, so they arrive as the keyboard layout and input method made them. The field drops control characters, so typing brings in neither a line break nor a tab; a text area's line breaks come from Enter (The text area). While the command modifier (Ctrl, or ⌘ with Mac keys) is held, typed text is ignored, so a chord never also types its letter. Option on macOS still types, as it does in every Mac app, and so does Ctrl+Alt with PC keys: that is how Windows and Linux report AltGr, which types `@` or `€` on many layouts.

Three keys the field leaves alone:

- Enter sets `submitted` and seals the undo step. Because the field uses the key, a dialog around the field doesn't also confirm on it (Keys and focus).
- Tab still moves focus to the next widget (Focus), in a text area too: neither ever types a tab character, so the keyboard can always leave them.
- Escape goes to the layer the field is in, which closes as on any other Escape (Dismissing), unless the field took Escape among its extra `keys` (Fields with more options), as a search field with text does to clear it.

15.6 The text area

`Ui.text_area(label, value, placeholder, rows, max_rows, size, disabled, read_only, width, context_menu)` edits `value` over several lines. It is the field's sibling, not a variant of it: it keeps its

own `TextEdit` in widget state and lays its text out in lines, but shares the field's undo, clipboard, edit actions and menu, caret blink and focus ring, and returns the same `TextFieldResult`.

```
if (ui.text_area("Notes", &demo.notes, placeholder: "Write a note").submitted)
ui.toast("Note saved");
```

Enter starts a new line, so submitting takes Ctrl+Enter, ⌘Return with Mac keys, which sets `submitted`. Tab moves the focus, as in a field.

15.6.1 Lines and scrolling

The area wraps its text at its own width. A widget is built before layout places it, so the width is the one the area had in the last frame, from `Ui.rect_of` on its first frame it wraps at the window's width, and the next frame corrects it. The lines come from `Ui.text_lines` with `trailing_line`, which adds an empty line after a final line break, so a caret after it has a line of its own to sit on. The selection is painted with `Ui.selection_rects`, the same as selectable text (Selectable text).

The area is `rows` lines tall (3 by default) and grows with its text up to `max_rows` (8); past that it keeps its height and scrolls vertically inside a scroll area (Scrolling). After every move or edit it scrolls just enough to keep the caret's line in view. The scroll range comes from the last layout, which hasn't seen a line the user just added, so the area checks again in the next frame.

At a soft wrap, where a line breaks for width and not at a line break, one byte offset is both the end of one line and the start of the next. The caret's **affinity** decides which: End, and a click past a line's end, keep the caret at the end of that line; every other move puts it at the start of the next. `TextLines.line_of(at, upstream, text)` finds the caret's line with that choice; after a line break the caret is always on the next line.

15.6.2 Keys in an area

The area reads the same keys as the field (Keys), with these added or changed. Shift extends the selection with any move.

Action	PC	MAC
Line up, down	Up, Down	Up, Down
Page up, down	Page Up, Page Down	Page Up, Page Down
Start, end of the line on screen	Home, End	Home, End, ⌘Left, ⌘Right
Start, end of the text	Ctrl+Home, Ctrl+End	⌘Up, ⌘Down
Delete back	as in a field	as in a field; ⌘Backspace, to the start of the line on screen
New line	Enter	Return
Submit	Ctrl+Enter	⌘Return

Up and Down move by visual line, the lines as wrapped on screen, and aim for a **goal x**: the caret's x when the first of a run of vertical moves began, so the caret keeps its column across a short line instead of drifting left. Any other move, a click or an edit forgets it. Up on the first line goes to the start of the text, and Down on the last to its end. Page Up and Page Down move by the lines in view less one, keeping one line of context, and stop at the first or last line.

The mouse works as in a field (The mouse): a click, a Shift+click, a drag across lines, a double-click for a word and a triple-click for the whole text.

15.7 Read-only fields and areas

`read_only` on `Ui.text_field`, `Ui.text_area` or `FieldOpts` makes text that the user can select and copy but not change: a file path, a license, a generated key. Unlike a `disabled` field, a read-only one takes focus and has its place in the Tab order, and the pointer selects in it as usual.

- The caret is hidden, but the keyboard still selects: the arrows and Shift+arrows, Home and End, an area's vertical keys, select all and copy.
- Typing, Backspace, Delete, cut, paste, undo and redo do nothing, and Enter neither submits nor starts a line, with or without Ctrl or ⌘. `Ui.field_can` says no to every action but copy and select all (Edit actions).
- The edit menu has only Copy and Select All (The edit menu).
- It is filled with `SURFACE` instead of `BACKGROUND` and keeps its border, and its text stays in `TEXT`, not the disabled color: it is content to read, not a control that is off.
- It doesn't call the platform's `text_input`, so focusing it brings up no on-screen keyboard (The platform and key routing).

Selectable text, `Ui.text` with `selectable: true` (Selectable text), offers the same selecting and copying without a box or a stop in the focus order; a read-only field suits a value that sits among other fields of a form.

15.8 Edit actions

What a field's command chords and its edit menu do is one set of actions, a `FieldAction`: `UNDO`, `REDO`, `CUT`, `COPY`, `PASTE`, `DELETE` and `SELECT_ALL`. Two functions carry them out, and both the keys and the menu call them:

- `Ui.field_can(ed, value, action, opts)` tells whether the action would do anything now: undo needs a step to undo and redo a step to redo; cut, copy and delete need a selection, and cut and copy are never possible in a password field; select all needs text; paste needs a platform. In a read-only field or area, only copy and select all are ever possible.
- `Ui.field_do(ed, value, action, opts)` does it, and does nothing when `field_can` says no. An undo or redo goes to the editor (The editor); cut, copy and paste go through the system clipboard; delete removes the selection as one undo step, a cut that leaves the clipboard alone.

The clipboard belongs to the `Platform` (The interfaces): copy and cut call `clipboard_set` with the selected text, and paste calls `clipboard_get`, which returns a copy made in the temp allocator, and hands the text to `Ui.paste_text`. Since a field holds one line, a paste there turns each line break (`\n`, `\r` or `\r\n`) and tab into one space. A text area, whose `FieldOpts` has `multiline`, keeps them: each line break becomes `\n` and tabs stay. Either way, the paste then runs the field's `filter` (Fields with more options) over what is left, and replaces the selection with the result as one step. Without a platform, as in tests that run the `Ui` alone, copy and paste do nothing and cut only deletes the selection.

15.8.1 The primary selection

Linux and the BSDs have a second clipboard, the **primary selection**: whatever text the user last selected, in any app, which a middle click pastes. Selecting is enough to set it, and the middle button pastes it without touching the clipboard. `ui::HAS_PRIMARY_SELECTION` is true on those systems; elsewhere none of this happens. The platform keeps it, with `primary_get` and `primary_set` (The interfaces).

`Ui.offer_primary(id, text, range)` makes a selection the primary selection. A field, an area and a selectable text (Selectable text) call it at the end of every frame in which a non-empty selection stands and the pointer isn't dragging, so the selection is offered when a drag is released, after a double or triple click, after Shift and the arrows, and after Select All. It remembers the widget and

the range it offered last and offers the same selection only once, so the platform is asked once per selection made, not once per frame. A password field never offers its selection.

A middle click in a field or area that is neither disabled nor read-only, heard through `Ui.middle_clicked` (Pointers and layers), pastes the primary selection where it landed. `Ui.middle_paste` reads it from the platform and, when it isn't empty, focuses the widget, puts the caret at the click and inserts the text through `Ui.paste_text` with Paste's rules: `\r\n` is one break, line breaks become `\n` in an area and spaces in a one-line field, and the field's filter applies. Placing the caret first lets go of any selection the field held, so the text goes in where the click points rather than over the selection. An empty primary selection leaves the field as it was.

`Ui.middle_click_paste` turns the pasting on or off. It comes from the OS: on for Linux and the BSDs unless the user turned it off (The system theme), off elsewhere. `Ui.set_middle_click_paste(on)` pins it until `ui.follow_os(MIDDLE_CLICK_PASTE)` (Overriding a setting). Turning it off stops only the pasting: selecting still sets the primary selection, as GTK does, so other apps can still paste it.

15.9 The edit menu

A text field has the edit menu that native fields have, a context menu (Menus, popovers and tooltips) of the actions above. It is on by default; `Ui.text_field(..., context_menu: false)`, or `FieldOpts.no_context_menu`, turns it off, and a disabled field has none. The search and number fields, built on `Ui.field_ex`, have it too (Inputs), and so does a text area: both widgets build it with `Ui.edit_menu`.

The menu opens however a context menu opens (Input and focus): a secondary press on the field, Ctrl+click with Mac keys, a long touch, or the Menu key or Shift+F10 while the field has focus. A secondary press outside the selection first moves the caret to the press, so the menu acts where the user pointed; a press inside the selection keeps it, so the user can cut or copy what they selected. The field takes the focus, and gets it back when the menu closes, so typing goes on where it was.

The rows follow the platform's own field menu, by `TextKeys`:

Keys	Rows
`PC`	Undo, Redo — Cut, Copy, Paste, Delete — Select All
`MAC`	Cut, Copy, Paste — Select All

A read-only field or area has Copy — Select All with either keys, the rows of selectable text's menu (Read-only fields and areas).

Each row shows its chord (Ctrl+Z, ⌘C), Delete excepted, and is disabled when `Ui.field_can` says it can't act, re-checked every frame while the menu is open. Paste is the exception: whether the clipboard holds text is asked once, when the menu opens. A chosen row calls `Ui.field_do`, as its chord would.

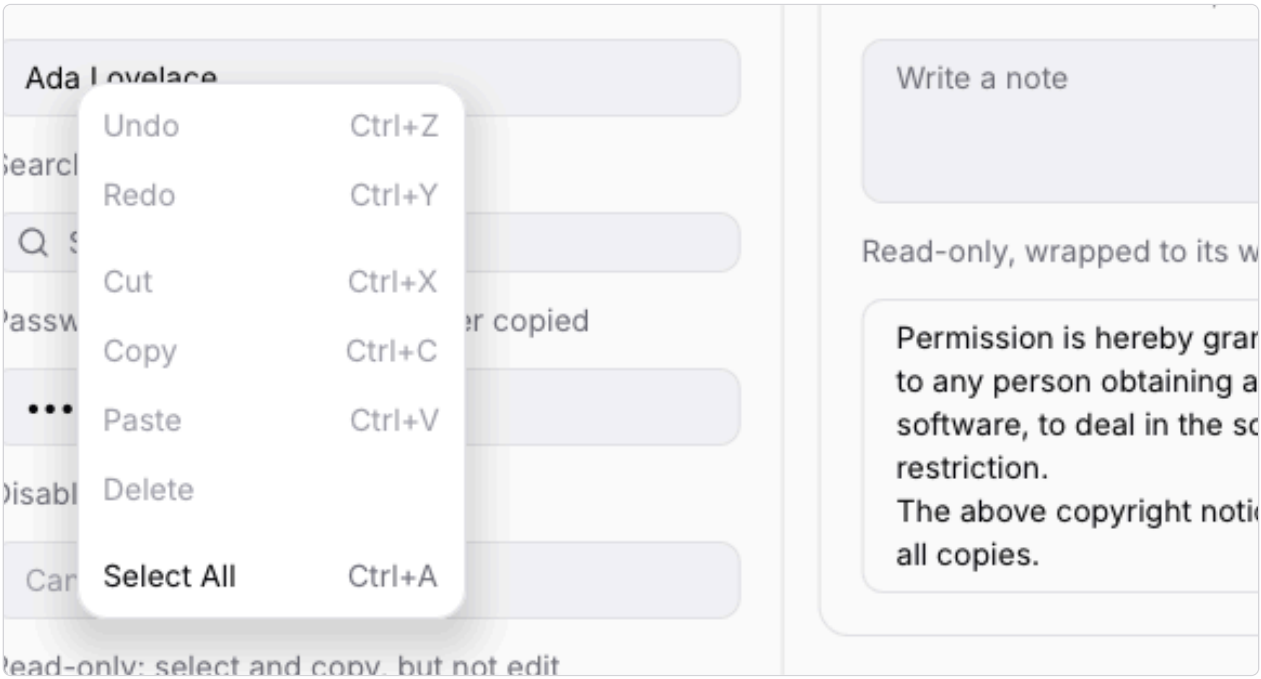


Figure 30: The edit menu of the Name field with PC keys. Nothing is selected, there is nothing to undo and the clipboard is empty, so only Select All can act.

15.10 The caret

A focused field's or area's caret blinks: on for `Ui.caret_blink` seconds, then off for as long. The rate is the user's, read from the OS with the other settings (Following the system); without one it is `ui::DEFAULT_CARET_BLINK`. When the user turned blinking off, `caret_blink` is −1 and the caret stays on. An app pins its own rate with `Ui.set_caret_blink`.

The cycle restarts, with the caret on, whenever the text changes, the caret or selection moves (by key or pointer), or the field gains focus, so the caret is visible where the user is working. It also stays on in three cases:

- after `ui::CARET_BLINK_TIMEOUT` seconds (10) since the cycle last restarted, as GNOME does, so an idle window stops redrawing for a caret nobody watches;
- under reduced motion (Reduced motion);
- when the user turned blinking off.

A blinking caret is the one thing on a still screen that changes, and an App-style loop sleeps while nothing moves (Idle). So while the caret blinks, the field asks for a frame at its next toggle with `Ui.wake_after`, and the loop wakes exactly then: on a platform that can sleep until a deadline (SDL3), a focused field draws two frames per blink cycle, and on one that can't (raylib), frames keep coming while the caret blinks (Timing). Once the caret stays on, it asks for nothing more.

15.11 Input method composition

Chinese, Japanese and Korean, among others, are typed through an input method: the user types "nihon", the input method offers "　" and other candidates, and the chosen text is committed into the field. Until it commits, the text is a **composition** that the input method owns. The backend reports it in `Input.composition`, with the input method's caret and the clause it is converting (The input of a frame).

The focused field or area shows the composition at its caret without putting it in the `DString`. The widget splices it into the text it shapes and lays out, so a right-to-left neighbor, bidi reordering and an area's wrapping place it as they would typed text, while clicks and keys still work on the text

itself. Each glyph of the composition is underlined one point thick, and those of the clause being converted two points; the underline goes glyph by glyph, so a composition that bidi splits in two is underlined where each part lands. The caret shows at the composition's own cursor, and the field or area scrolls to keep it in view.

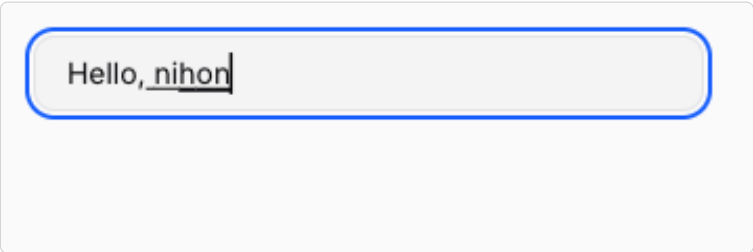


Figure 31: A field composing " nihon" after "Hello,": the composition underlined, "hon", the clause being converted, underlined thicker, and the caret at the composition's cursor.

A password field shows no composition, since input methods are off for passwords, and a read-only or disabled field shows none either.

Committed text arrives as typed text, in `Input.text`, and goes in through `TextEdit.type` like any other: the field's filter applies (Fields with more options) and it merges into the undo step being typed, as typing does (The editor). When composing starts over a selection, the selection is deleted at once, as a typing step that the committed text then joins, so one undo removes the committed text and brings the selection back.

While a composition is open, the keys are the input method's: it uses Enter to commit, Escape to cancel, Backspace to delete inside the composition and the arrows to move between clauses. `Ui.composing` is true while the focused field (`Ui.typing_field`) has a composition, and then `Ui.begin` marks every pressed key used and `Ui.typing_takes` takes every key. So Enter doesn't submit, Tab doesn't move focus, Escape doesn't close a dialog, no shortcut fires and Backspace doesn't delete from the field's text. The keys come back in the first frame without a composition.

A press in the field or area while it composes calls the platform's `end_composition()`, so the input method commits or drops the composition as the OS does, and places no caret: a later press does. The candidate list is the input method's own window, drawn by the OS; Kiln only tells the platform where the field and caret are, so it can place the window clear of the text (The platform and key routing).

15.12 The platform and key routing

When a field or area gains or loses focus, it tells the platform with `text_input(on, input)`, a `TextInput` that says what the field holds and where. This is the call that, on a platform with one, brings up the on-screen keyboard and starts or stops the input method. Its `kind` is the field's `InputKind` (`TEXT`, `NAME`, `EMAIL`, `USERNAME` or `NUMBER`, Fields with more options), `password` says the text is a password and `multiline` that Enter starts a new line, as in an area. `area` is the field's text rectangle, or for an area the caret's line across the area's width, and `caret` the caret's `x` from its left, both in window points, the text scale undone (Following the system), so the platform places the input method's windows next to the text.

While the field has focus, `text_input_area(area, caret)` follows the caret whenever it moves on screen: by typing, by moving the caret, or by scrolling. `Ui.tell_text_entry` makes both calls and keeps what it last told the platform, so a frame in which nothing moved tells it nothing. A read-only field or area makes neither call, since there is nothing to type (Read-only fields and areas). The raylib platform does nothing with them; Backends describes what the others do.

The rest of the frame has to know that a field is typing, so that its keys don't also do something else:

- `Ui.typing_takes` keeps `Ui.shortcut` from firing on chords the focused field uses: the keys that type, the editing keys, Enter, and the command chords for select all, copy, cut, paste, undo and redo, judged by `ui.text_keys` as the field judges them. A text area also takes Up, Down, Page Up, Page Down and Ctrl/⌘+Enter; `Ui.end` carries into the next frame whether the focused widget is an area. It works from the field that had focus in the last frame, so it applies even to shortcuts declared before the field is built. It also keeps the extra `keys` that field took (Fields with more options), which `Ui.end` carries into the next frame, but only while neither the command modifier nor Alt is held: a search field with text keeps Escape from a shortcut declared on it, and a number field keeps Up, Down, Page Up and Page Down, while Ctrl+Up still reaches its shortcut. Other chords, such as Ctrl+S, reach their shortcuts too. Typing in a text field describes it.
- `Ui.wants_keyboard` is true while a field or area has focus, so a Game-style loop doesn't move the player on the W key the user is typing into a chat box (Game style: your raylib loop). The field takes typed characters from `Input.text` and doesn't use the letter keys, so without this check the game would see them.

16 Inputs

Beyond the plain text field, a toolbar or a settings page needs a few more controls: a field to search with, a field for a number, a choice from a list, and a choice from a handful of options shown side by side. Kiln has one widget for each. The search and number fields are text fields with extra parts, built on the same field core as `Ui.text_field` the select is a button with a menu; the segmented control is a row of segments in one focus group. This chapter describes what each adds to the parts it is built from.

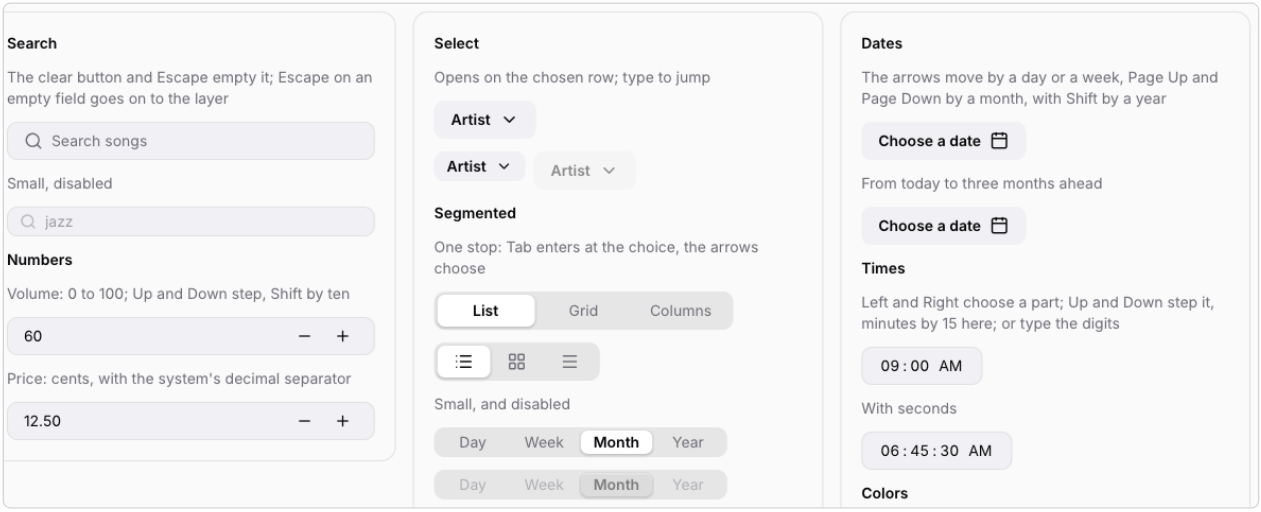


Figure 32: The gallery's inputs page in the light theme. On the left, a search field with its placeholder, a small disabled one, and two number fields with their − and + steppers; on the right, three selects (the last disabled), and four segmented controls: with labels, with icons only, small, and small and disabled.

16.1 The field core

The search and number fields don't copy the text field: they call `Ui.field_ex(label, value, opts)`, the text field with all its options, which is also what `Ui.text_field` calls. Its `FieldOpts` add three things to the text field's options, and the two widgets use all three:

- a `CharFilter`, a function that says whether the field accepts a code point, typed or pasted;
- a `FieldTrailing`, a function that builds widgets inside the field's box, after the text: the search field's clear button, the number field's steppers;
- a `KeySet` of keys the field takes while it has focus, beyond its own editing keys. The field core doesn't act on them; it reports them, so that `Ui.shortcut` doesn't fire on them while the field has focus (Typing in a text field), and the widget reads them itself after the field.

Everything else, the caret, the selection, undo, the mouse, the clipboard and the keys, is the text field's, as Text editing describes. The field's `TextFieldResult` also says when the focus `left` it this frame, which is when the number field checks what was typed.

16.2 Search

`Ui.search_field(label, value, placeholder, size, disabled, width)` is a text field with the `SEARCH` icon before the text and "Search" as its default placeholder. While the field holds text and isn't disabled, a clear button sits at its end: a small `GHOST` button with the `X` icon and the tooltip "Clear". It is gone while the field is empty, so an empty field shows nothing to clear.

```
ui.search_field("Search", &demo.query, placeholder: "Search the library");
```

The clear button and Escape both empty the field, and they do it the same way:

- the whole text is selected and replaced with nothing, so the clear is one undo step, and Ctrl+Z (⌘Z) brings the query back;
- the field keeps the focus, or takes it, so the user can type the next search at once; the caret's blink starts over, so the caret shows;
- the result has `cleared` set, along with `changed`.

The clear button isn't focusable. Tab goes from the search field to the next widget rather than to its own clear button, as in native search fields, and the keyboard has Escape to do what the button does.

16.2.1 Escape

Escape belongs to the search field only while the field has text. An empty field leaves it alone, so Escape goes on to the layer the field is in, and closes it as any Escape does (Dismissing). In a search field at the top of a popover or a dialog, the first Escape clears the query and the second closes the layer, as native search fields do.

The field says so in its `KeySet`: it adds Escape only in a frame where it has text. `Ui.shortcut` then leaves an Escape shortcut alone while a non-empty search field has focus, and fires it once the field is empty. The field takes the key with `Ui.key_pressed`, which uses it, so the layer, which looks at Escape after the widgets, doesn't also close.

16.3 Numbers

`Ui.number_field(label, value, min, max, step, format, size, disabled, width)` edits a `double`. It is a text field that accepts only the characters of a number, with − and + steppers at its end:

```
ui.number_field("Volume", &g.volume, min: 0, max: 100, step: 1, format: "%.0f");
ui.number_field("Price", &g.price, min: 0, max: 1000, step: 0.01, format: "%.2f");
```

`min` and `max` bound the value, and `step` is the grid it snaps to, counted from `min` (from 0 when `min` is so large that a `double` can't step finely there, as the default is); a `step` of 0 allows any value, and the keys and steppers then move by 1. `format` is a printf format for one `double`, `"%g"` by default. The field returns a `NumberResult`: `changed` when the value changed this frame, by a step or by typed text, and `submitted` when Enter was pressed in it.

16.3.1 What the field accepts

The field's `CharFilter` lets through the digits, `.`, the user's decimal separator, and `-` when `min` is below zero. Other characters are dropped as they are typed, and dropped from pasted text, so the field never holds letters. That doesn't make every text a number: `1.2.3` or a lone `-` can still be typed, and the field deals with them when it checks the text.

16.3.2 The text and the value

The field keeps its own text, apart from the app's value, in widget state (Widget state). While the field doesn't have focus, the text is the value written with `format`, rewritten whenever the value changes. Once the field takes focus, the text is the user's: the value may still change, by a step, but what the user types isn't replaced under the caret, and the value doesn't change with every keystroke, while `1` is on its way to `150`.

What was typed counts at two moments: when Enter is pressed, and when the focus leaves the field, by Tab or a click elsewhere. Then the field:

1. parses the text with `Ui.parse_number`
2. clamps the number to `min` and `max`, and rounds it to the nearest `step` from `min`;

3. stores it in `*value`, and writes the field's text again from the value, so the user sees `100` after typing `250` into a field whose maximum is 100.

Text that isn't a number leaves the value as it was, and the field shows the value again. After Enter, the field keeps the focus and the user goes on editing; after the focus leaves, the text follows the value again.

16.3.3 Steps

The keys step the value while the field has focus, and the steppers step it from the pointer:

Input	Step
Up, Down	one step up or down
Shift+Up, Shift+Down	ten steps
Page Up, Page Down	ten steps
− and +	one step, then repeating while held

A step starts from what was typed when the field is being edited and its text is a number, so typing `40` and pressing Up gives 41, not one more than the value before the typing. The result is clamped and snapped like typed text, and the field's text is rewritten from it.

The field takes Up, Down, Page Up and Page Down in its `KeySet`, so they don't reach shortcuts or move the focus while the field has focus (Typing in a text field).

The steppers are small `GHOST` buttons with the `MINUS` and `PLUS` icons. Like the search field's clear button, they aren't focusable: the keys already do what they do. The − stepper is disabled while the value is at `min`, and the + stepper while it is at `max`, so the user sees that a bound has been reached.

A stepper steps once when pressed, waits `ui::STEP_REPEAT_DELAY` seconds (0.4), and then steps every `ui::STEP_REPEAT_EVERY` seconds (0.06) while the pointer holds it, until it reaches its bound. Holding a stepper moves nothing on screen between steps, and an App-style loop sleeps while nothing animates (Idle), so the field asks for a frame at the next step with `Ui.wake_after`, as a tooltip asks for one when its delay runs out (Timing).

16.3.4 The decimal separator

Numbers are written and read with the user's decimal separator, `Ui.decimal_separator`: `.` in English, `,` in German and many other regions. `Ui.format_number(v, format)` formats with `format` and replaces the `.` with the separator, in the temp allocator. `Ui.parse_number(text)` reads the separator back as the decimal point; it also accepts a `.`, so a user who types a point in a region that writes commas still gets a number. Text that isn't a number is a fault.

The separator comes from the OS, as the other settings do (Following the system): from the region settings on Windows and macOS (`LOCALE_SDECIMAL`, `NSLocale`'s `decimalSeparator`), and from the `LC_NUMERIC` locale on Linux. Where the OS has none it is `.`. An app pins its own with `Ui.set_decimal_separator`, until `Ui.follow_os` with `OsSetting`'s `DECIMAL_SEPARATOR` hands it back (Overriding a setting). Kiln has no thousands separator: a number field shows `1234.5`, not `1,234.5`.

16.4 Select

`Ui.select(label, selected, options, size, disabled, full)` chooses one of `options` into `*selected`, an index; it returns true once, in the frame the choice changes. The `label` names it and gives its id (The core calls); `full` makes it as wide as its container.

```
ui.select("Sort", &demo.sort, { "Title", "Artist", "Album", "Date added" });
```

On screen it is a button showing the chosen option, with a chevron at its end. A press opens the list of options below it, a menu (Menus, popovers and tooltips) whose rows are the options, the chosen one with a check mark. The list is as wide as the button, and wider when its longest option needs it, never narrower than `ui::MENU_MIN_WIDTH`. Choosing a row sets `*selected` and closes the list.

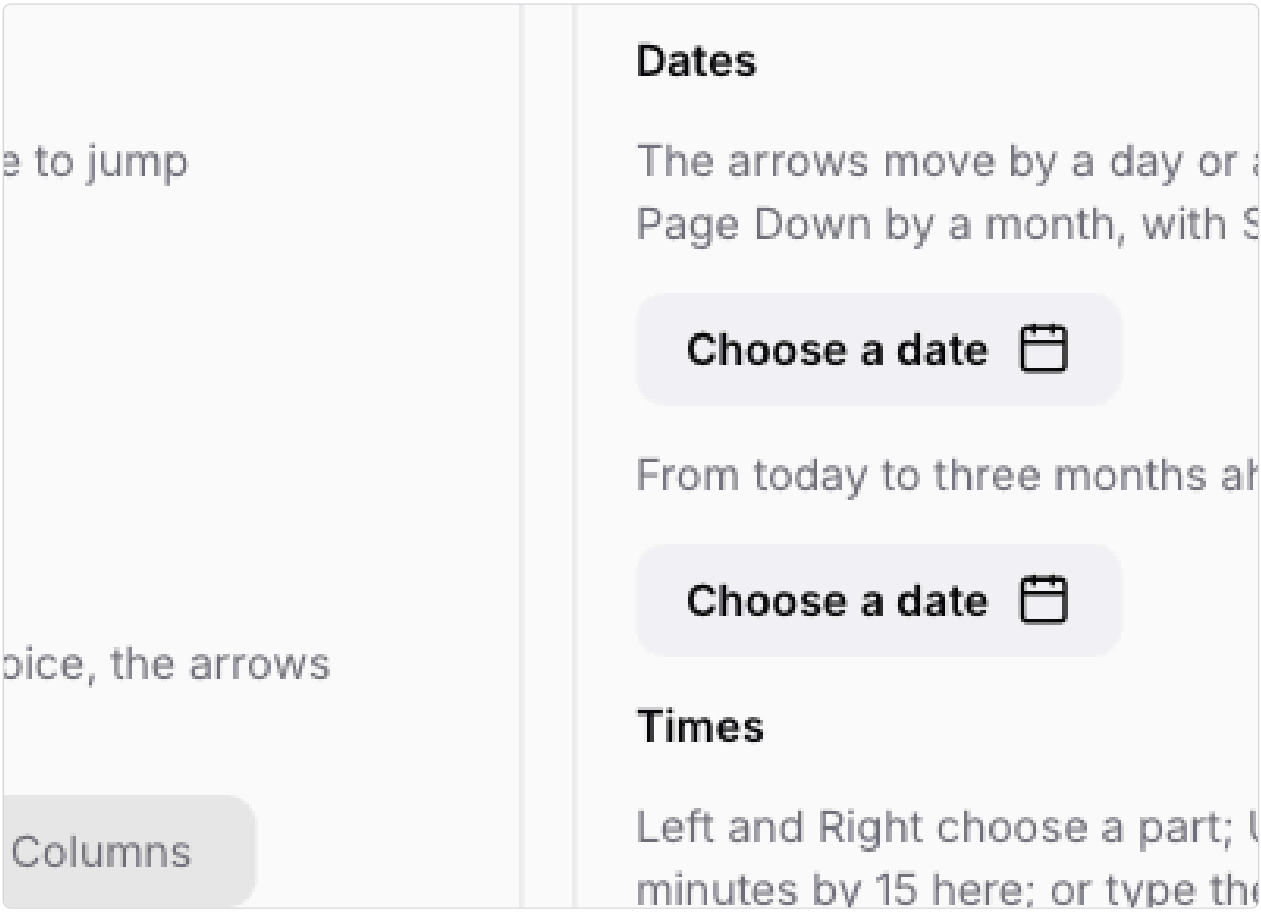


Figure 33: A select's list, opened below its button. The chosen option, Artist, carries the check mark; the list covers the controls beneath it until it closes.

16.4.1 Where the list opens

The list's place follows the platform, through `Ui.list_placement`, a `ListPlacement`:

- `BELOW`, on Windows, Linux, Android and the web: the list opens under the button, like any menu, and the button shows `CHEVRON_DOWN`.
- `OVER`, on macOS: the list opens over the button, with the chosen row exactly where the button was, so the pointer that clicked the button rests on the current choice. The button shows `CHEVRONS_UP_DOWN`, as a macOS pop-up button does.

`OVER` needs to know how far the chosen row is from the top of the list, and the list isn't laid out yet when it opens. So the select measures it from the last frame: while the list is open, it notes where the chosen row sat in the list, and places the list that far above the button's top, the row's middle on the button's middle. In the frame the list opens, before it has been laid out, it uses a guess from the list's padding and the rows above the chosen one. Since a layer only shows once it has been measured (Presence and motion), the user never sees the guess. When the window has no room above the button for the rows before the chosen one, the list shifts down to stay inside it, and the chosen row lands below the button.

The placement is the platform's by default, from `ui::platform_list_placement` `Ui.set_list_placement` pins it, until `Ui.follow_os` with `LIST_PLACEMENT` hands it back.

16.4.2 Keys

The list opens with the chosen row focused, so the arrows move from the current choice rather than from the top. It is a menu, so its keys are a menu's (Keyboard): Up and Down move, Enter or Space choose, and Escape or gamepad B close it. During a type-ahead search, Space is part of the name typed, so "date a" finds "Date added"; the key that opened the list never starts a search. When it closes, chosen or dismissed, focus goes back to the select's button.

The closed button has keys of its own while it has focus:

Key	On the closed button
Enter, Space	open the list, as for any button; Space typed during a type-ahead search is part of the name instead
Up, Down	open the list
Typing	chooses the option that starts with the typed text, without opening the list

Typing in the open list moves the focus to the matching row instead, and Enter then chooses it. Both use type-ahead (Type-ahead).

16.4.3 Ids

Each select builds its list as a menu named `"list"`, inside an id scope named by the select's label (`Ui.@id`). Two selects on one page therefore don't share their list's id, its state or its rows' ids, even though both call their menu `"list"` and may hold options with the same names.

16.5 Type-ahead

`Ui.type_ahead(ta, labels, from)` is how a list jumps to a row by typing its start, as native lists do: it takes this frame's typed text from `Input.text`, and returns the index of the label it leads to, or −1 when nothing was typed or nothing matches. The select uses it on its closed button and in its open list; a list of the app's own can use it the same way, keeping a `TypeAhead` in its state.

- Typed characters gather into one search while they come less than `ui::TYPE_AHEAD_PAUSE` seconds (1 s) apart; after a longer pause, the next character starts a new search. So typing "da" quickly finds "Date added", and "d", a pause, then "a" finds a row starting with "d" and then one starting with "a".
- A search starts on the row after `from`, the current one, and goes round to the start. A longer search may stay on the current row, so typing "Ar" while "Artist" is chosen keeps it.
- Typing the same letter again moves on to the next row that starts with it, rather than looking for a row that starts with "aa".
- Matching ignores ASCII case. Control characters are dropped, and the search holds at most 32 bytes.

16.6 Segmented control

`Ui.segmented(id, selected, options, icons, icons_only, size, disabled)` chooses one of a few options shown side by side, in a track with a raised pill under the chosen one. It returns true when the choice changes.

```
ui.segmented("view", &demo.view, { "List", "Grid" }, icons: { LIST, LAYOUT_GRID },
icons_only: true);
```

With `icons`, one per option, each segment shows its icon before its label, or the icon alone with `icons_only`. The label still gives the segment its id. A label may carry a `##` suffix (The core calls), which gives an id without being drawn.

16.6.1 One focus stop

A segmented control works like a radio group: it is one stop for Tab, not one per segment. Its segments are in a horizontal `Ui.@focus_group` (Focus), and the group's entry is the chosen segment (`Ui.set_group_entry`), so Tab lands on the current choice. Left and Right then move the focus along the segments, and moving the focus chooses: the segment that the arrows bring the focus to becomes the choice, without Enter. A click or Enter or Space on a segment also chooses it.

16.6.2 Layout

Every segment is as wide as the widest one, so the control doesn't look lopsided and the pill keeps one size as it moves. The widest is measured with its label in semibold, the weight of the chosen segment's label, so a label doesn't change the width when it is chosen. The other labels are regular, in the secondary text color.

The track is the text color at a low opacity, with the control corner radius. The pill sits 3 points inside it, a little smaller in radius, so the gap around it is even. It is `SURFACE_RAISED` with a soft shadow: white on a light theme, and on a dark theme a step lighter than that, mixed toward the text color, since a raised surface on dark is lighter rather than shadowed. An unchosen segment shows a faint tint while the pointer is over it, and the pointer cursor; a segment with visible keyboard focus draws a focus ring.

16.6.3 The pill

The pill slides from one segment to the next. Its left edge and its width are two springs (Duration and bounce) that go, with the `STATE` motion, toward where the chosen segment was laid out in the last frame. The first time the control is laid out, the springs start where the chosen segment is, so the pill doesn't fly in from the edge. Since the springs follow the segment's laid-out place, the pill also follows a control that changes width with its window. Under reduced motion, the springs settle at once (Reduced motion).

16.6.4 Disabled

A `disabled` control takes neither the pointer nor the focus. Its icons, labels and pill are drawn at `ui::DISABLED_OPACITY`, and the pill still shows the choice.

16.7 Dates

`Ui.date_picker(label, date, min, max, placeholder, size, disabled, full)` chooses a day into `*date`. It is a button showing the date as the user's region writes it, `25/11/2026` or `11/25/2026`, with a calendar icon. A press, or Down while it has focus, opens a month calendar under it. It returns true once, on the frame the date changes. `label` names the picker and is its id; while no date is set, the button shows `placeholder`.

```
Date due;
if (ui.date_picker("Due", &due, min: ui::today())) save_due(due);
```

A `Date` is a day of the Gregorian calendar: a year, a month from 1 to 12 and a day from 1 to 31, with no time and no time zone, so an app can store one as it is. Its zero value means no date (`Date.is_set`). `Date.add_days` and `Date.add_months` move it, the second keeping the day inside the new month (31 January and a month is 28 or 29 February); `Date.weekday` gives 0 for Sunday to 6 for Saturday, and `Date.compare` orders two dates. `ui::today` is the local date. `Ui.format_date` writes a date as the button does, in the temp allocator.

16.7.1 The calendar

The calendar is a popover (Menus, popovers and tooltips) with a month button and a year button between buttons for the previous and next month, a row of short weekday names, and six weeks of days: always six rows, so the popover keeps one height from month to month. Weeks start on the region's first weekday. Days of the months before and after are shown dimmed and can be chosen too. The chosen day is filled with `ACCENT`, and today has an `ACCENT` ring.

`min` and `max`, when set, bound the days that can be chosen. Days outside them are drawn at `ui::DISABLED_OPACITY` and take no clicks, and the previous and next buttons stop at the months that hold allowed days.

16.7.2 Keys

The grid of days is one focus stop holding a cursor day, as a list is one stop holding a cursor row (Collections), not 42 stops. The calendar opens with the cursor on the chosen date, or on today when none is set, and with focus on the grid.

Key	Cursor
Left, Right	a day back or on
Up, Down	a week back or on
Home, End	the start or the end of the week
Page Up, Page Down	a month back or on, keeping the day inside the month
Shift+Page Up, Shift+Page Down	a year back or on
Enter, Space	chooses the day and closes the calendar
Escape	closes without choosing

The gamepad's d-pad moves the cursor as the arrows do. The cursor stops at `min` and `max`, and moving it into another month shows that month. Escape, a press outside and the back gesture close the calendar as they close any layer, and `Ui.dismissal` reports them (How it was dismissed); the popover is named `calendar` under the picker's label, so `ui.@id("Due") { ui.dismissal("calendar") }` reads it. When the calendar closes, focus goes back to the button.

16.7.3 Months and years

The month button replaces the days with the year's twelve months, three to a row, and the year button with a page of twenty years, four to a row, from a year divisible by twenty. A second press on the same button goes back to the days. In these views the previous and next buttons move by a year or by a page. Choosing a month or a year shows its days, with the cursor on the same day of the month where the month has it, and focus back on the days; the date itself changes only when a day is chosen. Months and years with no day between `min` and `max` are dimmed and take no clicks.

Each view is one focus stop, as the days are. In the months the arrows move within the year and Page Up and Page Down move by a year; in the years the arrows move by a year or a row and Page Up and Page Down by a page. Enter or Space shows the days. Both views fill the area of the weekdays and the days, so the popover keeps its size.

16.7.4 The region's calendar

The first weekday, the order and separator of a short date, whether the day and month have a leading zero, and the names of the months and weekdays come from the OS, through `os::calendar_settings` (The calendar settings). `Ui.calendar` holds them. The `Ui` reads them the first time a picker asks, not when it starts, so an app without a picker never reads them and never

carries the code that does (What an app carries). Tests and apps that want fixed settings pass their own to `Ui.set_calendar`, as `os::english_calendar` builds them.

16.8 Times

`Ui.time_field(label, time, minute_step, seconds, size, disabled, full)` sets a `TimeOfDay` into `*time`. It is a field of parts: the hour, the minute, the second when `seconds` is set, and the day period on a 12-hour clock, written as the user's region writes a time: `09:30 AM`, `09.30`, `09:30`. It returns true on each frame the time changes, as a slider does, and `label` names it and is its id.

A `TimeOfDay` is an hour from 0 to 23, a minute and a second, with no date and no time zone; its zero value is midnight, so a field always holds a time. `TimeOfDay.seconds` counts seconds from midnight, to compare and sort times, and `ui::now_time` is the local time. `Ui.format_time` writes a time as the region does, in the temp allocator: `09:05` on a 24-hour clock, `9:05 AM` on a 12-hour one. The field pads the 12-hour hour to two digits as well, so its width doesn't change as the hour steps.

16.8.1 Keys

The field is one focus stop holding a focused part, which is filled with `SELECTION` while the field has focus. Tab arrives on the first part, Shift+Tab on the last, and a press on a part focuses the field on it; a press on the day period also switches it.

Key	Does
Left, Right	the part before or after; they stop at the ends
Up, Down	the part one step on or back, wrapping without changing the other parts; the minute steps by `minute_step`, landing on its multiples, and the hour steps through the day periods as a clock does
Page Up, Page Down	ten on or back
Home, End	the part's lowest or highest value
digits	fill the part, below
A, P	AM or PM, on the day period; so do the first letters of the region's words
Escape	the time the field had when it took focus
Enter	confirms a dialog around the field, as in a text field

The gamepad's d-pad moves and steps as the arrows do. While the field has focus it takes typing (Typing in a text field), so digits and letters don't reach shortcuts or type-ahead. Escape is taken only while there is a time to put back; otherwise it goes on to the layer or dialog around the field. On a touch screen the field asks the platform for a numeric keypad.

16.8.2 Typing digits

A part takes up to two digits. A first digit that can't begin a two-digit value fills the part and moves on at once: 3 to 9 for a 24-hour hour, 2 to 9 for a 12-hour one, 6 to 9 for a minute or a second. Any other first digit is shown and waits: a second digit that makes a valid value fills the part and moves on, and one that doesn't (`2` then `5` for an hour) starts the part again as a first digit. A 12-hour hour keeps its day period, so `0` then `9` on 2 PM gives 9 PM. The separator, or a space, after a single digit moves on as well. Typed minutes aren't rounded to `minute_step`.

16.9 Colors

`Ui.color_picker(label, color, alpha, swatches, size, disabled)` chooses a `Color` into `*color`. It is a button showing a chip of the color and its hex value, `#3B82F6`, with an outline. A press, Enter,

Space or Down opens a popover under it, and the color changes as the user moves in it: the picker returns true on each frame the color changes, as a slider does. `label` names the picker and is its id. With `alpha` the popover has a bar for the opacity and the hex value carries it, `#3B82F680`; without it the color is opaque. `swatches` are colors the app offers to choose with one press, such as its brand colors.



Figure 34: The color picker open on the gallery's Accent, with Tailwind's 500 colors as swatches: the saturation and brightness area, the hue bar, the hex field and the swatches, the current one ringed.

16.9.1 The popover

From the top, the popover holds:

- **the saturation and brightness area**: saturation from left to right, brightness from top to bottom, painted with two gradients over the pure hue;
- **the hue bar**, drawn by `DrawList.hue_rect`

- **the alpha bar**, with `alpha`: the color fading from clear to opaque over a checkerboard;
- **the hex field**: it takes hex digits and `#`. What was typed counts on Enter and when the focus leaves, as in a number field (Numbers): `#rgb`, `#rgba`, `#rrggbb` and `#rrggbbaa`, in either case. Text that isn't a color puts the color's hex back. A hex without alpha keeps the color's own opacity;
- **the swatches**, when there are any: rows of eight, the one that is the current color ringed. A swatch keeps its own alpha only when the picker has `alpha`.

The popover opens with focus on the area. Escape closes it and puts back the color it had when it opened, so Escape means close without choosing, as in the date picker; `Ui.dismissal` reports it, under the name `color` in the picker's label. A press outside or the back gesture closes it and keeps the color. Either way, focus goes back to the button.

16.9.2 Keys

The area, each bar and the swatches are one focus stop each, in that order with the hex field between the bars and the swatches.

Where	Keys
area	Left and Right move the saturation by 1%, Up and Down the brightness; with Shift by 10%. Home and End go to no and full saturation, Page Up and Page Down to full and no brightness
hue bar	Left and Down, Right and Up move 1°, with Shift 10°; Home and End go to 0° and 360°
alpha bar	the same, by 1% and 10%; Home is clear and End opaque
swatches	the arrows move the cursor swatch, a row of eight up or down; Home and End go to the first and last; Enter or Space chooses

The gamepad's d-pad moves as the arrows do. A press in the area or on a bar sets the color there, and dragging follows the pointer.

16.9.3 HSV and the hue of a grey

The picker shows HSV (`Hsv`), the model people expect of a color picker: every HSV color is an sRGB color, so nothing needs mapping into the gamut. Themes and palettes use OKLCH (Theme), which matches how light colors look but has colors no screen shows. `Color.to_hsv` and `Hsv.to_color` convert; `Color.to_hex` and `ui::color_from_hex` write and read hex.

A grey, white or black has no hue, and black no saturation either. So the picker keeps its own HSV while the user works: dragging to black and back up keeps the hue. It reads the color again only when the app changes `*color` to something else, which it notices by comparing it with the color it last wrote.

16.10 Across platforms

	Select list	Decimal separator
Windows	below the button	the region's (`LOCALE_SDECIMAL`)
macOS	over the button, chosen row on it	the region's (`NSLocale`)
Linux	below the button	the `LC_NUMERIC` locale's
Android	below the button	.
Web	below the button	.

The rest is the same everywhere: the search field's Escape, the number field's keys and steppers, type-ahead and the segmented control's keys. The text editing inside the search and number fields follows the platform's keys, as every text field does (Keys).

17 Navigation

An app with more than one page needs a way between them, and a page with more than fits needs a way to show part of it at a time. Kiln has three widgets for this: tabs, a row of pages across the top of a view; a sidebar, a column of pages along the window's edge that turns into a drawer on a narrow window; and disclosures, sections that open and close in place, alone or as an accordion. None of them shows the content: each tells the app what is chosen or open, and the app builds that content, as with any widget that binds to the app's data (Return values and binding).

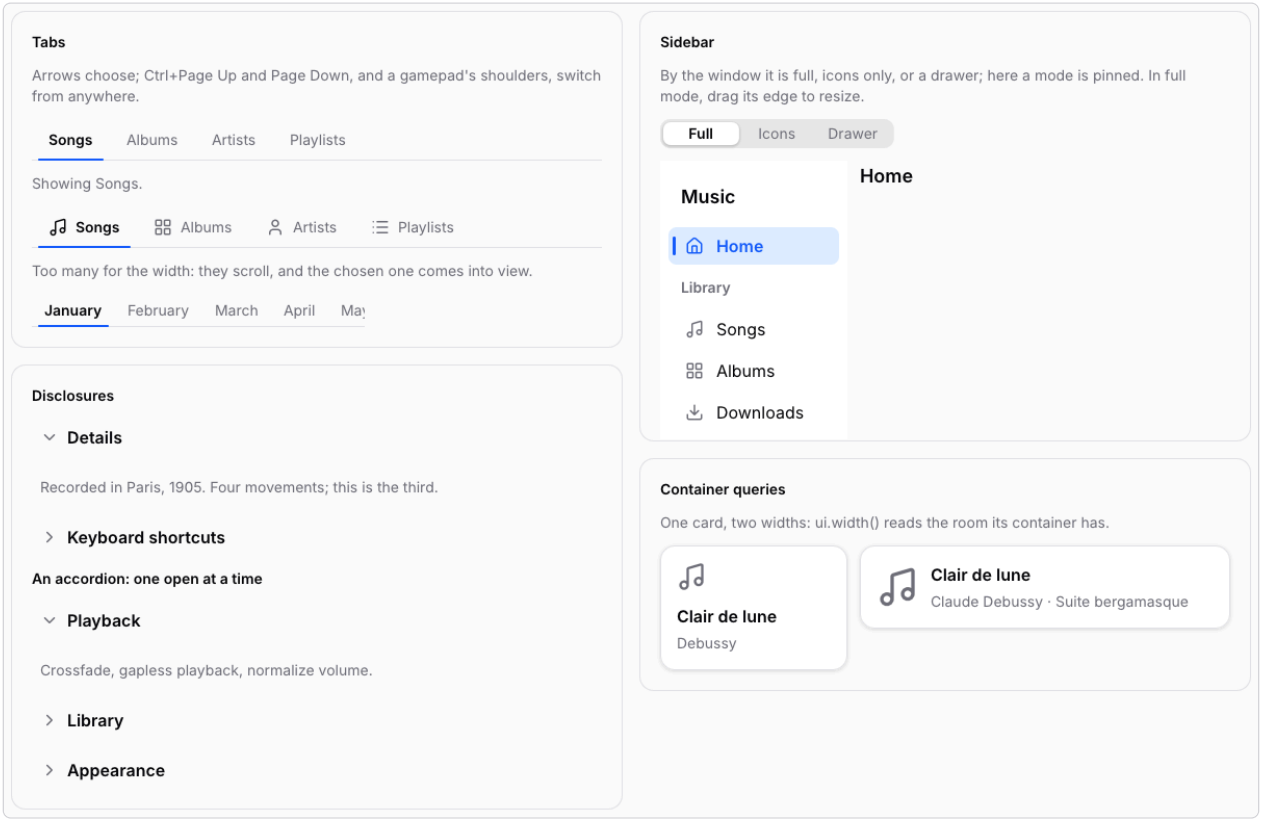


Figure 35: The gallery's navigation page in the light theme. On the left, three tab bars (plain, with icons, and small months scrolling sideways), two disclosures and an accordion; on the right, a sidebar pinned to its full mode, and one card at two widths, built for the room its container has.

17.1 Tabs

`Ui.tabs(id, selected, labels, icons, size, keys, badges)` chooses one of `labels` into `*selected`, an index, and returns true in the frame the choice changes. With `icons`, one per label, each tab shows its icon before its label. With `badges`, also one per label, a tab shows a count after its label when its badge is above 0, a dot at 0, and nothing below 0 (Badges).

```
ui.tabs("library", &g.tab, { "Songs", "Albums", "Artists", "Playlists" });
switch (g.tab)
{
    case 0: songs(ui, g);
    case 1: albums(ui, g);
    ...
}
```

17.1.1 The look

Each tab is as wide as its label, with the control size's padding on both sides; the bar doesn't stretch them to fill its width. The chosen tab's label is in `TEXT` and semibold, the others in `TEXT_SECONDARY` and regular, and a tab that isn't chosen shows a faint tint under the pointer. A hairline in `BORDER` runs under the whole bar, as wide as its container, and a 2-point `ACCENT` underline marks the chosen tab, a little narrower than the tab.

The underline slides from tab to tab. Its left edge and its width are two springs (Springs find their frame), moving with the `STATE` motion token (Motion tokens), the one a checkbox or a switch uses. Their targets come from where the chosen tab was laid out last frame, read with `rect_of`: a widget doesn't know its own place while it is built, and a frame later the tab has not moved. So the underline glides to a newly chosen tab and stretches or shrinks to its width on the way. In the bar's first frame, before there is a last frame to read, there is no underline; in the second, it appears under the chosen tab without sliding.

17.1.2 Keys

The bar is one focus stop, a horizontal focus group (Focus) whose entry is the chosen tab. Tab enters the bar at the chosen tab and leaves it at the next stop; Left and Right move along it, and choose the tab they move to, as a radio group does. A click, Enter, Space or gamepad A on a tab chooses it too.

Two inputs switch tabs from anywhere in the window, not only from the bar, and go round from the last tab to the first and back:

Input	Tab
Ctrl+Page Down, Ctrl+Page Up	the next and the previous; ⌘ instead of Ctrl with Mac keys
Gamepad RB, LB	the next and the previous

The keys are a `Ui.shortcut` with `primary` set, so they follow the platform's command modifier and stay quiet when a text field uses the chord, and the shoulder buttons go through `Ui.pad_pressed`. Both act only while the bar's layer is on top: with a dialog open over the page, they don't switch the page's tabs beneath it. A page with two tab bars gives one of them `keys: false`, so a single press doesn't switch both; that bar is then chosen with the pointer and its own arrows.

17.1.3 More tabs than room

The tabs sit in a horizontal scroll area (`Ui.@scroll` with `HORIZONTAL`, Scrolling), as tall as the tabs. When they are wider together than the bar, they scroll sideways with the wheel or a drag, and the hairline stays put under the area. When the choice changes to a tab that is partly or wholly out of view, by the keys or the shoulders, the bar scrolls it in with `Ui.scroll_to`, so that it just fits at the edge it was past. This happens once per change, so the user can scroll away from the chosen tab afterwards and it stays where they left it.

17.2 Sidebar

`Ui.@sidebar(id, title, mode, width, resizable, min_width, max_width, drawer)` is a column of pages along the window's edge. Its body builds the items and the headings between them:

```
ui.@sidebar("pages", title: "Kiln")
{
    for (int i = 0; i < PAGES.len; i++)
    {
        if (i == 1) ui.sidebar_section("Library");
```

```
        if (ui.sidebar_item(PAGES[i], PAGE_ICONS[i], selected: demo.selected == i))
demo.selected = i;
    }
};
```

`Ui.sidebar_item(label, icon, selected, badge)` is one page: it returns true when chosen, and the app keeps which page is selected and passes it back as `selected`. `label` is the item's id. `badge` marks the item: above 0 it shows that count at the row's end, at 0 a dot, and in icons mode a dot on the icon; below 0, the default, nothing (Badges). `Ui.sidebar_section(title)` heads the items after it. `title` heads the sidebar itself, and `width`, 220 points by default, is its width where it shows labels. A label too long for that width is cut with "…" and shows in full as a tooltip, so a long playlist name never widens the sidebar. Anything else wider than the sidebar, such as a scroll area of rows, is cut off at its edge.

With `resizable:`, a full sidebar has a grip at its end edge: the pointer becomes a left-right resize cursor over it, dragging sets the width between `min_width` (160) and `max_width` (360), and a double-click puts `width` back. The grip is invisible until the pointer is over it. Icons mode stays 56 points wide, and the drawer keeps `width`, clamped so a strip of the page stays visible. A custom sidebar, such as the gallery's, gets the same grip from `Ui.sidebar_resize_grip(panel, &width, current, reset, min, max)`, and puts its page list in a scroll area so the list follows the window height.

17.2.1 Modes

A `SidebarMode` says how the sidebar shows:

Mode	The sidebar
`FULL`	icons and labels, `width` points wide, under the title
`ICONS`	icons only, `ui::SIDEBAR_ICONS_WIDTH` (56) points wide, each label a tooltip; no title, and a separator for each section
`DRAWER`	a menu button; the full sidebar opens over the page from the window's start edge
`AUTO`	by the window's breakpoint (Responsive layout): `FULL` at `LG` and wider, `ICONS` from `MD`, `DRAWER` below

`AUTO` is the default, so one sidebar fits a desktop window, a narrow one and a phone without the app deciding. An app pins a mode where it knows better, as the gallery does to show all three side by side.

17.2.2 The look

The sidebar is a plain panel, the full height of where it is put and a shade off the page: `SURFACE_RAISED` on the page's `BACKGROUND`, with no rounding, border or shadow. It is not a card floating on the page but a part of the window, so an app puts it flush against the window's edge and pads its content instead of the window. The `hello` example's shell does that:

```
// The sidebar runs along the window's edge; the content keeps its margin.
ui.@stack(ui.at(MD, Dir.ROW, Dir.COLUMN), width: ui::GROW, height: ui::GROW)
{
    sidebar(ui, demo);
    ui.@column(width: ui::GROW, height: ui::GROW, pad: 6) { cards(ui, demo); };
};
```

From `MD` up the sidebar runs down the left edge beside the content; below `MD` it is in drawer mode, and its menu button heads a column above the content.

An item is a row 36 points tall with the icon and the label. Under the pointer it takes a tint; once selected, the tint is `ACCENT_SUBTLE`, the icon and the label turn `ACCENT`, the label semibold, and an accent bar at the item's start edge grows from the top to nearly its full height. The tint and the bar follow one spring with the `ROW` motion token, so moving the selection fades one item out as the next fades in. In icon mode, the icon is centred in the narrow column, there is no bar, and the label shows as the item's tooltip (Tooltips). A section is a small `TEXT_SECONDARY` heading in full mode, and a separator in icon mode, where there is no room for words.

17.2.3 Keys

The items form a vertical focus group whose entry is the selected item. Tab enters the sidebar at the current page, Up and Down move between items, and Enter, Space or gamepad A choose one. The arrows move focus without choosing: unlike a tab, an item may open a page that is slow to build, so the user picks it on purpose.

17.2.4 The drawer

In drawer mode, the sidebar is a menu button, an icon button with the `MENU` icon and the title as its tooltip. Pressing it opens a modal layer (Layers) placed against the window's start edge with `at_edge` (Against a window edge), as tall as the window and `width` wide, clamped so a strip of the page stays visible, holding the full sidebar: title, sections and items, with a large shadow. The page beneath is dimmed and takes no input while it is open.

The drawer closes when the user chooses an item, presses Escape or gamepad B, or presses outside it (Dismissing). As any layer that takes focus, it moves the focus into itself on opening and back to the menu button on closing.

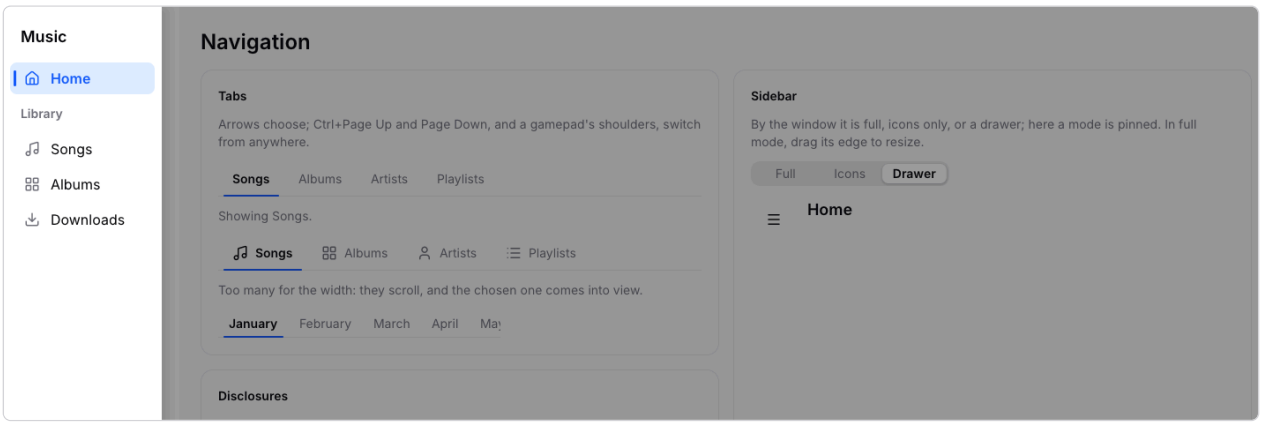


Figure 36: The sidebar in drawer mode, opened. The drawer covers the window's start edge with the full sidebar; the page beneath is dimmed, and its menu button, under the drawer, sits beside the selected page's title.

The sidebar keeps the drawer's open flag in its own state. With `drawer:`, the app holds it instead, in a `bool` of its own, to open the drawer from a shortcut, say, or to know whether it is open. A sidebar that leaves drawer mode, because the window grew, closes the drawer, so it doesn't reappear when the window shrinks again.

17.3 Disclosures

`Ui.@disclosure(title, open)` is a section that opens and closes under a header. The header shows a chevron, pointing right while closed and down while open, then the title; a click, Enter, Space or gamepad A on it flips `*open`. `title` is its id. The body is whatever the macro's body builds:

```
ui.@disclosure("Details", &g.details)
{
    ui.secondary("Recorded in Paris, 1905. Four movements; this is the third.");
};
```

Each disclosure has its own flag, so any number of them can be open at once.

17.3.1 Sliding open and closed

The body's height slides between nothing and its own height with the `DISCLOSURE` motion token. Since a widget doesn't know its height while it is built, the disclosure builds the body inside an inner box with an id, and reads that box's height from the last frame. Around it, an outer box is as tall as that height times how far open the spring is, and clips what doesn't fit, so the body is uncovered from the top as it opens and covered again as it closes.

Once the spring settles open, the outer box is `FIT` instead of a fixed height and stops clipping. The body's content can then change, a line of text grow or a nested disclosure open, and the section grows with it instead of cutting it off. In the frame a disclosure opens, before its body has been measured, the outer box is `FIT` as well. Under reduced motion the spring jumps (Reduced motion), so the body appears and disappears at once.

17.3.2 Focus while closed

While a disclosure closes, its body is still built, to be seen sliding away, but its widgets take no focus: the disclosure marks them inert, so they register no focus stops and Tab skips them. Once the body has slid shut, it isn't built at all, so a closed section's widgets cost nothing and are no focus stops. An app keeps its data in its own state, not in the closed section's widgets, so nothing is lost when they stop being built.

17.3.3 Accordions

`Ui.@accordion(id, open_index)` keeps at most one of its sections open. Each `Ui.@section(title)` in its body is a disclosure that is open when it is the accordion's open one. `*open_index` is that section's index, counting the sections in the order they are built, and −1 when none is open. Opening a section closes the one open before, and each slides on its own spring, so one shuts while the next opens; closing the open section leaves none open.

```
ui.@accordion("settings", &g.setting)
{
    ui.@section("Playback") { ui.secondary("Crossfade, gapless playback, normalize
volume."); };
    ui.@section("Library") { ui.secondary("Watched folders and how they are
scanned."); };
    ui.@section("Appearance") { ui.secondary("Theme, accent and density."); };
};
```

A section belongs inside an accordion; outside one, its contract fails.

17.4 Fitting the container

Navigation changes with the room it has: the sidebar picks its mode from the window, and a page often needs to pick its own layout from the width of the container it lands in rather than the window's. `Ui.width` reads the width of the container being built, and `Ui.width_at(min, wide, narrow)` picks between two values by it, as `Ui.at` does by the window (Layout). The gallery's song card uses it to lay itself out in a row when it has 320 points or more, and in a column below, so one card shows two ways on one page:

```
fn void song_card(Ui* ui, String id, Size width)
{
    ui.@card(id, width: width, pad: 4, gap: 2)
    {
        bool wide = ui.width_at(320, true, false);
        ui.@stack(wide ? Dir.ROW : Dir.COLUMN, width: ui::GROW, gap: 3)
        {
            ui.icon(MUSIC, wide ? 40 : 28, TEXT_SECONDARY);
            ...
        };
    };
}
```

18 Collections

Most apps show many things of one kind: playlists, songs, files, contacts. Kiln has three widgets for them. A list is a column of rows the user can select and open; a long list builds only the rows in view, so its length costs nothing. A table is a long list with columns, which the user can sort and resize. A grid lays out cards or tiles in columns that wrap, in the manner of Tailwind's grid. Lists and tables never hold the app's data: the app keeps its rows in its own arrays, tells the widget how many there are, and builds or describes each row when asked, as any widget binds to the app's data (Return values and binding).

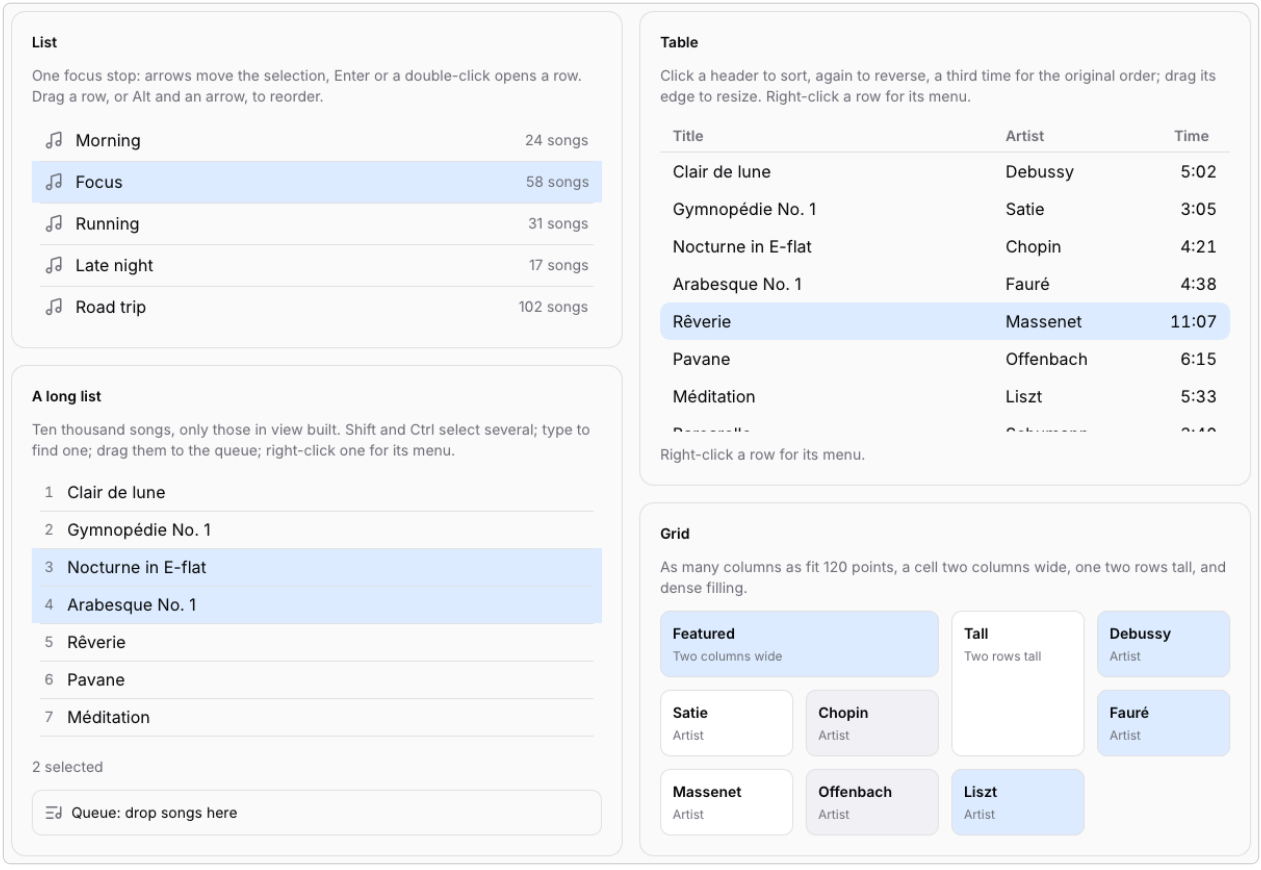


Figure 37: The gallery's collections page in the light theme. On the left, a list of playlists with dividers and one selected row, which can be dragged into a new order, and a long list of ten thousand songs with two rows selected and a queue to drag them to; on the right, a table of tracks with its Time column aligned to the end, and a grid of tiles: one two columns wide, one two rows tall, with Debussy and Fauré in the column beside the tall tile.

18.1 Selection

A list or a table selects one row or several. For one, the app passes `selected:`, a pointer to an `int` holding the selected row, −1 for none. For several, it passes `selection:`, a `Selection` it owns:

```
Selection picked;
picked.init();        // once
...
picked.free();        // when done
```

A selection is a bitset over the app's row indices, one bit per row, with an anchor, the row Shift ranges start from (-1 for none), and a count of the rows selected. Rows are the app's indices, not the positions they show at, so a table that sorts its rows keeps the same rows selected.

The bitset starts empty and grows only when a row past its end is selected. A selection over ten thousand rows of which the first few are chosen holds a word or two, and steady frames allocate nothing. `Selection.has` asks about one row, `Selection.@each` runs over the selected rows in order, and `Selection.first` gives the first; `Selection.set`, `Selection.select_only`, `Selection.add_range`, `Selection.select_all` and `Selection.clear` change it from the app's side.

```
g.picked.@each(; int row)
{
    play(g.songs[row]);
};
```

18.1.1 Clicks

Lists and tables share one click rule, the one native list boxes follow:

Click	Selection
Click	the row alone, and the anchor moves to it
Shift+click	the rows from the anchor to this one, in the order they show
Ctrl+click (⌘ on macOS)	toggles this row, which becomes the anchor
Double-click	selects the row and activates it
Right-click	selects the row alone first when it isn't selected; then its context menu opens

The range of a Shift+click runs in display order: in a sorted table it covers the rows between the two on screen, not between their indices in the app's array. With `selected:` only the plain click applies, and every click selects one row. With Mac keys (Context menus), Ctrl+click is not a toggle but a request for the context menu, as on macOS. A plain press on a row in a selection of several keeps the selection until the release. The release then selects the row alone, so a drag from it takes all the selected rows (Drag and drop inside the app). A right-click on a row that is already selected keeps the selection, so the menu acts on all the selected rows; on a row outside it, the row is selected first, as Finder, Explorer and GTK do, so the menu acts on the row the user pointed at.

18.2 Lists

A list is one focus stop holding a **cursor row**, as native list boxes are, not a focus group of rows (Focus). A long list builds only the rows in view, and a row that isn't built can't be a focus stop, so the list keeps the cursor itself and answers the keys while it has focus. When focus first arrives, the cursor goes to the first selected row, else the top; after that it keeps its row (Removing rows). It shows as a focus ring around its row while focus is visible.

18.2.1 Keys

Key	What the list does
Up, Down, the d-pad	moves the cursor, and selects the row it moves to
Home, End	moves to the first or the last row

Key	What the list does
Page Up, Page Down	moves by a view less one row in a long list, by ten rows in a plain list
Shift with a move	selects from the anchor to the cursor (several)
Ctrl with a move (⌘ on macOS)	moves the cursor alone, leaving the selection (several)
Ctrl+Space (⌘ on macOS)	toggles the cursor row (several); Space alone selects it
Ctrl+A (⌘A on macOS)	selects every row (several)
Enter, gamepad A	activates the cursor row, as a double-click does
Letters	move to the next row whose text starts with them, when the list has a `text` callback

The rows marked “several” apply to a list with a `selection`. Typed letters use type-ahead over rows that aren’t all built (`Ui.type_ahead_rows`, Type-ahead), searching in display order. When a key moves the cursor out of view, the list scrolls it in. Keys in a list explains how the list takes these keys without them reaching navigation.

`Ui.list_activated(id)` returns the row activated this frame by Enter, a double-click or gamepad A, and −1 in other frames; the app asks after the list. `Ui.list_cursor(id)` returns the cursor’s row.

18.2.2 Every row built

`Ui.@list(id, dividers, selected, selection, reorder)` builds every row its body makes with `Ui.@list_row(row)`. Rows are built in the order they show, and each names the app’s row it stands for. A row lays its body out in a row:

```
ui.@list("playlists", dividers: true, selected: &g.playlist)
{
    foreach (i, name : PLAYLISTS)
    {
        ui.@list_row((int)i)
        {
            ui.icon(MUSIC, 18, TEXT_SECONDARY);
            ui.text(name);
            ui.spacer();
            ui.text(string::tformat("%d songs", PLAYLIST_SIZES[i]), SM, color:
TEXT_SECONDARY);
        };
    }
};
int opened = ui.list_activated("playlists");
if (opened >= 0) open_playlist(g, opened);
```

`dividers` draws a hairline between rows, inset from the ends; without it, rows have rounded corners. A row tints under the pointer and fills with `ACCENT_SUBTLE` while selected, both fading on springs (Springs find their frame). Widgets inside a row take the pointer, so a row can hold a button or a checkbox, but they aren’t Tab stops of their own: the list is the one stop. A `Ui.@context_menu` declared after a `@list_row` block belongs to that row alone (Context menus); a menu every row shares is declared once with `Ui.@row_menu` after the list (One menu for every row).

18.2.3 Only the rows in view

`Ui.@long_list(id, count, row_height, height, dividers, selected, selection, text, data, reorder; int row)` is a list of `count` rows, all `row_height` points tall, in a scroll area of its own `height` tall. Its body builds the row it is given:

```
ui.@long_list("songs", SONG_COUNT, 36, height: ui::px(260), dividers: true, selection:
&g.picked,
    text: &song_title; int row)
{
    ui.text(string::tformat("%d", row + 1), SM, color: TEXT_SECONDARY, tabular: true);
    ui.text(TITLES[row % TITLES.len]);
    ui.drag_source(SONGS, &g.picked, TITLES[row % TITLES.len]);
};

ui.@row_menu("songs"; int row)
{
    if (ui.menu_item("Play", PLAY)) play_selected(g);
    ...
};
```

Since every row has the same height, the list knows from its scroll offset and its view's height which rows show, without laying any out. It builds those, plus one on each side so a row sliding in during a scroll is already there, between two spacers as tall as the rows above and below. The scroll area sees the full height, so the scrollbar and the wheel behave as for any content of that size (Scrolling), and a frame costs as many rows as are in view, whether the list holds a hundred rows or a million.

The list reads its keys before it picks its rows, so a cursor moved past the view is scrolled in and built in the same frame. Before its first layout, a list doesn't know its view's height: it takes its own `height` when that is fixed, else the window's, which builds too many rows for a frame but never too few. `text`, a `ListText` callback, returns a row's label for type-ahead; without it, typing doesn't move the cursor.

18.2.4 One menu for every row

A list, long list or table can declare one context menu for all its rows with `Ui.@row_menu(id; int row)`, called after it every frame in the same id scope. Its body is a menu's; it is given the app's row the menu was opened on, not the display position, so a sorted table still names the row the user meant:

```
ui.@long_list(...) { ... };

ui.@row_menu("songs"; int row)
{
    if (ui.menu_item("Play", PLAY)) play(g, row);
    if (ui.menu_item("Remove", TRASH_2)) remove(g, row);
};
```

The menu opens at the pointer on a secondary press, Ctrl+click with Mac keys (Context menus) or a long touch, and under the cursor row on the Menu key or Shift+F10. A right-click on a row outside the selection selects it first, as native list boxes do (Selection). The open state and the point belong to the list, not to the row: the menu stays where it opened when its row scrolls out of view, and its row stays the app's row when a table sorts.

A widget inside a row keeps its own context menu: a button or a link with a `Ui.@context_menu` wins over the row's, and a long touch on it opens its menu rather than its tooltip. One without a menu of its own falls back to the row's, so a row menu works over a cell's controls (Context menus).

Use `Ui.@row_menu` when every row shares a menu, and a `Ui.@context_menu` after a `@list_row` block when one row, or one kind of row, needs its own. A list should use one or the other, not both.

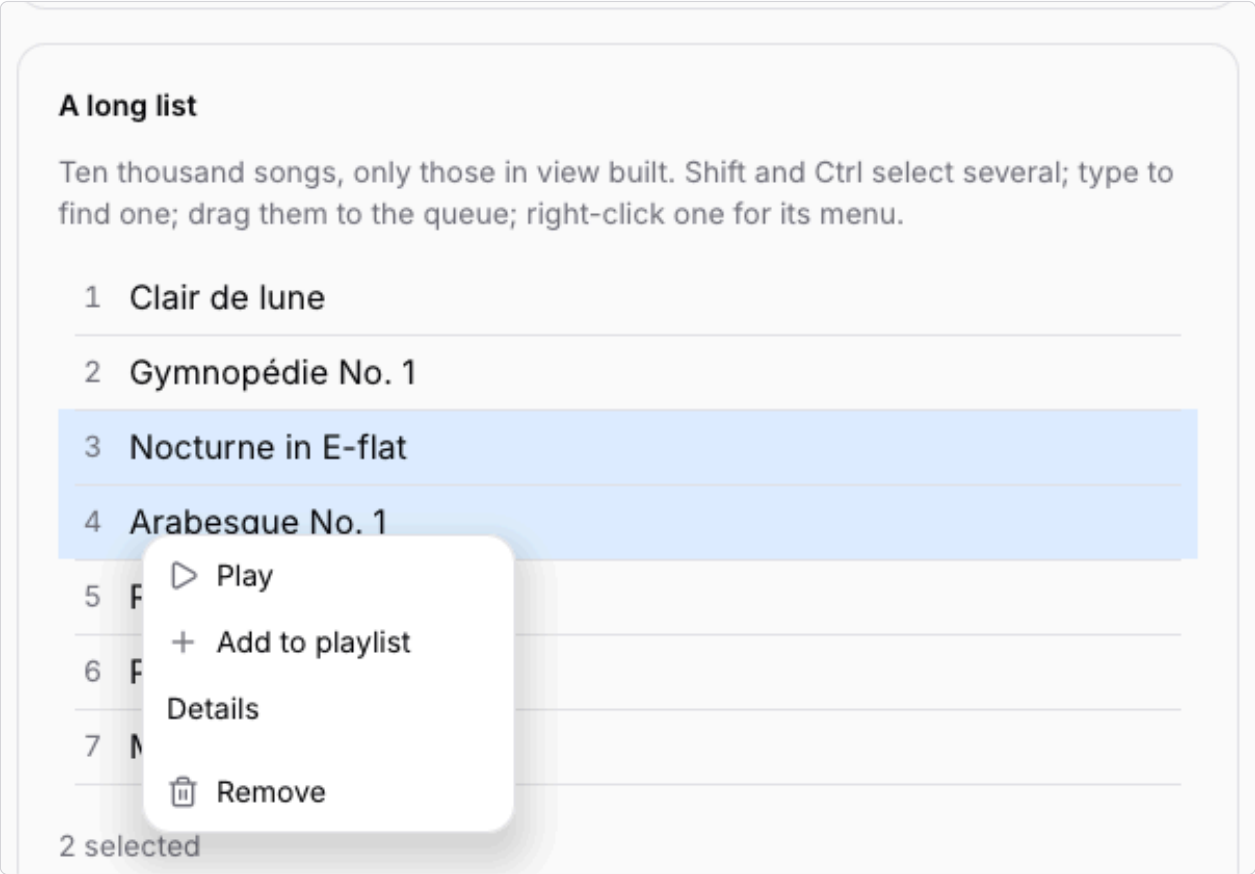


Figure 38: The long list with two rows selected and its row menu opened by a right-click on one of them. The selection stays, so Play, Add to playlist and Remove act on both rows; Details names the row the menu opened on.

18.3 Tables

A table is a long list with columns and a header. Its rows are one height, only those in view are built, and the cursor, keys, clicks and selection are a list's; the header adds sorting and resizing.

18.3.1 Columns

A `TableColumn` describes one column:

Field	Meaning
`title`	the header's text
`width`	a `Size`: points, a percentage, or `GROW` to share the room left; `FIT` is taken as `GROW`, since a column fitting each row's cell would not line up from row to row
`min_width`	the narrowest a drag makes it; 0 for 40 points
`align`	its cells' alignment: `START`, `CENTER`, or `END` for numbers, which `ui.table` draws with tabular figures
`unsortable`	its header doesn't sort, and is no focus stop

Field	Meaning
`compare`	a `TableCompare` of two rows; null for the default comparison of the cells' text

The cells' text comes from a `TableText` callback, `fn String(void* data, int row, int col)`, given the table's `data` pointer. The gallery's table of tracks:

```
const TableColumn[*] TRACK_COLUMNS = {
    { .title = "Title", .width = ui::GROW },
    { .title = "Artist", .width = { SizeKind.FIXED, 120, 120, 120 } },
    { .title = "Time", .width = { SizeKind.FIXED, 72, 72, 72 }, .align = END, .compare =
&by_time },
};

ui.table("tracks", &TRACK_COLUMNS, TABLE_ROWS, null, &track_text, selected: &g.track,
height: ui::px(300));
```

`Ui.table(id, columns, rows, data, text, selected, selection, sort, widths, height, row_height, reorder, search_column)` draws each cell's text on one line, cut short when it doesn't fit. `Ui.@table` takes the same arguments and a body that builds each cell, given the row and the column, for a cell with an icon, a badge or a control; `text` still gives each cell's text, for sorting and type-ahead. Type-ahead searches the `search_column`, such as the title when a " column comes first; without one, the sorted column, else the first. Rows are #api(ui::TABLE_ROW_HEIGHT") (36) points tall unless `row_height` says otherwise, under a header `ui::TABLE_HEADER_HEIGHT` (32) points tall. The `hello` example sizes a table to show all its rows without scrolling:

```
ui.table("recent", &RECENT_COLUMNS, RECENT.len, null, &recent_text, selected:
&demo.recent,
    height: ui::px(ui::TABLE_HEADER_HEIGHT + RECENT.len * ui::TABLE_ROW_HEIGHT));
```

The table doesn't scroll sideways: its columns share its width, and the `GROW` columns take what the others leave.

A table takes the same `Ui.@row_menu` after it. Its body's row is the app's row, even when the table is sorted, so the app acts on the right track:

```
ui.table("tracks", &TRACK_COLUMNS, TABLE_ROWS, null, &track_text, selected: &g.track,
height: ui::px(300));

ui.@row_menu("tracks"; int row)
{
    if (ui.menu_item("Play", PLAY)) play(g, row);
    if (ui.menu_item("Details")) show(track_text(null, row, 0), track_text(null, row,
1));
};
```

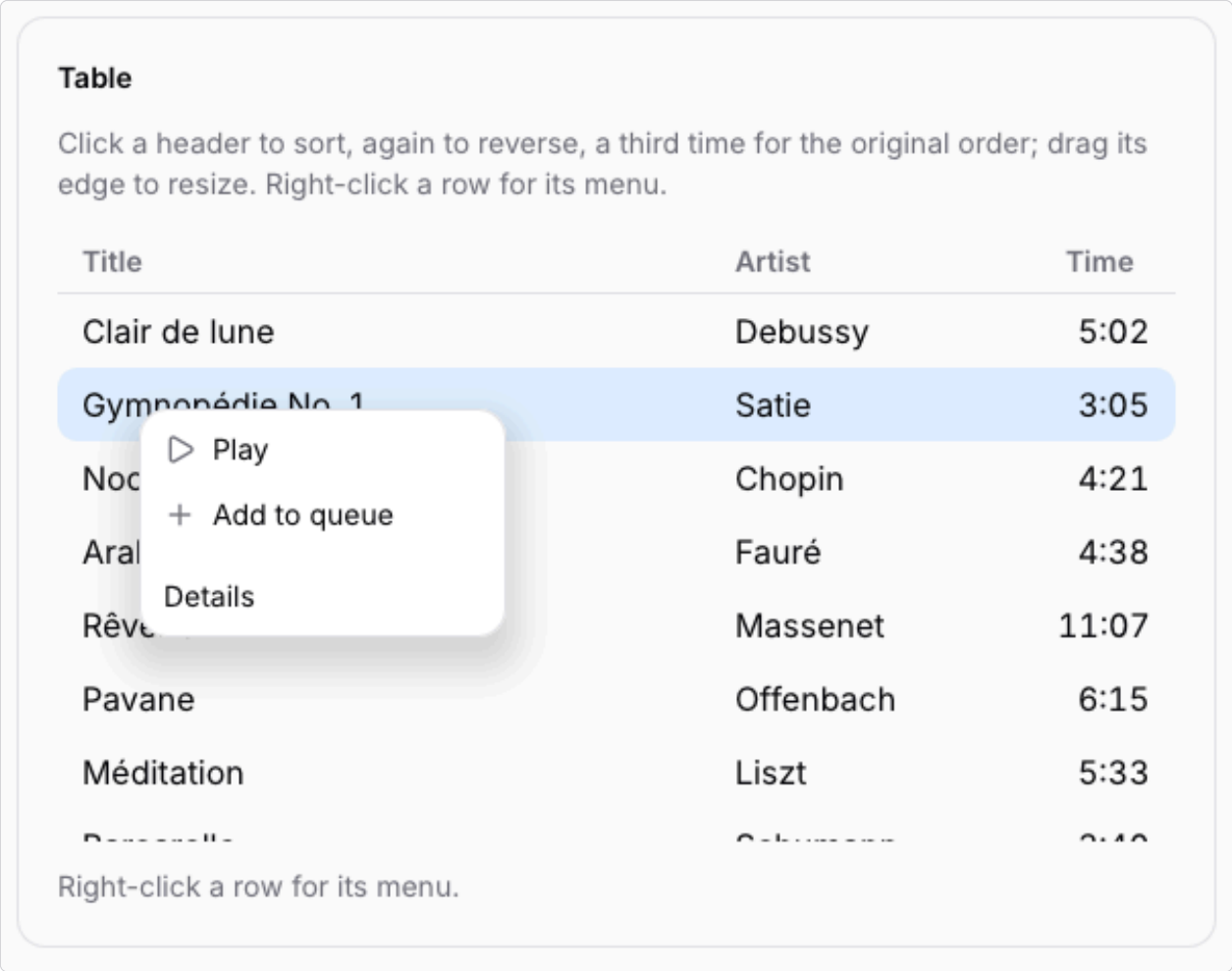


Figure 39: The table's row menu, opened on Gymnopédie No. 1. Its items act on the app's row, even after the table is sorted.

18.3.2 Sorting a view

Kiln sorts a view of the rows, not the rows. The table keeps an **order**, an array from display position to the app's row, and builds the rows in that order; the app's array never moves. The app keeps indexing its data by row, a selection kept by row survives every sort, and the app needs no code to sort. `Ui.table_row_at(id, pos)` returns the row shown at a display position, for an app that wants to act on rows in the order the user sees them.

A click on a sortable column's header, or Enter, Space or gamepad A on it, cycles its sort: ascending, then descending, then back to the app's own order. The sorted column's title turns `TEXT`, with a chevron pointing up for ascending and down for descending. The headers form a horizontal focus group (Focus), a focus stop before the rows, so Tab reaches them and Left and Right move between them.

A `TableSort` is the sort: whether the table is sorted, by which column, and whether descending. Its zero value is the app's order. The table keeps its own unless the app passes `sort:`, a `TableSort` of its own that the header then changes, to save it or to start sorted; `Ui.table_sort(id)` reads the table's own.

The sort is a stable merge sort, so rows that tie keep the order they had: sorting by artist, then by title, gives titles in order with each title's artists still in order. The table sorts again only when the sort changes, when the number of rows changes, or after `Ui.table_changed(id)`, which an app calls

when it edits rows in place. Other frames reuse the order, so a steady frame neither sorts nor allocates; a sort takes its scratch array from the frame arena (Allocators).

18.3.3 Comparing cells

Without a column's `compare`, `Ui.compare_cells` orders two cells by their text. When both read as numbers, with an optional sign and a fraction after `.` or the user's decimal separator (The decimal separator), they compare as numbers, so 9 comes before 10 and 2,5 before 10 in German. A number goes before text. Two texts compare in the `Ui`'s text order, with `Ui.compare_text`.

A `TextOrder` is one of two:

- `NATURAL`, Kiln's natural order (`text::natural_compare`), as file managers sort names: runs of digits compare as numbers, so "track 2" goes before "track 10"; letters compare with case folded (ASCII, Latin-1, Greek and Cyrillic); anything else by code point. Texts equal under those rules compare by their bytes, so the order is total and the same on every machine.
- `SYSTEM`, the OS's order for the user's language, through `os::collate`: `CompareStringEx` on Windows and `localizedStandardCompare` on macOS, the orders Explorer and Finder sort by, which know that "é" sorts with "e" in French and after "z" in Swedish. Linux has no such service, so there the order is `NATURAL`.

While the theme follows the system (`themes::SYSTEM`, Following the system), the text order is `SYSTEM` where the OS has one. `Ui.set_text_order` pins either, until `Ui.follow_os` hands it back (Overriding a setting). A `Ui` made by `ui::create`, as tests use, starts in `NATURAL`, so a test's sorted rows don't depend on the machine.

A column whose text doesn't sort as text brings its own comparison. The gallery's Time column holds "m:ss", which as text would put "11:07" before "2:58":

```
fn int by_time(void* data, int a, int b, int col) => seconds_of(TIMES[a % TIMES.len]) -
seconds_of(TIMES[b % TIMES.len]);
```

18.3.4 Resizing columns

Each header's end edge is a grip `ui::TABLE_RESIZE_GRIP` (4) points to either side, showing the resize cursor under the pointer. Dragging it sets the column to a fixed width, the header's width when the drag began plus how far the pointer moved, never narrower than its `min_width`. The `GROW` columns take whatever room is left, so widening one column narrows them. A double-click on the grip puts the column's own width back.

The table keeps the dragged widths in its state. An app that wants them, to save them with the window's layout, passes `widths:`, an array of `float` with one per column, 0 for the column's own width; the drag then writes there, and the app's values are the widths the table uses.

18.4 Reordering rows

A list, a long list or a table given `reorder: true` lets the user put its rows in a new order: by dragging a row, or the selection when the row grabbed is in it, or with Alt+Up and Alt+Down on the cursor row. While a row is dragged over its own list, a line in the accent color shows where it would land, and the list scrolls when the pointer nears its top or bottom edge, so a row can travel the length of a long list (Drag and drop inside the app).

The list doesn't move anything. On the frame of the move, `Ui.list_reordered` reports a `Reorder`: the rows moved, in their order, and the row they now go before, or −1 for the end. The app moves its own data, with `Reorder.apply` for a plain slice:

```
ui.@list("playlists", selected: &g.playlist, reorder: true)
{
    foreach (i, p : g.order) ui.@list_row((int)i) { ui.text(PLAYLISTS[p]); };
};
if (try moved = ui.list_reordered("playlists")) moved.apply(g.order[..]);
```

An app whose rows live on a server, such as a playlist on a streaming service, sends the move instead: `rows` and `before` are what a "move these rows before that one" call takes, one call per run of adjacent rows. Rows are known by their index, so on the frame it reports a move the list also moves its cursor and the app's `selected` or `Selection` to where the rows go, as if the app had applied it; `Reorder.new_index` says where any row goes. A drop where the rows already are reports nothing.

A sorted table doesn't reorder: a move only means something in the data's own order, so while a sort is on, rows show no line, Alt+Up and Alt+Down do nothing, and the drag ends without a move.

18.5 Removing rows

Rows are known by their index, so when the app removes some, the list's cursor and the app's `Selection` still hold the old indices: the cursor lands on another item, and the selection marks rows nobody chose. The app says which rows went, as it deleted them:

```
if (ui.shortcut(DELETE, local: true))
{
    int[] gone = picked_rows(&picked);   // the app's own rows, as they were
    remove_songs(gone);
    ui.list_removed("songs", gone);
    picked.remove(gone);
}
```

`Ui.list_removed(id, rows)` keeps the cursor on its item, which moves up by the removed rows before it. When the cursor's own row went, it takes the row after the removed ones, so a keyboard user who deleted a song is on the next one; in a sorted table, where the next row in the data isn't the next one shown, it takes the row now shown in its place. `Selection.remove(rows)` takes the rows out of the app's selection and moves the rest up, the anchor with them. A row removed without `list_removed` still leaves the cursor in its place: when its row is gone, the cursor takes the row at the same position, else the last, rather than jumping to the top.

18.6 Grids

`Ui.@grid(id, cols, min_cell, gap, gap_x, gap_y, dense, stretch)` lays out the cells its body makes with `Ui.@cell(span, start, rows)` in columns that wrap onto new rows. It follows Tailwind's grid, so a Tailwind layout reads across directly:

Tailwind	Kiln
`grid grid-cols-3`	`ui.@grid(cols: 3)`
`gap-4`, `gap-x-2`, `gap-y-6`	`gap: 4`, `gap_x: 2`, `gap_y: 6`, in spacing steps as for boxes
`grid-cols-1 md:grid-cols-3`	`cols: ui.at(MD, 3, 1)` by the window, or `cols: ui.width_at(600, 3, 1)` by the container (Responsive layout)

Tailwind	Kiln
`grid-cols-[repeat(auto-fill,minmax(260px,1fr))]`	`min_cell: 260`: as many columns as fit cells that wide
`col-span-2`	`ui.@cell(span: 2)`
`col-start-3`	`ui.@cell(start: 3)`, counting from 1
`row-span-2`	`ui.@cell(rows: 2)`
`grid-flow-row-dense`	`dense: true`
`items-stretch`, `items-start`	`stretch: true` (the default), `stretch: false`

The `hello` example lays its cards out in as many columns as fit 260 points, each card in a cell and as tall as its row:

```
ui.@grid("top cards", min_cell: 260, gap: 4)
{
    top_cards(ui, demo);
};

fn void top_cards(Ui* ui, Demo* demo)
{
    ...
    ui.@cell()
    {
        ui.@card(width: ui::GROW, height: ui::GROW, pad: 5, gap: 3, shadow: MD)
        {
            ui.heading("Wrapping", 3);
            ...
        };
    };
    ...
}
```

The gallery's grid mixes spans and dense filling:

```
ui.@grid("tiles", min_cell: 120, gap: 3, dense: true)
{
    ui.@cell(span: 2) { tile(ui, "Featured", "Two columns wide", ACCENT_SUBTLE); };
    ui.@cell(rows: 2) { tile(ui, "Tall", "Two rows tall", SURFACE_RAISED); };
    for (int i = 0; i < 7; i++)
    {
        ui.@id(i)
        {
            ui.@cell() { tile(ui, ARTISTS[i], "Artist", TILE_COLORS[i %
TILE_COLORS.len]); };
        };
    }
};
```

Every item of a grid goes in a `ui.@cell`, which places it; anything built directly in the grid's body would land outside the grid's rows. A grid has at most `ui::GRID_MAX_COLUMNS` (64) columns, and `Ui.grid_columns(id)` reports how many it had when last built.

18.6.1 Placing cells

Clay lays children out in one row or one column and never wraps (Layout), so the grid places cells itself, with CSS's auto-placement. It keeps an occupancy map, a bit per column for each grid row, in the frame arena. Each cell takes the first place after the previous cell where its `span` columns are free for its `rows` rows; with `start`, it goes at that column, on a later row when needed. With `dense`, every cell searches from the top instead, so a later, smaller cell fills a gap an earlier, wider one left,.

Each grid row is a Clay row, `gap_x` between its cells, and the rows stack `gap_y` apart. A cell's width comes from the grid's width: the grid reads its own width from last frame, as a container query does (Container queries), and divides it among the columns and gaps. After a resize the columns lag one frame, and in the grid's first frame the width is that of the container it is in. A cell isn't fixed at that width but grows up to it, so a width guessed wrong, too wide by a frame, never holds the grid wider than its parent. With `stretch`, the cells in a row are as tall as the tallest.

18.6.2 Cells that float

A cell spanning several rows can't be a child of one Clay row. The grid keeps an empty slot for it in its first row and attaches the cell to that slot as a Clay floating element, as tall as the rows it spans and the gaps between them, read from last frame. A cell that dense filling places in a gap of a row already built floats the same way, onto the free slot the grid left there. The rows below see the slots and lay out around them.

A floating cell's content can be taller than the rows it spans. Its content is measured, and the last row it spans is made that much taller from the next frame, as CSS grows the rows a spanning item overflows. The row keeps that room while the content needs it, and gives it back when the cell goes or fits again.

19 Status

An app tells the user how things stand: a download is halfway, three messages wait, a sync failed, plays rose this week. Kiln has four kinds of widget for this. A progress bar shows how far a task has come; a badge marks something with a word, a count or a dot; an alert puts a message in the page; a metric shows a key number and how it changed. They share one vocabulary of colors, the tone, and they react the same way to a user who asked the OS for more contrast. None of them takes focus of its own: they report, and the controls around them act.

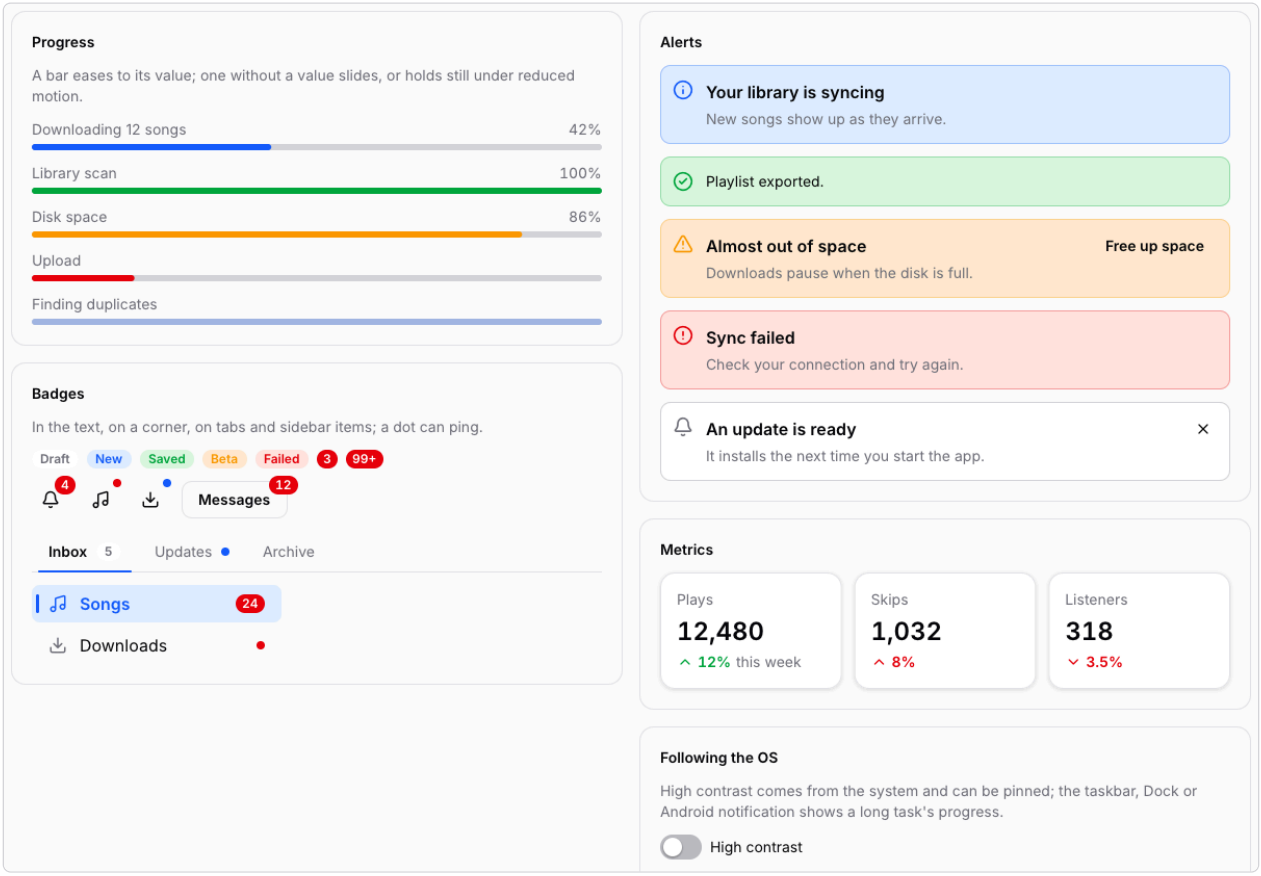


Figure 40: The gallery's status page in the light theme. On the left, progress bars in four tones and an indeterminate one, and badges: pills in each tone, counts, badges on the corners of icon buttons and a button, on tabs and on sidebar items. On the right, an alert in each tone, the last one with a ; three metrics in cards, the Skips change red because fewer skips are better; and the switch that pins high contrast.

19.1 Tones

A Tone says what a status widget means: `NEUTRAL`, `INFO`, `SUCCESS`, `WARNING` or `DANGER`. Each tone stands for two color tokens and an icon, so a widget asks its tone rather than naming colors:

Tone	Strong	Subtle	Icon
`NEUTRAL`	`TEXT_SECONDARY`	`SURFACE_RAISED`	`BELL`
`INFO`	`ACCENT`	`ACCENT_SUBTLE`	`INFO`
`SUCCESS`	`SUCCESS`	`SUCCESS_SUBTLE`	`CIRCLE_CHECK`
`WARNING`	`WARNING`	`WARNING_SUBTLE`	`TRIANGLE_ALERT`
`DANGER`	`DANGER`	`DANGER_SUBTLE`	`CIRCLE_ALERT`

`Tone.strong` is the color of an icon, a fill or a border, and of text set on the tint; `Tone.subtle` is the soft background; `Tone.icon` is the icon an alert shows. `INFO` borrows the accent, so information takes the app's own color. The three status tints are derived from their status colors as `ACCENT_SUBTLE` is from the accent (Status tints): a theme that changes `DANGER` gets a matching `DANGER_SUBTLE` without naming it, and a scoped `push_token(DANGER, …)` tints the alerts inside the scope to match.

19.2 Progress bars

A progress bar is a rounded track `ui::PROGRESS_HEIGHT` (6) points tall, in the `TRACK` token, with a fill in the tone's strong color (`TEXT_SECONDARY` for `NEUTRAL`). Neither kind of bar takes the pointer or focus: a bar reports work, and the button that cancels it is a widget of its own.

19.2.1 A known amount

`Ui.progress(label, value, tone, width, percent)` fills the bar to `value`, from 0 to 1, clamped. The label sits above the track in small secondary text, with the value as a percentage at its end unless `percent` is false. The label is also the bar's id, so a `##` suffix (The core calls) tells apart two bars with the same text, and a bar with no visible label still has a name:

```
ui.progress("Downloading 12 songs", 0.42f);
ui.progress("Library scan##done", 1.0f, SUCCESS);
ui.progress("Upload##failed", 0.18f, DANGER, percent: false);
```

Work reports progress in jumps: a job finishes a file, a chunk arrives. The fill doesn't jump with it. It is a spring (Springs find their frame) moving with the `STATE` motion token (Motion tokens), the one a switch uses, so each new value is reached with a short ease, and under reduced motion at once (Reduced motion). In the bar's first frame the spring starts at the value, so a bar that appears halfway doesn't sweep up from zero. The percentage shows the value itself, not the easing fill, so the number is always the true one. It is rounded to a whole percent and written with `Ui.format_number`, which uses the user's decimal separator (The decimal separator), in tabular figures so it doesn't shake as it counts. The fill never draws shorter than the track is tall, so even 1% shows as a round dot rather than a sliver.

19.2.2 An unknown amount

`Ui.progress_indeterminate(label, tone, width)` is for work whose end isn't known. A segment two fifths of the track wide slides across it, entering at the start and leaving past the end, with an ease in and out on each pass. Its phase comes from `Ui.loop_phase` with the `SLIDE` loop, 1.4 seconds a pass (Looping motion), so the bar needs no state, keeps frames coming only while it is built, and every indeterminate bar on screen moves in step. There is no percentage.

A sliding segment frozen at the start of its pass would read as a stuck bar at 0%, so under reduced motion the segment is dropped: the whole track holds still in a quarter-strength wash of the fill color, as a skeleton holds halfway into its breath (Loading looks). The figure's "Finding duplicates" bar is drawn so, since screenshots are taken with reduced motion.

19.2.3 Driving a bar from a job

A bar usually follows a background job (Background work). The job calls `Job.set_progress` as it goes, and the frame reads the value from the job's `JobState` and hands it to the bar; the bar's spring smooths the steps. The gallery's Background page counts primes this way:

```
JobState s = ui.job("primes", &count_primes, &g.primes_below, key: g.primes_run);
switch (s.status)
```

```
{
    case PENDING:
        ui.@row(width: ui::GROW, gap: 3, align: END)
        {
            ui.progress("Counting…", s.progress);
            if (ui.button("Cancel", GHOST, SM)) ...
        };
    ...
}
```

A bar is only content, so it disappears when the app stops building it: the `hello` example shows “Saving…” while a save runs and builds nothing once it is done.

19.2.4 On the taskbar or Dock

A long task is worth seeing from another app too. `Ui.set_progress(value)` puts the progress on the app’s taskbar button or Dock icon: a value from 0 to 1, `os::PROGRESS_UNKNOWN` for work without a known end, or `os::PROGRESS_NONE` to take it away when the work is done. The app calls it every frame with the current value, the same one its bar shows; the `Ui` rounds it to a percent and tells the OS only when that changes, and does nothing without a window, so tests run unchanged. How each OS shows it is in Progress on the taskbar or Dock.

19.3 Badges

A badge marks something without taking room or attention from it: a word on a row, a count on a button, a dot on an icon. Badges take no pointer and no focus.

19.3.1 In the text

`Ui.badge(text, tone)` is a pill `ui::BADGE_HEIGHT` (18) points tall, in the tone’s tint, with its text in extra small semibold in the tone’s strong color (`TEXT_SECONDARY` for `NEUTRAL`, the default). It sits in the flow like any text: on a list row, after a heading, in a table cell.

`Ui.badge_count(n, tone)` is a count in a solid pill of the tone’s strong color, `DANGER` by default, white text on top whatever the theme, as on a phone’s app icon. It is never narrower than it is tall, so a single digit is a circle. Past 99 it reads “99+”: the exact number of unread messages past a hundred tells the user nothing more, and a wide pill would cover what it marks. At 0 or less it draws nothing, so the app passes its count without a test. `Ui.count_text` gives the same text for widgets that place a count themselves.

`Ui.badge_dot(tone, ping)` is a dot `ui::BADGE_DOT` (8) points across, for “something is new here” when the number doesn’t matter. With `ping`, a ring grows from the dot to twice its size and fades, over and over, as Tailwind’s `animate-ping` does, on the `PING` loop, 1.5 seconds a ring (Looping motion). A ring frozen in place would only look like a bigger dot, so under reduced motion the ring is dropped and the dot stays.

19.3.2 On a corner

`Ui.@badged(count, dot, ping, tone; body)` builds its body, any widget, and floats a badge on the body’s top end corner: a count when `count` is above 0, else a dot when `dot` is set. With neither, the body shows alone, so the badge comes and goes with the app’s data and the body never moves:

```
ui.@badged(dot: true, ping: true, tone: INFO) { ui.icon_button(DOWNLOAD,
"downloading"); };
ui.@badged(count: g.unread) { ui.button("Messages", OUTLINE); };
```

The badge is a Clay floating element attached to the box around the body, its center on the box's top end corner and nudged a little inside (4 points for a count, 2 for a dot), one z-index above the page or the layer it is built in (Z-order). Clay lays a floating element out apart from the flow, so the body keeps its own size and the row it sits in doesn't shift when a count appears. The badge declares no widget id, so hit testing never finds it: a click on the badge reaches the button under it, and focus, the focus ring and the tooltip stay the body's.

Three widgets take a badge directly, each as an `int`: above 0 a count, 0 a dot, below 0 none, the default.

Widget	Where the badge goes
`Ui.icon_button(…, badge)`	on the button's corner, through `ui.@badged`
`Ui.sidebar_item(…, badge)`	a count or a dot at the row's end; in icons mode, where there is no room for a count, a dot on the icon
`Ui.tabs(…, badges)`	one per label: a neutral pill with the count after the label, or an `INFO` dot

A tab's count is a quiet `NEUTRAL` pill rather than a red one: on a tab it says how many items the page holds, not that something needs attention.

19.3.3 A mark

`Ui.@marked(icon, color, label; body)` builds its body, any widget, and floats a status mark on the body's top start corner: a `ui::MARK_SIZE` (28) point `SURFACE_RAISED` circle, `ui::MARK_INSET` (8) points inside, with `icon` in `color` 15 points across. It is for a cover that needs tags or has a match ready, where a second line of text would push the album's own title and artist aside:

```
ui.@marked(TAG, WARNING, "Needs tags") { ui.image(cover, title, width: ui::GROW, aspect:
1); };
```

With `icon` as `NONE` the body shows alone, so the mark comes and goes with the app's data and the widget never moves. Clay lays the mark out apart from the flow, one z-index above the page or the layer, so it draws over the cover. The body keeps its size, pointer and focus. `label` names the mark's widget id; Kiln has no accessible-name API, so it is not a screen-reader name.

19.4 Alerts

`Ui.@alert(id, tone, title, open, action; body)` puts a message in the page, in the flow, as wide as its container. It is a box in the tone's tint with a 1-point border in the tone's strong color at 30% opacity, the tone's icon at the start, the `title` in semibold when there is one, and the body below it. The body builds the message, so it may be wrapping text, a link, or any widgets. The macro returns true in the frame the `action` button is pressed:

```
if (ui.@alert("warning", WARNING, "Almost out of space", action: "Free up space")
{
    caption(ui, "Downloads pause when the disk is full.");
}) free_up_space(g);
```

With `action`, a small ghost button with that text sits at the alert's end. With `open`, a pointer to the app's `bool`, a button follows it; pressing it sets `*open` to false. The app owns the flag, so it decides whether a dismissed alert comes back, and setting the flag true again shows it.

When `*open` turns false, the alert slides shut: its height follows a spring on the `DISCLOSURE` motion token (Motion tokens), the one a disclosure opens with (Disclosures), applied to the alert's height measured last frame, with the content clipped as it shrinks. Under reduced motion it closes at once. Once the spring reaches zero, the alert isn't built at all, so a closed alert costs nothing and leaves no gap. Opening again slides it open the same way.

An alert isn't a focus stop itself; the message is read, not operated. Its action and its are ordinary buttons, Tab stops in the page's order. While an alert closes, its buttons are inert, so Tab can't land on a that is sliding away.

19.5 Metrics

`Ui.metric(label, value, change, up_is_good, detail)` shows a key number as dashboards do: the `label` in small secondary text, the `value` large and semibold under it, and a line with the change and a `detail`.

```
ui.metric("Plays", "12,480", 0.12f, detail: "this week");
ui.metric("Skips", "1,032", 0.08f, up_is_good: false);
```

`value` is a string the app has already formatted, since only the app knows whether it is a count, a duration or an amount of money, and in which units. It is drawn in tabular figures, so metrics side by side line their digits up. `change` is a fraction, 0.12 for 12%, shown as a chevron up or down and the percentage without its sign, whole from 10% on and with one decimal below, with the user's decimal separator. NaN, the default, shows no change.

The change's color says whether it is good news, not which way it went. `ui::metric_tone(change, up_is_good)` decides: `SUCCESS` when the change goes the good way, `DANGER` when it doesn't, `TEXT_SECONDARY` with no chevron when it is zero. `up_is_good` is true by default; for costs, errors or skips it is false, so a rise shows red. In the figure, Plays and Skips both rose, and only Plays is green.

A metric is plain content with no box of its own. Put each one in a card, and lay the cards out in a `Ui.@grid` (Grids) so they wrap on a narrow window:

```
ui.@grid("metrics", min_cell: 150, gap: 3)
{
    ui.@cell() { ui.@card(width: ui::GROW, height: ui::GROW, pad: 4)
{ ui.metric("Plays", "12,480", 0.12f, detail: "this week"); }; };
    ...
};
```

19.6 High contrast

A soft tint behind colored text is exactly what a user who asked for more contrast finds hard to read, and a faint border around a tinted box all but vanishes. `ui.high_contrast` holds that request; the status widgets read it while they are built and draw edges that don't depend on the tint:

Widget	Under high contrast
`ui.badge`	the pill is outlined in the tone's strong color, and its text is `TEXT`
`ui.badge_count`	the solid pill is outlined in `SURFACE`, setting it off from what it covers
`ui.badge_dot`	the dot is ringed in `SURFACE`, 1.5 points wide

Widget	Under high contrast
`ui.@alert`	the border is the tone's strong color at full strength
`ui.progress`, `ui.progress_indeterminate`	the track is outlined in `TEXT_SECONDARY`, so its end shows however little is filled

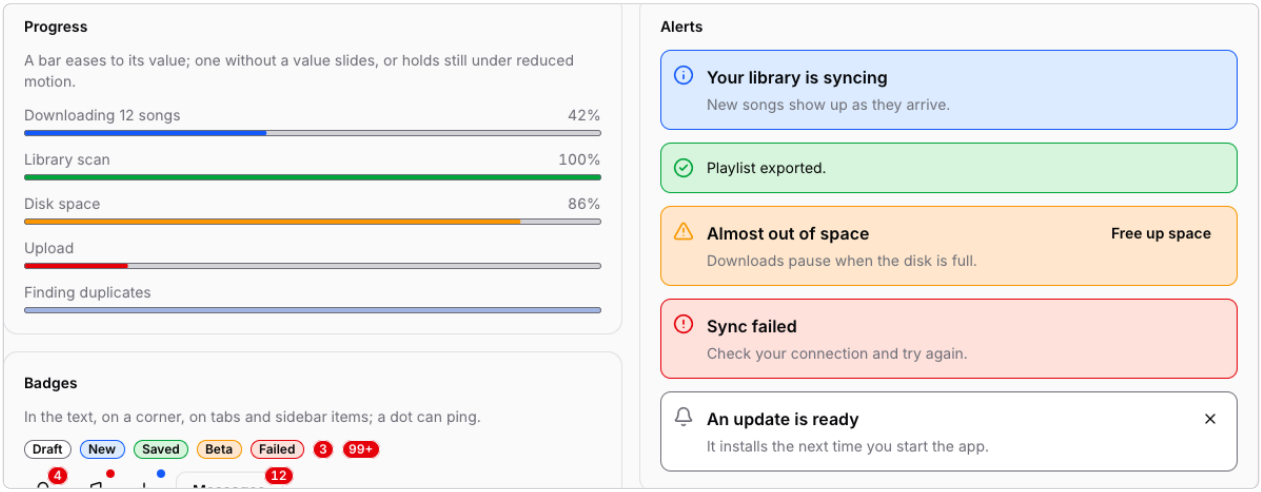


Figure 41: The same page under high contrast. The pills in the text are outlined in their tone's color with dark text, the alerts have full-color borders, and every progress track has an outline that marks where it ends.

The flag follows the OS's high-contrast setting while the theme follows the system (Following the system). `Ui.set_high_contrast` pins it on or off whatever the OS says, and `ui.follow_os(HIGH_CONTRAST)` lets go again (Overriding a setting); tests keep it off unless they pin it, as the figure's screenshot does. A custom widget that relies on a tint can read `ui.high_contrast` the same way.

20 Images

Apps show pictures: album covers, avatars, thumbnails, photos in a list. In Kiln an image is made from RGBA pixels, which the app provides or Kiln decodes from a JPEG or PNG file in the background. Kiln keeps a copy, and draws it in a box that takes Tailwind's image options: how the picture fills the box, where it sits, the box's shape, rounded corners, opacity and a shadow.

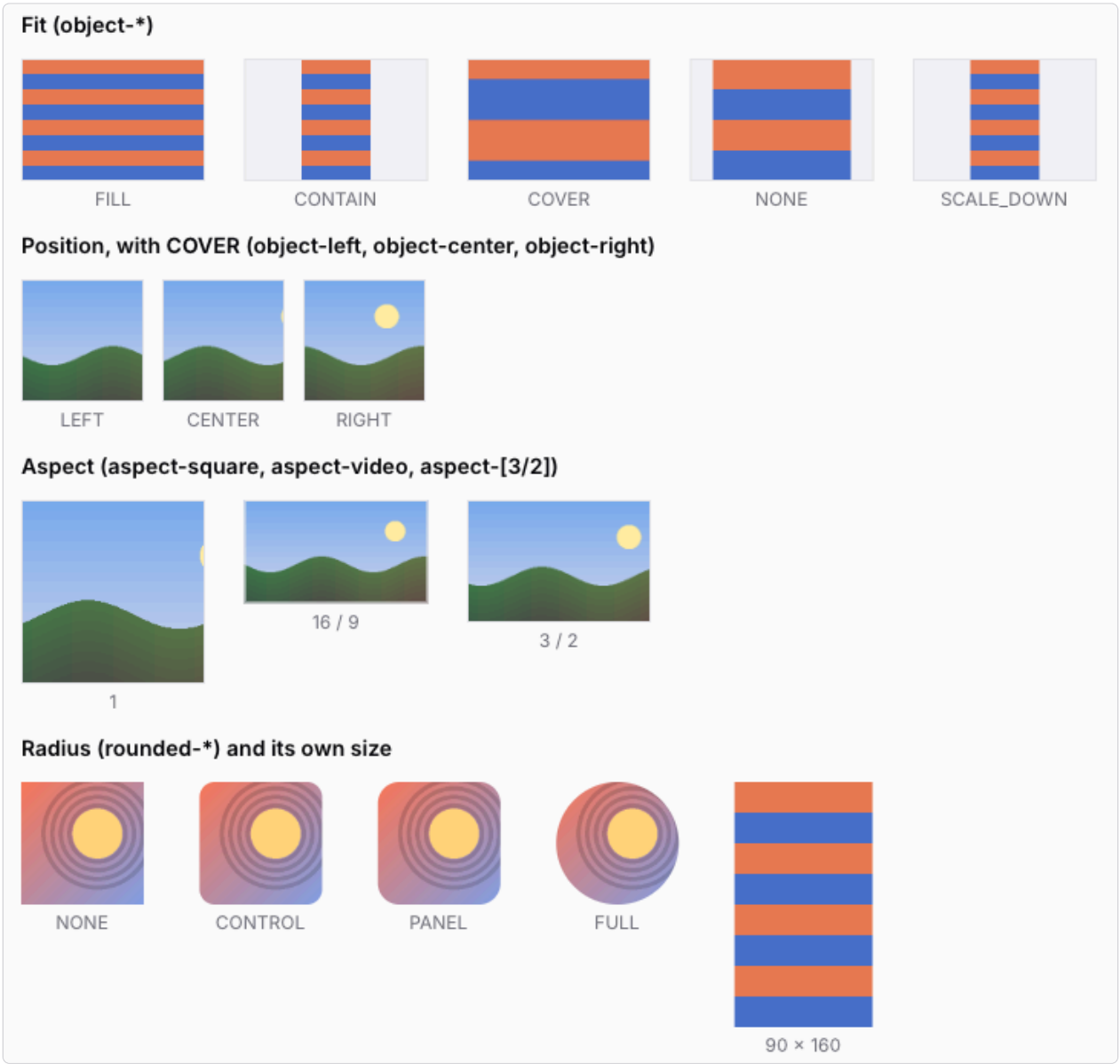


Figure 42: The gallery's images page in the light theme. The first row draws one tall striped picture in a wide box with each fit: `FILL` stretches the stripes, `CONTAIN` and `SCALE_DOWN` show all of it with empty sides, `COVER` fills the box and cuts the top and bottom, `NONE` shows the middle at its own size. The second row covers a square box with a wide landscape and keeps its left, middle or right: the sun is only in the last. The third sets the box's shape with `aspect`. The last row rounds an album cover with `NONE`, `CONTROL`, `PANEL` and `FULL`, the last making a circle, and draws the striped picture at its own size.

20.1 Making an image

`Ui.image_create(width, height, rgba)` makes an image from `width` × `height` pixels, four bytes each, row by row from the top: RGBA8 in sRGB with straight alpha, as decoders give them. The

pixels are copied, so the app can free its buffer at once. The result is an `ImageId`, a handle the app keeps in its state; `ui::NO_IMAGE` (0) is none.

```
ImageId cover = ui.image_create(w, h, rgba);   // rgba: w * h * 4 bytes
```

`Ui.image_update(image, rgba)` replaces the pixels with new ones of the same size, as for a picture that changes, and `Ui.image_destroy(image)` frees an image, its pixels and its textures; a freed handle draws nothing, and its slot is reused by the next image made. `Ui.image_size` gives the size in pixels. The `Ui` owns every image, and `Ui.destroy` frees those still alive, so an app that keeps its pictures for its whole run never frees them itself.

20.2 Loading a picture

`Ui.image_load(bytes, max_size)` makes an image from the bytes of a JPEG or PNG file, such as a cover downloaded from a music service; `Ui.image_load_file(path, max_size)` reads the file too. Both return the image's handle at once and decode in the background, since decoding a cover takes milliseconds the frame shouldn't wait for. The bytes are copied, so a download's buffer can be freed straight after.

```
ImageId cover = ui.image_load(response.body, max_size: 256);
...
ui.image(cover, album.title, fit: COVER, aspect: 1, radius: PANEL, width: ui::px(64));
```

`Ui.image_state` tells where the image stands (`ImageState`): `LOADING` while it decodes, `READY` from the frame after it is decoded, and `FAILED` when the data wasn't a JPEG or PNG Kiln could read, the file couldn't be read, or the picture is too large. While it loads, `ui.image` keeps its box and fills it with the loading skeleton's color (Loading looks), so a list of covers keeps its layout as they arrive; a failed image draws nothing, and the app can check the state to draw a placeholder of its own. The size is zero until the picture is decoded, so a box for a loading image takes a size or an `aspect`, as the cover above does. Destroying an image while it loads cancels its decode.

Decoding runs as a background job (Background work) on a worker thread, and a finished one wakes an App-style loop that is sleeping, as any job does. In the browser, which decodes pictures only asynchronously, the browser's own decoder does it (`createImageBitmap`), and Kiln asks once a frame whether it is done.

`max_size` keeps no more pixels than the app can show: the decoded picture is halved, each pixel the average of four as for the smaller levels below, until neither side is more than twice `max_size`. A list of album covers drawn at 64 points needs no 640-pixel originals; at `max_size: 128`, a 640-pixel cover keeps 160 × 160 pixels, and a thousand of them about 100 MB instead of 1.6 GB.

`os::decode_image(bytes, max_size, allocator)` and `os::decode_image_file` are the decoder itself, for an app that wants the `Pixels` rather than an image, such as to pick a cover's main color. They are synchronous and touch nothing shared, so an app can call them from its own jobs. They read baseline and progressive JPEG (grayscale, color or CMYK) and PNG (any bit depth, palettes, transparency, interlacing), and give RGBA8 in sRGB with straight alpha. A picture more than `os::MAX_IMAGE_SIDE` pixels on a side fails with `os::IMAGE_TOO_LARGE`, found from its header before any pixels are allocated, so a damaged or hostile file can't ask for gigabytes; anything else unreadable fails with `os::NOT_AN_IMAGE`. In the browser they fail with `os::UNSUPPORTED`, and only `Ui.image_load` decodes there.

The decoder outside the browser is stb_image, built into Kiln's C shim with only its JPEG and PNG readers. It costs an app about 45 KB, and only an app that loads a picture: an app that never calls `Ui.image_load` or `os::decode_image` links none of it. Each OS's own decoder (WIC on Windows,

ImageIO on macOS and iOS, `BitmapFactory` on Android) is to take its place there, as the roadmap says.

20.3 Drawing an image

`Ui.image(image, label, fit, position, aspect, width, height, radius, opacity, shadow)` draws an image in a box laid out like any other. An album cover, square, rounded like a card and 240 points wide:

```
ui.image(cover, "Album cover", fit: COVER, aspect: 1, radius: PANEL, width:
ui::px(240));
```

The box's size follows the same rules as an `<img>` in CSS:

- With no size, the box is the picture's own size, a pixel to a point.
- With one size, the width or the height, the other follows the picture's shape, so a 1600 × 900 photo given a width of 320 is 180 tall.
- `aspect` is the box's width divided by its height, as Tailwind's `aspect-*`. With one size it sets the other: a square box from `aspect: 1`, whatever the picture's shape. The picture then fills the box by `fit`.

Each option has its Tailwind or HTML counterpart:

Tailwind / CSS / HTML	Kiln
`object-fill`, `object-contain`, `object-cover`, `object-none`, `object-scale-down`	`fit: ImageFit` `FILL` (the default), `CONTAIN`, `COVER`, `NONE`, `SCALE_DOWN`
`object-center`, `object-top`, `object-left-bottom`, …	`position: ImagePosition` `CENTER` (the default), `TOP`, `LEFT_BOTTOM`, …
`aspect-square`, `aspect-video`, `aspect-[3/2]`	`aspect: 1`, `16.0f / 9`, `1.5f`
`w-*`, `h-*`	`width:`, `height:` sizes, such as `ui::px(240)` or `ui::GROW`
`rounded-*`	`radius: CONTROL`, `PANEL`, `SURFACE` or `FULL`
`opacity-*`	`opacity:` from 0 to 1
`shadow-*`	`shadow: SM`, `MD` or `LG`
`alt`	`label`

`position` places a picture that doesn't fill its box (`CONTAIN`, `NONE`, `SCALE_DOWN`), and for `COVER` it picks the part that shows: `LEFT` keeps the left of a wide picture in a square box. `ui::image_fit` does the

arithmetic, from the box, the picture's size, the fit and the position to the rectangle drawn and the part of the picture shown in it, as CSS defines them.

The `label` names the image, as HTML's `alt` does. It is never shown: it is the image's id, so a `##` suffix or a `ui.@id` scope (The core calls) tells apart two images with the same label, and it is what assistive technology reads for the picture. "Album cover" says less than the album's title; a picture that only decorates still has a label, since the id needs one.

20.4 How it's drawn

An image is one quad in the draw list, textured with the picture, with the same rounded-rectangle distance the shader computes for any box (One shape, one quad). Its corners are rounded and antialiased in the fragment shader, so the edge is smooth at any scale, and `radius: FULL` on a square picture makes a circle, as for an avatar. The radius never exceeds half the shorter side of the drawn rectangle. A shadow is drawn under the picture as under a card (Shadows and elevations), and `opacity` scales the picture's alpha.

The pixels go to the GPU the first time the image is drawn, not when it is made, so an image built and never shown costs no texture. Image textures are RGBA and filtered linearly, unlike the glyph atlas, which is drawn texel for texel.

A picture drawn much smaller than its pixels would shimmer: each screen pixel samples one texel out of many, and the rest are skipped. So an image drawn at half its size or less, in device pixels, uses a smaller level instead: its pixels halved on the CPU, each pixel the average of four, and halved again until the level has at least as many texels as the pixels it covers (`ui::image_level`). A level is made the first time it is needed and kept, with its own texture, so a 512-pixel picture drawn at 64 points stays an even, sharp picture at no extra cost after the first frame. There are at most `ui::MAX_IMAGE_LEVELS` levels, down to one pixel.

20.5 Memory

The `Ui` keeps a copy of each image's pixels and of every level it made, besides the textures on the GPU. A picture drawn at its own size or larger has only its own level; one drawn small takes about a third more, the sum of the halves. `Ui.image_update` drops the smaller levels, which are made again from the new pixels as they are needed. For pictures decoded from files, `max_size` (Loading a picture) is the larger saving: it keeps a picture from ever holding more pixels than it is drawn with.

Keeping the pixels is what lets a lost GPU context recover. When a renderer loses its context and every GPU object with it (The interfaces), `Ui.forget_image_textures` forgets each image's textures without freeing them, and each is uploaded again from the kept pixels the next time it is drawn; the app's handles stay valid and it makes nothing again.

21 Canvas

Some things on screen aren't made of boxes: a chart, a waveform, a route on a map, a signature. A canvas is a box the app draws into with paths, fills, strokes and text, as on an HTML canvas. The app describes the drawing each frame in the canvas's body; Kiln records it, lays the canvas out like any other box, and paints the recording into the draw list as antialiased triangles. What a canvas recorded can also be written out as an SVG file.

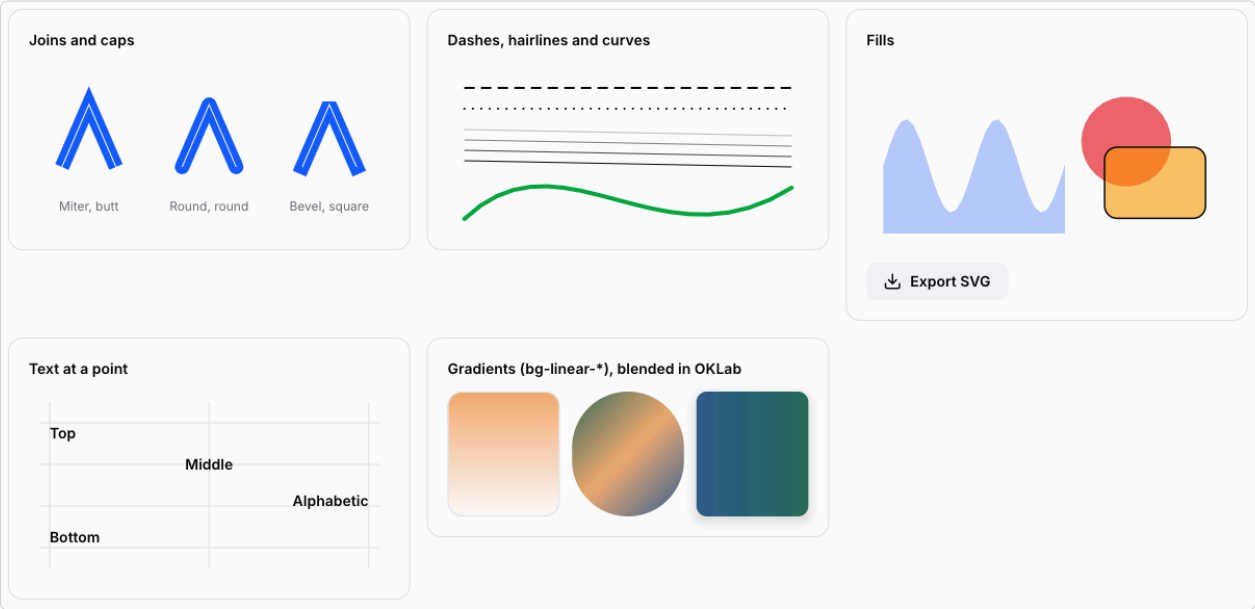


Figure 43: The gallery's canvas page in the light theme. Top left, one path stroked 14 points wide with each join and cap, the path itself drawn thin on top; next to it, a dashed line, a dotted one (round caps on dashes almost zero long), four hairlines from a quarter point to one point wide that fade rather than break, and a cubic curve. Top right, a concave fill under a wave, and a translucent circle and rounded rectangle overlapping, the rectangle stroked as well. Bottom, text placed at a point with each alignment and baseline against guide lines.

21.1 Drawing in a canvas

`Ui.@canvas(id, width, height; Canvas* c)` declares a box `width` by `height`, by default as wide as it can grow and 200 points tall, and runs its body with a `Canvas` to draw into:

```
ui.@canvas("chart", ui::GROW, ui::px(160); Canvas* c)
{
    c.move_to({ 0, c.size.y });
    foreach (i, v : g.samples) c.line_to({ i * c.size.x / (g.samples.len - 1), c.size.y
* (1 - v) });
    c.stroke(ui.c(ACCENT), { .width = 2, .join = ROUND });
};
```

The body draws nothing itself. Each call records a command, and the points, the paths and the commands all live in the frame arena (Lifetimes), so a canvas that draws the same thing every frame allocates nothing once the arena has grown to fit it. The canvas is a `Ui.custom` box whose paint function plays the recording back when the box has been laid out (From Clay to the draw list), so the drawing lands wherever layout puts the box.

Coordinates are in points from the canvas's top-left corner, x to the right and y down, as `Point` values. The body runs before this frame's layout, so it can't know the box's size yet. `c.size` is the size the canvas had last frame, the rectangle `interact()` uses too, and a drawing that lays itself out

to `c.size` fills its box. In the canvas's first frame, `c.size` is the window's width by 200 points. After a resize the drawing is a frame behind and settles on the next one, as a container query does (Container queries).

A canvas has an id, `c.id`, made from the name passed to `@canvas`. Passing it to `Ui.interact` makes the canvas react to the pointer like any widget; `Canvas.pointer` returns where the pointer is over the canvas, from its top-left, while it hovers it, or a fault when it doesn't:

```
ui.@canvas("plot", ui::GROW, ui::px(200); Canvas* c)
{
    draw_plot(ui, c);
    if (try p = c.pointer()) draw_crosshair(ui, c, p);
};
```

21.2 Paths

A path is one or more **subpaths**, each a run of connected points. The calls follow an HTML canvas:

Call	Adds
`Canvas.move_to(p)`	starts a new subpath at `p`
`Canvas.line_to(p)`	a straight line to `p`
`Canvas.quad_to(c, p)`	a quadratic Bézier curve through control point `c` to `p`
`Canvas.cubic_to(c1, c2, p)`	a cubic Bézier curve through `c1` and `c2` to `p`
`Canvas.arc(center, radius, from, to)`	an arc from angle `from` to `to`, in radians clockwise on screen from the x axis, joined to the path by a line
`Canvas.close()`	closes the subpath back to its start
`Canvas.rect(r)`	a closed rectangle subpath
`Canvas.rounded_rect(r, radius)`	a closed rectangle with round corners, the radius clamped to half the shorter side
`Canvas.circle(center, radius)`	a closed circle subpath

Curves are flattened into short lines as they are recorded, never later. Each curve is cut into as many pieces as keep it within a quarter of a pixel of the true curve at the display's scale: the number comes from the length of its control polygon for a Bézier and from its radius and angle for an arc and a circle. A path is from then on a list of points, which keeps the rest simple: strokes, fills and SVG see only polylines.

A point with NaN in it leaves a gap: the subpath ends there, and the next real point starts a new one. A chart passes its samples through as they come, with NaN for the missing ones, and the line breaks where the data does, with no test in the app's loop.

21.2.1 Fill and stroke

`Canvas.fill(color)` and `Canvas.stroke(color, style)` draw the path built since the last fill or stroke, then start a new one:

```
c.circle({ 60, 60 }, 40);
c.fill(ui.c(DANGER).alpha(0.6f));
c.rounded_rect({ 80, 70, 100, 60 }, 12);
c.fill(ui.c(WARNING).alpha(0.6f));     // the rectangle alone, not the circle again
```

An HTML canvas keeps its path after a fill until `beginPath()`, so a forgotten `beginPath()` fills every earlier shape again, each time darker under a translucent color, and a path grows a little every frame. In Kiln there is nothing to forget: a fill or stroke takes its path. To fill and stroke one shape, the app builds it twice, as the gallery's rounded rectangle does; building a path costs little, since it is only points in the arena.

`fill` fills each subpath as its own polygon, closed or not. A polygon may be concave, as the wave in the figure is, but must not cross itself. Subpaths don't cut holes in each other, as they would under a fill rule: two circles one inside the other fill as two discs. A ring is a circle stroked wide.

21.3 Strokes

A `StrokeStyle` says how a path is stroked. Its zero value is a line one point wide, miter-joined and butt-capped, so `c.stroke(color)` draws a plain line:

Field	Meaning
`width`	the width in points, 0 for 1
`join`	how segments meet, a `LineJoin`: `MITER`, a sharp corner; `ROUND`; or `BEVEL`, the corner cut off
`miter_limit`	the longest miter, in widths, before a miter join bevels instead, 0 for 4 (SVG's default)
`cap`	how open ends finish, a `LineCap`: `BUTT`, flat at the end point; `ROUND`, a half circle past it; `SQUARE`, flat half the width past it
`dashes`	dash and gap lengths in points, repeating; empty for a solid line
`dash_offset`	how far into the pattern the line starts

A dashed line is cut into its dashes, and each dash is stroked as an open line with the style's caps, so round caps on dashes almost zero long draw a row of dots, the gallery's second line. A closed subpath is stroked with a join where it meets its start and no caps. The `dashes` slice is copied into the frame arena when the stroke is recorded, so the app may pass a temporary array.

A line thinner than a pixel can't be drawn thinner than its antialiased edge. Kiln draws it a pixel wide and scales its alpha by how much thinner it is, so at a scale of 1 a quarter-point line is a pixel wide at a quarter of the color's alpha: hairlines fade rather than break into dashes where they cross pixel boundaries, the four slanted lines in the figure.

A stroke's triangles never overlap one another, at joins and caps either, so each pixel of a stroke is covered once: a translucent line is even all along, with no darker spot at the corners.

21.4 How a canvas is drawn

The painting tessellates every path into triangles of the draw list's `MESH` mode, plain triangles with a color at each vertex (Meshes). There is no multisampling: each edge of a shape gets a **fringe**, a band one pixel wide whose vertices on the outer side have alpha 0, and the color interpolated across the band ramps the edge from full to nothing. That is how NanoVG and Dear ImGui antialias their paths, and it keeps every edge one pixel soft at any scale, as the `SHAPE` quads' edges are (One shape, one quad).

A stroke is laid out as **stations**, cross-sections of the line. Each station has four vertices on four **rails** running along the stroke: outer left and outer right at the half width plus half a fringe, alpha 0, and left and right at the half width less half a fringe, full alpha. Between two stations, three quads join the rails: the fringe on each side and the solid core between them.

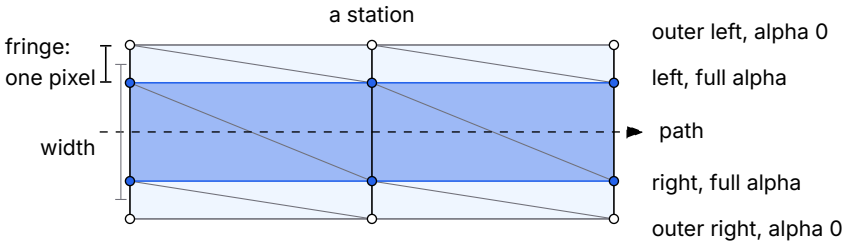


Figure 44: A straight piece of stroke between three stations. The fringes (pale) ramp from full alpha on the left and right rails to 0 on the outer rails; the core between them is solid. Each quad between two stations is two triangles. The fringe is drawn wider than it is: it is one pixel, centered on the stroke's true edge.

The stroke is a strip: each station is joined to the one before it, so a join or a cap is just more stations at one point, swinging the rails around it.

- A **miter** join is one station whose rails are pushed out along the miter, the vector halfway between the two segments' normals, lengthened so the edges meet. Past the miter limit it becomes a bevel.
- A **bevel** join is two stations at the corner: on the inner side both use the point where the inner edges meet, and on the outer side one faces the incoming segment and one the outgoing, so the triangle between them cuts the corner.
- A **round** join is a fan of stations whose outer rails swing from one segment's normal to the other's in steps fine enough to stay within a quarter pixel of the arc.
- A **butt** or **square** cap ends with a station half a fringe inside the end and a last station half a fringe outside it with all four vertices at alpha 0, so the end is antialiased too. A square cap moves the end out by half the width first.
- A **round** cap is a fan of stations whose left and right rails both swing forward until they meet at the tip.

On the inner side of a sharp join, the meeting point is kept from passing the ends of the shorter segment, so a stroke folding back on itself still doesn't overlap. A long stroke that fills a mesh command, 65,536 vertices, continues in a new one, with the last station laid down again to join the two.

A fill builds two rings around its polygon: an inner ring half a fringe inside the outline at full alpha, and an outer ring half a fringe outside it at alpha 0. Each vertex moves along the miter of its two edges' outward normals, kept within two fringes of its vertex so a needle-sharp corner leaves a short spike rather than a long one. Two triangles per edge fill the fringe between the rings, and ear clipping triangulates the inside of the inner ring: it cuts off one corner at a time whose triangle holds no other vertex, which handles concave polygons such as the wave. A polygon that crosses itself runs out of ears; what is left is fanned, which is why such a fill comes out wrong rather than failing.

The two tessellators are `DrawList.fill_polygon` and `DrawList.stroke_polyline`, and a paint function of any widget can call them on its own points, in its own box (The five-step pattern).

21.5 Text

`Canvas.text(at, text, size, weight, color, align, baseline)` draws one line of text at a point: `align` places it across, `START`, `CENTER` or `END` of `at`, and `baseline`, a `TextBaseline`, up and down:

Baseline	The point is
`TOP`	at the top of the line
`MIDDLE`	halfway down the line

Baseline	The point is
`ALPHABETIC`	on the baseline, the default
`BOTTOM`	at the bottom of the line

The line is the theme's line for `size`, so a label set `MIDDLE` on a gridline sits centered on it as a label in a row would. The color defaults to the `TEXT` token. The text is copied to the frame arena and shaped when it is recorded, with the same fonts, fallbacks and caches as every other text (Fonts); it doesn't wrap. `Canvas.text_width` measures a string first, to lay labels out around it: to leave room for an axis's widest tick label, or to skip every other label when they would touch.

The canvas draws its text with `Ui.draw_text_at(list, shaped, at, size, weight, color, align, baseline)`, which any paint function can call to put a label at a point, and `Ui.baseline_y` gives the baseline a `TextBaseline` puts at a given height.

21.6 Clips

`Canvas.push_clip(r)` limits what follows to the rectangle `r`, in the canvas's coordinates, within any clip already pushed; `Canvas.pop_clip` ends it. A chart clips its series to the plot area so a line running off the scale doesn't draw over the axis labels. Clips pushed and not popped end with the canvas, and a pop without a push does nothing, so a mistake can't clip the rest of the page. Without a clip, drawing outside the canvas's box is not cut off.

21.7 SVG

`Ui.canvas_svg(id, allocator)` returns the canvas named `id` that was built since the last `Ui.begin` as an SVG document, in `allocator`, or an empty string when there is none; `Canvas.svg` does the same for a `Canvas*` in hand. The document is `c.size` big and replays the recording:

Recorded	In the SVG
a fill	a `<path d>` per subpath, closed, with `fill` and `fill-opacity`
a stroke	a `<path d>` per subpath with `fill="none"`, `stroke`, `stroke-opacity`, `stroke-width`, `stroke-linejoin`, `stroke-linecap`, `stroke-miterlimit` for a miter, and `stroke-dasharray` and `stroke-dashoffset` when dashed
text	a `<text>` in Inter, weight 400 or 600, with `text-anchor` from the alignment and `y` at the baseline the `TextBaseline` gives
a clip	a `<clipPath>` rectangle and a `<g>` that uses it, closed at the pop

The SVG draws the same thing the screen does, but it isn't the screen's triangles: the browser or vector editor that opens it strokes and antialiases the paths itself. Curves arrive as the short lines they were flattened into, a quarter pixel from the true curve at the scale they were drawn, which is fine to look at and to print at that size but shows facets when scaled far up. Numbers keep two decimals at most.

The canvas must have been built in the current frame before it is asked for, since the recording lives in the frame arena. The gallery's fills canvas has an Export SVG button. It asks the OS where to save with `os::save_file_dialog` (File dialogs), and in the frame the dialog answers, after building the canvas, writes the document with the standard library's `file::save`:

```
ui.@canvas("fills", ui::GROW, ui::px(180); Canvas* c) { draw_fills(ui, c); };
if (ui.button("Export SVG", icon: DOWNLOAD) && !g.svg_dialog.is_open())
{
```

```
    g.svg_dialog.close();
    g.svg_dialog = os::save_file_dialog("Export the fills", "fills.svg", { { "SVG",
{ "*.svg" } } });
}
...
if (try paths = g.svg_dialog.result())
{
    if (paths.len) (void)file::save(paths[0], ui.canvas_svg("fills", tmem));
    os::free_paths(paths);
}
```

22 Charts

A chart shows how numbers go: plays per month, listening by genre, tempo against energy, hours by artist. Kiln draws charts on a canvas (Canvas) in three kinds. A line chart draws lines, areas, stacked or not, and points over a shared x, and can zoom and pan along it (Zoom and pan). A bar chart draws a value per category, its bars grouped or stacked, upright or lying across (Bar charts). A strip chart draws live values as a line that scrolls as they arrive (Strip charts). All three share axes whose ticks fall on round numbers, a legend that hides and shows series, a crosshair or highlight and a tooltip that follow the pointer or the arrow keys. Line and bar charts ease to new data when it changes; a strip chart shows it at once. A sparkline is the small chart without axes that sits next to a number.



Figure 45: The gallery's charts page in the light theme. Top, three line series with the keys stepped to June: the focus ring around the plot, the crosshair with a marker on each series, and the tooltip listing every series at June, Skips with a dash where its data has a gap, as its line does. Bottom left, three stacked areas showing the same crosshair through the shared cursor, without a tooltip of their own; their x labels skip every other month for room. Bottom right, a scatter plot whose x range rounds out to 60 and 180 bpm, with the axes' titles above the y axis and under the x axis.

22.1 A chart

`Ui.chart(id, series, opts)` draws a slice of `Series` with a `ChartOpts`, and returns the x index chosen this frame, or −1 (Choosing a value). The gallery's top chart is three lines over the months, with labels from a callback:

```
fn String month_label(void* data, double v)
{
    int i = (int)math::round(v);
    return i >= 0 && i < MONTHS_SHORT.len ? MONTHS_SHORT[i] : "";
}
```

```
ui.chart("plays", {
    { .name = "Plays", .ys = &PLAYS },
    { .name = "Skips", .ys = &SKIPS },
    { .name = "Saves", .ys = &SAVES },
}, { .height = 240, .x = { .format = &month_label }, .y = { .title = "Plays per
month" }, .cursor = &g.chart_cursor });
```

A series is a set of values and how to draw them:

Field	Meaning
`name`	the series' name, in the legend and the tooltip
`ys`	the values; NaN for a missing one
`xs`	the x of each value; empty for 0, 1, 2..., an index such as a month
`kind`	a `SeriesKind`: `LINE`, `AREA`, a line filled down to the axis, or `POINTS`, markers only, as a scatter plot
`color`	zero for the series' place in the palette (Series colors)
`width`	the line's width in points; 0 for 2
`stacked`	for an area, set on top of the stacked areas before it

The chart reads the slices while it is built and keeps no pointer to them, so the app passes its own arrays, or temporary ones, every frame.

The options set the chart's size, its axes, its legend and the cursor. Their zero value is a chart as wide as its parent and 240 points tall: `width` is a `Size` whose zero value, `FIT`, is taken as `GROW`, since a chart has no natural width; `height` is in points, 0 for 240. `legend` places the legend (Legend) and `cursor` shares the crosshair between charts (Hover and keys).

`x` and `y` are each a `ChartAxis`, whose zero value fits its data and labels it:

Field	Meaning
`title`	the axis's name, above the y axis or under the x axis
`hidden`	no labels, and no gutter for them
`grid`	a `GridLines`: `AUTO` draws lines across the chart for y and none for x; `ON` and `OFF` draw them or not
`fixed`, `min`, `max`	the range is `min` to `max`, not taken from the data
`include_zero`	the range reaches 0; an area's always does, since its fill starts there
`format`, `format_data`	an `AxisFormat` that labels a value; null for numbers with the ticks' decimals

An `AxisFormat` is `fn String(void* data, double value)`. It labels the axis's ticks and, for x, the tooltip's heading and, for y, its values. The string it returns must live through the frame, since the canvas draws it after the chart returns: a literal, or a string formatted into `tmem`, as the gallery's `bpm_label` does with `string::tformat("%.0f bpm", v)`.

22.2 The frame

PLplot, the plotting library, keeps three spaces apart: the **device**, the surface it draws on; the **viewport**, the part of it the plot takes; and the **world**, the data's own coordinates. A chart in Kiln is

laid out the same way. Its device is the canvas; each frame, `ChartFrame` holds the rest: `plot`, the rectangle in the canvas the data is drawn in, and a `Scale` for each axis that maps the world onto it, with each axis's `Ticks`.

A `Scale` is a linear map from a data range, `d0` to `d1`, to points, `r0` to `r1`. `Scale.map` takes a value to points and `Scale.invert` takes points back to a value. The y scale runs from the plot's bottom to its top, so larger values sit higher. `ChartFrame.to_screen(x, y)` maps a data point into the canvas and `ChartFrame.to_data(p)` maps a canvas point back. Drawing, picking the value under the pointer and stepping with the keys all go through these two, so the series, the crosshair and the tooltip always agree on where a value is.

The plot is what is left of the canvas once the labels have room. The chart measures them rather than guessing:

- **Left:** the widest y label, each formatted as it will be drawn, plus 8 points.
- **Top:** half a line, since the top label is centered on its tick and would otherwise be cut, and a line more for the y title when there is one.
- **Bottom:** a line for the x labels, and one more for the x title.
- **Right:** half the last x label, which is centered on its tick at the plot's end, and at least 8 points.

The canvas's size is the one it had last frame (Canvas), so after a resize the frame settles a frame later, as the canvas does.

Text in Kiln doesn't rotate, so the y title doesn't stand on its side next to the axis as it does in a printed plot. It sits above the axis, in the top gutter, at the canvas's left edge: read across like any other label, as in the figure's "Plays per month" and "Energy".

22.3 Ticks

`ui::nice_ticks(min, max, target)` chooses about `target` ticks that cover `min` to `max`. The step is 1, 2 or 5 times a power of ten, the steps people count in: the range divided by the target is rounded to the nearest of them, so 0 to 330 in about five ticks steps by 50. The first tick is at or below `min` and the last at or above `max`, so the ticks enclose the data. An empty range, a single value, is widened by a tenth of it either way, or by 1 around zero. The result, `Ticks`, is a `start`, a `step` and a `count`; `Ticks.at(i)` is tick `i`'s value and `Ticks.end` the last.

The labels take as many decimals as the step needs, none for a step of 50 and one for 0.2, so all of an axis's labels have the same number of decimals. Once the numbers reach five digits they are shortened with a suffix, k for thousands, M for millions and G for billions: 15k rather than 15000, 2.5M rather than 2500000. Scientific plots write a shared $\times 10^{n}$ at the axis's end instead; a dashboard wants each label to read on its own, and a suffix keeps it short. `Ui.tick_label` writes a value with its ticks' decimals and suffix through `Ui.format_number`, so the decimal separator is the user's (The decimal separator), and turns a value within a millionth of a step of zero into 0, so an axis never reads "−0".

How many ticks the chart asks for depends on its size: about one for each 48 points of height, from 2 to 8, and one for each 90 points of width, from 2 to 10, so a taller chart gets more grid lines and a narrow one fewer labels. Unless the axis is `fixed`, the y range then becomes the ticks' range, so the top grid line is a round number above the highest value and the data never touches the plot's edge. The x labels are centered on their ticks; one that would come within 8 points of the label before it is skipped, as the figure's stacked areas skip every other month.

The x range rounds out to its ticks only for a scatter plot, a chart whose series are all `POINTS`: the figure's tempo axis runs from 60 to 180 bpm though the slowest song is at 70. A chart with lines or areas over an index keeps its exact range, so the first value sits on the plot's left edge and the last on its right, January to December, with no empty month either side.

22.4 Series colors

Eight color tokens, `SERIES_1` to `SERIES_8`, color a chart's series in order: blue, orange, green, red, purple, teal, pink and yellow, their hues in `ui::SERIES_HUES`. The hues are spread around the color wheel so that neighbors in the order sit far apart: blue next to orange, green next to red. `ui::default_series(i, dark)` makes series `i`'s color in OKLCH (Colors in OKLCH), every hue at one lightness and chroma, so no series stands out more than another: lightness 0.62 and chroma 0.17 on a light theme, 0.72 and 0.15 on a dark one, where a lighter line reads better on the dark surface.

A theme sets its own palette in `colors.series`; a color it leaves zero gets Kiln's for the theme's lightness, so a theme that says nothing about charts still has a palette that fits it. `Ui.series_color(i)` is the color of series `i`, going round the eight tokens, so a ninth series takes the first color again. A series whose `color` is zero takes `ui.series_color(i)` by its place in the slice, and its swatch in the legend and the tooltip follow.

22.5 Drawing

The chart is a canvas the height of the options, and it draws in layers, back to front:

1. **Grid and labels.** The y grid lines in the `BORDER` token, the y labels to the left of the plot, the x labels under it, and the titles.
2. **The x axis line.** On zero when the y range spans it, so values below zero hang under the line, and at the plot's bottom otherwise.
3. **Areas.** Each area is filled between its top and its base at 28% of its color. An area's base is zero, or for a stacked area the top of the stacked area shown before it, so stacked areas pile up in order and the top one's edge is the total. Areas come first so that the lines and points sit on them.
4. **Lines.** Each line or area's edge is stroked at its width, 2 points by default, with round joins and caps (Strokes), so a sharp change of direction doesn't spike.
5. **Points.** A marker 3.5 points across for each value of a `POINTS` series.

The series are clipped to the plot, with a few points' margin for the markers and line width, so a fixed range smaller than the data cuts the lines off at the plot's edge rather than drawing them over the labels.

A NaN in `ys` is a missing value, and it leaves a gap. A line's points go to the canvas as they are, NaN included, and the canvas ends the line at the NaN and starts again after it (Paths): the figure's Skips line has no May and June. An area is filled in pieces between its gaps. A stacked area over a gap in the one below stands on zero there.

22.6 Hover and keys

The chart shows one x at a time in detail: a dashed crosshair down the plot, a marker ringed in the surface color on each series' value, and a tooltip.

With the pointer over the plot, or within 4 points of its sides, the chart picks the x nearest the pointer, along x only: the pointer can be anywhere up and down the plot. The series share their x, so the first series picks it. A scatter plot is different: its points don't share x, and a crosshair through an x between them means nothing. When every series is `POINTS`, the chart picks the point nearest the pointer in both directions, and only within `ui::CHART_PICK_RADIUS` (12) points of it; elsewhere it shows nothing.

The tooltip is a layer (Next to a point) placed above the highest value shown at that x, with the x's label as its heading, then a row for each series shown: a swatch, the name and the value, in tabular figures so the rows line up. A series with a gap there shows "—" rather than a made-up number.

The tooltip takes no focus, lets the pointer pass and never closes on a press (`dismiss = NEVER`), as `Ui.tooltip` does, so it never takes the hover or a press from the chart under it.

The chart is one focus stop, as a list is (Focus): Tab lands on it, not on each value. While it has focus, Left and Right, or the d-pad's left and right, step through the x values, starting at the first or the last; Home and End go to the ends; and Escape hides the crosshair and tooltip again. The focus ring is drawn around the plot rather than the whole chart, since the plot is what the keys move through, as in the figure. The pointer wins over the keys while it hovers the plot.

`opts.cursor` points to an `int` the app keeps, the x index shown, −1 for none. A chart that has the pointer or the keys writes its index there, and a chart that has neither shows the index it finds there. Charts given the same cursor share the crosshair: the figure's stacked areas show June because the chart above them is on June. The tooltip appears only on the chart the pointer or keys are on; the others show only the crosshair and markers, since a tooltip on each would cover the charts it is there to explain.

22.7 Choosing a value

A chart is also a way to pick one of its values. `ui.chart` returns the x index that a click, or Enter or the gamepad's A on the position the keys stepped to, chose this frame, and −1 otherwise; `ui.bar_chart` returns the category the same way. `Ui.chart_step`, which both charts' keys go through, sets it. A click chooses only with the pointer on a position, so a drag that pans the chart (Zoom and pan) chooses nothing. The dashboard example opens a track's sheet from its top-tracks bar chart:

```
int chosen = ui.bar_chart("top", names, { { .name = "Plays", .ys = values } }, { .height
= 240, .horizontal = true });
if (chosen >= 0) self.open_track(rows[top[chosen]].track);
```

22.8 Legend

`opts.legend` is a `LegendPlace`: `AUTO`, the default, puts the legend on top when there are two series or more and leaves it out for one, whose name the chart's title already gives; `TOP`, `BOTTOM` and `END`, beside the chart, place it; `NONE` leaves it out.

Each entry is a swatch and the series' name, the swatch drawn by kind: a short line for `LINE`, a filled square for `AREA`, a dot for `POINTS`. A click on an entry hides its series, and another shows it again. The entries are a focus group (Focus), a Tab stop of their own before the plot or after it, as the legend is placed, so the arrows move between them and Enter toggles the one with focus. A hidden series' entry is drawn faded.

On top or at the bottom the entries are laid out in lines, as many to a line as fit the room the legend has, so a chart narrower than its legend, on a phone, wraps it onto a second line rather than pushing past its edge. A column at the end doesn't wrap. An entry takes the width of its swatch, its name and their padding (`Ui.legend_entry_width`). The room is the legend's own box as it was laid out last frame, and that box clips, so the entries can't widen the box they are measured against. Measured against a box its own content widens, a row would never wrap: its entries would stretch the box to fit them, and so hold the page wider than the window.

A hidden series leaves the chart entirely, not only the drawing: its values no longer count in the range, so hiding a series of large values lets the others fill the plot; it leaves the tooltip; and a stacked area above it settles on the one below. The chart keeps which series are hidden in its state, by position in the slice.

22.9 Animation

When the data changes, the lines don't jump to it. The chart keeps the values it drew last frame and a spring on the `STATE` motion token (Motion tokens); when a hash of the values changes, the spring restarts from what was drawn, and each value eases from there to the new one. A change in the middle of an ease starts from where the lines are, not from the old data. Pressing the gallery's "Second data" shows it.

The ease only runs when the data keeps its shape, the same number of values in all. New data of another length has no old value to start each new one from, so it is drawn at once; so is a chart's first frame. A value that was or becomes NaN doesn't ease either: there is no line to move. Under reduced motion the spring settles at once (Reduced motion), and every change is drawn as it is.

Hiding a series doesn't ease: the range jumps to the series left.

Two calls let a test see what a chart did. `Ui.chart_drawn(id, index)` is the value the chart drew last frame at `index`, its series one after another, on its way to the data while it eases. `Ui.chart_hides(id, i)` says whether the legend hid series `i`.

22.10 High contrast

Color alone tells the series apart, and a user who can't tell blue from teal, or who asked the OS for more contrast, loses that. PLplot gives each line a style, a dash pattern, besides its color; Kiln does the same under high contrast (`ui.high_contrast`, as for the status widgets), so each series has a second channel:

Series	Line	Marker
1st, 5th	solid	circle
2nd, 6th	long dashes	square
3rd, 7th	dots	diamond
4th, 8th	dash and dot	triangle

A line with 40 values or fewer gets its marker on each value; a longer one would be all markers. A scatter plot's points and the crosshair's markers take the series' marker too. Areas are filled more lightly, at 18%, so the dashed edges stand out on them, and the x axis line is drawn in `TEXT_SECONDARY` rather than the faint `BORDER`. `Canvas.marker(p, r, i, shapes)` draws series `i`'s marker, for a custom chart that wants the same shapes.

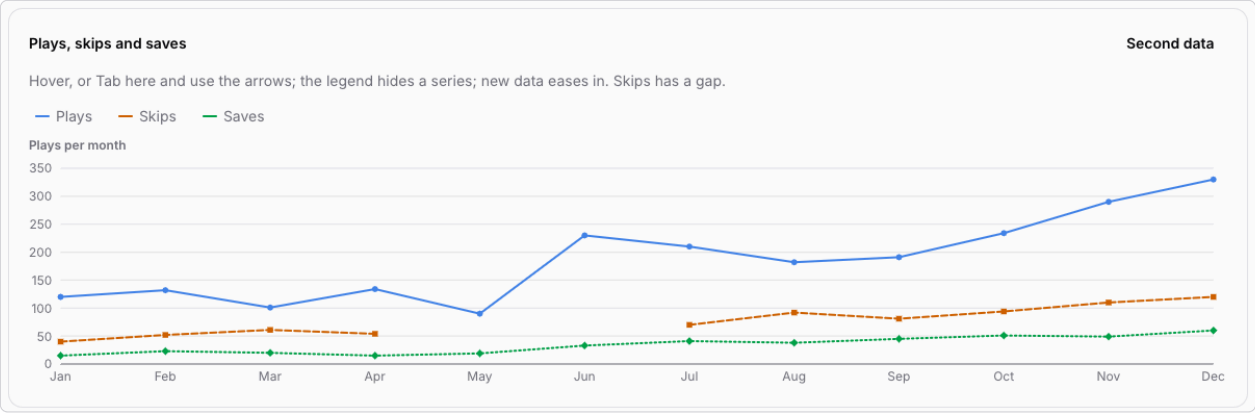


Figure 46: The top chart under high contrast. Plays is a solid line with round markers, Skips long dashes with squares, Saves dots with diamonds, so the three read apart without their colors.

22.11 Bar charts

`Ui.bar_chart(id, categories, series, opts)` draws a bar for each category in each series: the series' `ys` hold one value per category, in the order of `categories`, and the category names label the bands. It returns the category chosen this frame, or −1, as a line chart returns its x (Choosing a value). The gallery's quarters chart groups three series by quarter, one of them negative, with a segmented control that switches the layout; its artists chart lies across, a single series with the names down the side:

```
ui.bar_chart("quarters", { "Q1", "Q2", "Q3", "Q4" }, {
    { .name = "Jazz", .ys = { 120, 160, 140, 190 } },
    { .name = "Classical", .ys = { 90, 80, 110, 105 } },
    { .name = "Returns", .ys = { -20, -35, -15, -30 } },
}, { .height = 220, .layout = BarLayout::from_ordinal(g.bar_layout) });

ui.bar_chart("artists", { "Debussy", "Satie", "Chopin", "Fauré", "Liszt" },
    { { .name = "Hours", .ys = { 42, 35, 28, 19, 12 } } }, { .height = 220, .horizontal
= true });
```

A series' `kind`, `xs` and `stacked` don't apply to bars; its `name` and `color` do. `BarOpts` is the bar chart's `ChartOpts`, with the same zero value, a chart as wide as its parent and 240 points tall:

Field	Meaning
`width`, `height`	as for a line chart: `FIT` is taken as `GROW`, and 0 is 240 points
`values`	the value axis, a `ChartAxis`: up the side, or along the bottom when `horizontal`
`layout`	a `BarLayout`: `GROUPED`, the series side by side in each category, or `STACKED`, one on another
`horizontal`	the categories down the side and the bars across
`legend`	a `LegendPlace`, as for a line chart (Legend)
`cursor`	the category shown, shared with other charts (Hover and keys)

Each category has an equal band of the category axis, and its bars take `ui::BAR_BAND` (0.72) of it, centered, so a gap of the rest separates one category from the next. Grouped bars share that width side by side, with 2 points between them; a stacked category is one bar the whole width.

Bars grow from zero, not from the axis's end: a positive value stands up from the zero line and a negative one hangs below it, or to the left of it when the chart lies across, as the gallery's returns do. Stacked, the positives pile up from zero in the series' order and the negatives pile down from it, so a category's stack shows its gains above the line and its losses below. The range of the value axis always reaches zero, since that is where every bar starts, and then rounds out to its ticks as a line chart's y does (Ticks); a `fixed` axis keeps its own `min` and `max`. The zero line is drawn over the bars' feet.

`Canvas.bar(r, radius, far)` draws a bar with the end at `far` rounded, `ABOVE`, `BELOW`, `START` or `END`, and the end at zero square, so the bar sits flat on the line. The chart rounds the end away from zero by up to 4 points, and less for a thin bar. In a stack only the outermost bar on each side of zero is rounded; the bars inside it meet square, so the stack reads as one bar with a rounded top.

The category labels are placed by the bars' direction. Under upright bars they are centered on their bands, and one that would come within 6 points of the label before it is skipped, as a line chart's x labels are. Lying across, they are right-aligned down the side, each on its band, and the side's

gutter grows to the widest name, up to two fifths of the chart's width, so the bars keep the larger part; a longer name is shortened with "…" to fit (`Ui.fit_text`, which any label can use).

Hover and keys work by band. The pointer over the plot picks the band under it, along the category axis only; the band is shaded, and a tooltip next to the bars' far end lists each series shown with its value, and, when stacked with two values or more, their total. With focus, Left and Right step through the categories, or Up and Down when the chart lies across, through the stepping a line chart uses, with Home, End and Escape as there (Hover and keys). `cursor` shares the category with other charts as it shares a line chart's x.

The legend is a line chart's, its swatches all boxes, as a bar is. The values ease to new data as a line chart's do, a stack's bars together, and under high contrast each bar gets a 1-point outline in the `TEXT` token, so bars of two series that look alike stay apart, and the zero line darkens to `TEXT_SECONDARY`.

22.12 Strip charts

A strip chart is a line of live values, a sensor's level or a meter's reading, that scrolls as they arrive: the newest sample at the right edge, the older ones moving left until they drop off.

The app keeps its samples in a `StripData`, a ring buffer of timestamps and values. `StripData.init(capacity)` allocates it once; `StripData.push(t, v)` adds a sample at time `t`, in seconds, rising, and once the buffer is full each new sample replaces the oldest, so a running chart never allocates. `StripData.at(i)` is sample `i`, oldest first, as a pair of time and value. A NaN value leaves a gap in the line, as in a line chart. `StripData.free` frees the buffer.

`Ui.strip_chart(id, data, window, name, opts)` draws the samples of the last `window` seconds as a line chart (`opts` is a `ChartOpts`) whose series is named `name`. Its x range runs from `window` seconds before the newest sample to the newest, not to the clock, so the line moves only when data comes and stands still when it stops. It plots each sample against its time before the newest one, so x runs from `-window` to 0 and the ticks stay put as the line scrolls beneath them: "−10s", "−8s" and so on to "now" at the right edge (with tenths for a window under five seconds). A format the app gives `opts.x` is handed those same seconds before the newest. Its values show at once (`ChartOpts.instant`): easing each value towards the next frame's would make the line wobble, since a sample's neighbors change as the strip scrolls. A strip chart doesn't zoom.

Nothing in the input says a sample is due, so the app keeps frames coming while the samples arrive. The gallery takes ten a second from the clock, pushing the samples whose time has come and asking for a frame when the next one is due with `Ui.wake_after` (Idle), so an App-style loop that sleeps while nothing moves still draws the strip:

```
// Back after a while away: carry on from now rather than catch up.
if (g.level_next < ui.time - 1) g.level_next = ui.time;
while (g.level_next <= ui.time)
{
    double t = g.level_next;
    g.level.push(t, 50 + 30 * math::sin(t * 1.3) + 12 * math::sin(t * 4.1));
    g.level_next = t + 0.1;
}
ui.wake_after(g.level_next - ui.time);
ui.strip_chart("level", &g.level, 10, "Level", { .height = 200, .y = { .fixed =
true, .min = 0, .max = 100 } });
```

The y axis is `fixed` from 0 to 100 here, so the scale stays put as the values move; a strip that fits its y to the data shifts its grid whenever a peak scrolls in or out.

22.13 Zoom and pan

A line chart with `ChartOpts.zoomable` set zooms and pans along x, for a long series that is too dense to read whole. `Ui.chart_zoom` reads the gestures each frame and keeps the range in the chart's state for the next:

Gesture	Effect
Ctrl/⌘ and the wheel	zooms in or out at the pointer, by 1.2 a notch, so the value under the pointer stays under it
Shift and the wheel, or a sideways wheel	pans, a tenth of the range a notch
a drag	pans: the data follows the pointer
a pinch of two touches	zooms at the point between them, by how far apart they move
Ctrl/⌘ + and −, with focus	zoom in or out at the middle, by 1.5
a double-click, or Ctrl/⌘ 0 with focus	shows all the data again

A plain wheel is left alone. A chart sits in a scrolling page, and a chart that zoomed on the wheel would stop the page scrolling under the pointer, a trap for anyone scrolling past it; so the wheel zooms only with the modifier held, as maps in a page do. The chart takes the wheel from the page, clearing `ui.input.wheel`, only when it uses it: a plain vertical wheel passes through to the scroll area around it (Scrolling).

The range stays within the data: a zoom out stops at all of it, and a pan stops at either end, keeping the range's width. It never gets narrower than a thousandth of the data's, so zooming in can't shrink it to nothing. A range that comes back to the whole data counts as not zoomed. While zoomed, y fits the data in view rather than all of it, unless `opts.y` is `fixed`, so zooming into a flat stretch shows its detail; and a scatter plot's x no longer rounds out to its ticks, which would widen the range the user chose.

`Ui.chart_zoom_range(id)` reads the range a zoomable chart shows, as a pair of x values, and fails with `NOT_FOUND` when it shows all of its data.

The tooltip doesn't get in the way of a drag: it lets presses through to the chart under it rather than closing or taking them (Hover and keys), so a drag that starts on it pans.

22.14 Sparklines

`Ui.sparkline(id, values, tone, area, width, height)` is the small chart, Edward Tufte's word-sized graphic: a line through `values` with no axes, labels or legend, 120 by 32 points by default, filled under the line unless `area` is false, with a dot on the last value, the one a reader compares. The line is scaled to the values' own range, so it shows the trend, not the size. `tone` colors it as a `Tone` colors a status widget (Status), `INFO` by default, so a sparkline of something that should fall can be `DANGER`. A NaN breaks the line.

A sparkline takes no pointer and no focus: it is a picture of a trend beside a number that says how much. It goes next to a metric (Metrics):

```
ui.@row(gap: 6, align: CENTER)
{
    ui.metric("Plays", "330", 0.14f);
    ui.sparkline("plays spark", &PLAYS);
    ui.metric("Skips", "120", 0.09f, up_is_good: false);
```

```
    ui.sparkline("skips spark", &SKIPS, DANGER, area: false);
};
```

23 Rendering

Kiln's core never talks to a GPU. Each frame it turns the laid-out interface into one **draw list**: a vertex array, an index array and a short list of commands. A renderer draws that list with one shader. This chapter covers what the list holds, how one shader draws every shape and every path, how Clay's render commands become drawing calls, and how the commands are batched. The renderers themselves are in Backends.

23.1 The draw list

A vertex carries a position in logical points, a `uv`, a color, four shape parameters, a blur width, the clip rectangle it is drawn in and its draw mode:

```
struct Vertex
{
    float[<2>] pos;      // logical points
    float[<2>] uv;       // SHAPE, IMAGE: position relative to the shape's center; TEXT: texture
coordinates
    Color color;
    float[<4>] sdf;      // SHAPE: half width, half height, corner radius, stroke width (0 = filled);
IMAGE: the same three, then the texture's u
    float blur;          // SHAPE: shadow softness in points, replacing the 1-pixel edge ramp; IMAGE: the
texture's v
    float[<4>] clip;     // the clip rectangle's corners in points (x0, y0, x1, y1); fragments outside it
are dropped
    float mode;          // the `DrawMode` ordinal: how its fragments are colored
}
```

The mode is the ordinal of a `DrawMode`, stored as a float (`ui::MODE_SHAPE`, `MODE_TEXT`, `MODE_IMAGE`, `MODE_MESH`) so the shader reads it like any other attribute. The clip is the rectangle's corners in points, `x0, y0, x1, y1`; every drawing call stamps the clip in force, which the list keeps in `clip_corners` as `push_clip()` and `pop_clip()` change it. Since each vertex carries its own clip and mode, vertices are grouped into commands only by texture. A command names that texture and the range of indices it draws:

```
enum DrawMode : char
{
    SHAPE,   // a rounded box from the vertex's sdf, filled or stroked, maybe blurred
    TEXT,    // the vertex color, its alpha scaled by the texture's red channel (glyph
coverage)
    IMAGE,   // the texture's RGBA times the vertex color, inside a rounded box as
SHAPE's, antialiased
    MESH,    // any triangles, the vertex color interpolated; an alpha-0 fringe
antialiases their edges
    GLASS,   // a live backdrop material: blur, tint and rim from the scene behind
    INK_TEXT,// text whose color comes from the backdrop behind it: black or white,
whichever reads
}
```

```
struct DrawCmd
{
    TextureId texture;   // 0 while only shapes and meshes are in it, which need none
    TexturePending pending;   // what `texture` stands for until `Ui.draw` swaps in a
real texture
    uint vertex_base;    // added to every index of this command
```

```
    uint first_index;    // where its indices start in the list's indices
    uint index_count;    // six per quad, three per mesh triangle
}
```

The four modes share one shader. `SHAPE` evaluates a signed distance (next section). `TEXT` takes the coverage of a glyph from the red channel of an atlas page and multiplies it into the vertex color. `IMAGE` multiplies an RGBA texture by the vertex color inside a rounded box like `SHAPE`'s (Images). `MESH` draws arbitrary triangles in their interpolated vertex color, for paths and fills (Meshes).

The draw list holds the three arrays a renderer reads, plus the clip stack and current clip corners the drawing calls use while the list is built:

```
struct DrawList
{
    List{Vertex} vertices;
    List{ushort} indices;
    List{DrawCmd} cmds;
    List{Rect} clips;
    float[<4>] clip_corners;   // the current clip as x0, y0, x1, y1, stamped on each
vertex
    float fade;          // the floating layer being painted fades out: 0 opaque, 1
invisible
    uint draw_calls;     // set by a GPU renderer as it draws the list: its draw calls
    uint gl_calls;       // and every graphics call it made, draws included
}
```

Indices are 16-bit (`ushort`) and count from the command's `vertex_base`, not from the start of the vertex array; Batching explains why.

23.2 One shape, one quad

Every shape (a rectangle, outline, circle, shadow or triangle) is a single quad of four vertices and six indices. For a `SHAPE` vertex, `uv` is the vertex's offset from the shape's center, and `sdf` holds the shape's half width, half height, corner radius and stroke width, with a stroke width of 0 meaning filled. The quad is a little larger than the shape: one point on every side for the antialiased edge, and the blur width on top of that for a shadow. Figure 47 shows the parameters on one quad.

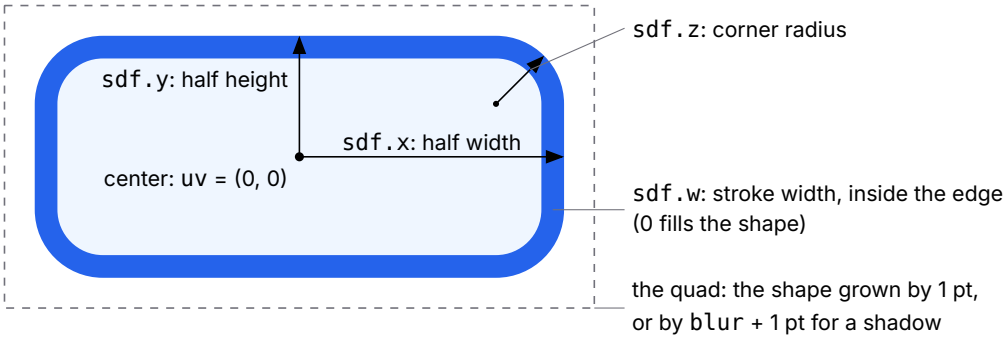


Figure 47: One `SHAPE` quad. The shader computes the distance from each fragment to the edge of the rounded box described by `sdf`; the outlined band is what a stroke width draws. The gap between the shape and the quad is drawn larger than it is.

The fragment shader does the rest. It lives in `backends/gl/shader.c3` (module `kiln::glsl`), and the raylib and OpenGL renderers compile it: `RaylibRenderer` and `GlRenderer`, which the SDL3 backend uses outside Linux and as the fallback (Backends). It holds the shader twice, in GLSL 3.30 for

desktop OpenGL, which both renderers use, and in GLSL ES 1.00 for raylib in the browser (WebGL); the 3.30 vertex and fragment shaders:

```
const ZString VS_330 = `#version 330
in vec2 vertexPosition;
in vec2 vertexTexCoord;
in vec4 vertexColor;
in vec4 sdfParams;
in float blurParam;
in vec4 clipRect;
in float drawMode;
uniform vec2 viewport;
uniform vec2 framebufferSize;
uniform float pixelScale;
out vec2 fragUv;
out vec4 fragColor;
out vec4 fragSdf;
out float fragBlur;
out vec4 fragClip;
out vec2 fragPixel;
out float fragMode;
void main()
{
    fragUv = vertexTexCoord;
    fragColor = vertexColor;
    fragSdf = sdfParams;
    fragBlur = blurParam;
    fragClip = vec4(floor(clipRect.xy * pixelScale), ceil(clipRect.zw * pixelScale));
    fragPixel = vertexPosition / viewport * framebufferSize;
    fragMode = drawMode;
    vec2 ndc = vertexPosition / viewport * 2.0 - 1.0;
    gl_Position = vec4(ndc.x, -ndc.y, 0.0, 1.0);
}
`;
```

```
const ZString FS_330 = `#version 330
in vec2 fragUv;
in vec4 fragColor;
in vec4 fragSdf;
in float fragBlur;
in vec4 fragClip;
in vec2 fragPixel;
in float fragMode;
uniform float pointsPerPixel;
uniform sampler2D texture0;
out vec4 finalColor;

float roundBox(vec2 p, vec2 halfSize, float radius)
{
    vec2 q = abs(p) - halfSize + radius;
    return length(max(q, 0.0)) + min(max(q.x, q.y), 0.0) - radius;
}

// A triangle filling the box of halfSize, pointing by kind: 1 down, 2 up, 3 left, 4 right.
float triangle(vec2 p, vec2 halfSize, float kind)
{
    float w = halfSize.x;
    float h = halfSize.y;
    vec2 p0 = vec2(w, 0.0); vec2 p1 = vec2(-w, h); vec2 p2 = vec2(-w, -h);
    if (kind < 1.5) { p0 = vec2(0.0, h); p1 = vec2(-w, -h); p2 = vec2(w, -h); }
    else if (kind < 2.5) { p0 = vec2(0.0, -h); p1 = vec2(w, h); p2 = vec2(-w, h); }
    else if (kind < 3.5) { p0 = vec2(-w, 0.0); p1 = vec2(w, -h); p2 = vec2(w, h); }
    vec2 e0 = p1 - p0; vec2 e1 = p2 - p1; vec2 e2 = p0 - p2;
    vec2 v0 = p - p0; vec2 v1 = p - p1; vec2 v2 = p - p2;
    vec2 pq0 = v0 - e0 * clamp(dot(v0, e0) / dot(e0, e0), 0.0, 1.0);
    vec2 pq1 = v1 - e1 * clamp(dot(v1, e1) / dot(e1, e1), 0.0, 1.0);
```

```
    vec2 pq2 = v2 - e2 * clamp(dot(v2, e2) / dot(e2, e2), 0.0, 1.0);
    float s = e0.x * e2.y - e0.y * e2.x > 0.0 ? 1.0 : -1.0;
    vec2 d = min(min(vec2(dot(pq0, pq0), s * (v0.x * e0.y - v0.y * e0.x)),
                     vec2(dot(pq1, pq1), s * (v1.x * e1.y - v1.y * e1.x))),
                     vec2(dot(pq2, pq2), s * (v2.x * e2.y - v2.y * e2.x)));
    return d.y > 0.0 ? -sqrt(d.x) : sqrt(d.x);
}

void main()
{
    if (any(lessThan(fragPixel, fragClip.xy)) || any(greaterThan(fragPixel, fragClip.zw))) discard;
    int mode = int(fragMode + 0.5);
    if (mode == 1) { finalColor = vec4(fragColor.rgb, fragColor.a * texture(texture0, fragUv).r); return; }
    if (mode == 2)
    {
        // An image: the texture at (sdf.w, blur), inside the rounded box of its other fields.
        float edge = roundBox(fragUv, fragSdf.xy, fragSdf.z);
        vec4 c = texture(texture0, vec2(fragSdf.w, fragBlur)) * fragColor;
        finalColor = vec4(c.rgb, c.a * clamp(0.5 - edge / pointsPerPixel, 0.0, 1.0));
        return;
    }
    if (mode == 3) { finalColor = fragColor; return; }
    if (mode == 5) { finalColor = vec4(fragColor.rgb, fragColor.a * texture(texture0, fragUv).r); return; }
    float d = fragSdf.w < 0.0 ? triangle(fragUv, fragSdf.xy, -fragSdf.w) : roundBox(fragUv, fragSdf.xy,
fragSdf.z);
    if (fragSdf.w > 0.0) d = abs(d + fragSdf.w * 0.5) - fragSdf.w * 0.5;
    float coverage;
    if (fragBlur > 0.0) coverage = 1.0 - smoothstep(-fragBlur * 0.5, fragBlur, d);
    else coverage = clamp(0.5 - d / pointsPerPixel, 0.0, 1.0);
    finalColor = vec4(fragColor.rgb, fragColor.a * coverage);
}
`;
```

Each fragment first passes the clip test, then the shader branches on its interpolated mode. The vertex shader hands on two things for the test: the vertex's position in framebuffer pixels (`vertexPosition / viewport * framebufferSize`), and its clip in whole pixels, the top-left corner rounded down and the bottom-right corner rounded up after scaling by `pixelScale`. The fragment shader discards a fragment whose pixel center lies outside that rectangle. A scissor rectangle rounded outward the same way keeps exactly the pixels whose centres lie inside it, so the clip keeps the same pixels a scissor would, and the GPU renderers clip like the software renderer (Backends). Because every vertex of a quad or mesh carries the same clip, the clip is constant across a triangle and its interpolation is exact. The GLSL ES 1.00 version declares the clip and pixel-position varyings `highp` where the GPU supports it (`GL_FRAGMENT_PRECISION_HIGH`): `mediump` cannot hold a pixel position past about 2,048 to the half pixel the test depends on.

The shader takes no per-command state besides its texture. Its uniforms are `viewport`, `framebufferSize` and `pixelScale` in the vertex shader, and `pointsPerPixel` and `texture0` in the fragment shader. The SDL_GPU renderer carries the same shader in `SdlGpuRenderer`'s `shader.vert` and `shader.frag`, as Vulkan GLSL compiled to SPIR-V, with the same uniform and attribute names; only the vertex shader differs, flipping its y for Vulkan's top-left framebuffer origin (The SDL_GPU renderer).

For a shape, `roundBox` returns the signed distance in points from the fragment to the rounded box: negative inside, positive outside. A stroke width folds that distance into a band of that width just inside the edge, which is how `DrawList.border` draws an outline without a second shape. The distance then becomes coverage:

- **Without blur**, coverage ramps from 1 to 0 across one pixel centered on the edge. The `pointsPerPixel` uniform converts the distance to pixels, so edges stay one pixel soft at any display scale.

- **With blur**, coverage falls with a smoothstep from half the blur width inside the edge to the full blur width outside it. That soft falloff is a shadow.

A negative stroke width selects the one other shape, a filled triangle, and says which way it points: −1 down, −2 up, −3 left, −4 right. The triangle fills the box of the half width and half height, its point at the middle of one edge and its base along the opposite edge; the shader's `triangle` returns the signed distance to it in place of `roundBox`, and the coverage follows as for any shape, so its edges are antialiased the same way. A triangle has no corner radius, no outline and no blur; an outline is a second, slightly larger triangle drawn beneath it.

So fills, outlines, circles (a square whose radius is half its side), soft shadows and triangles all come from the same quad and the same shader, with no tessellated corners and no shadow textures. What a quad cannot describe, an arbitrary path or polygon, is a mesh (next section). The software renderer evaluates the same distances and coverage per pixel, the triangle's through `ui::triangle_distance`, the C3 twin of the shader's function (Backends).

23.2.1 Images

An image quad uses the shape fields as a filled `SHAPE` quad does: `uv` is the vertex's offset from the picture's centre, and `sdf.xyz` its half width, half height and corner radius. A picture has no stroke and no blur, so the two fields left carry its texture coordinates: `sdf.w` is u and `blur` is v. `DrawList.image` stretches the part of the texture that shows, `tex` (u0, v0, u1, v1), over the picture's rectangle, and carries the coordinates on at the same rate over the one-point antialiasing margin, where the sampler clamps them to the texture's edge.

The fragment shader samples the texture at (`sdf.w`, `blur`) and multiplies it by the vertex color, white at the image's opacity. It then multiplies the alpha by the rounded box's antialiased coverage, the same one-pixel ramp a filled shape gets from `roundBox`. Rounded and circular pictures, such as a card's cover or an avatar with `radius: FULL`, so come out with smooth edges, with no mask texture and no clip.

A texture's format sets its filtering. RGBA8 textures, the images, are filtered linearly, so a picture scales smoothly when it is drawn at another size than its texture's; a picture drawn at half its size or less draws from a smaller level of the image, so the filter never skips texels. The glyph atlas (R8) is sampled texel for texel, since a glyph is drawn at the size it was rasterized.

23.2.2 Gradients

A gradient fill (`Gradient`, Tailwind's `bg-linear-*`) adds no field to the vertex and no branch to the shader. `DrawList.gradient_rect` draws the rounded rectangle as strips across the gradient, sixteen of them, or an eight-by-eight grid for a diagonal one, and gives each vertex the gradient's color where it sits. The GPU blends vertex colors across each triangle, and the software renderer blends a quad's four corner colors bilinearly. Every strip carries the whole rectangle's centre and `sdf`, so the corners, the antialiased edge and the clip come out exactly as one `DrawList.rect` would draw them; only the outer edges are grown for antialiasing, so the strips meet without overlapping. The gradient stays in the page's single draw call.

The gradient line runs through the rectangle's centre at the gradient's angle and is as long as CSS makes it, the width times the angle's sine plus the height times its cosine, so its start and end touch the edges at any angle. Between stops, `Gradient.at` blends in OKLab, as Tailwind v4 does, so red to blue passes through purple rather than a dull grey; between strips the renderers blend linearly, which follows that curve closely at this many steps. A strip is two triangles, and the GPU blends straight across each, so they are split along the color bands (for a diagonal gradient, along the diagonal nearest across it): a peak such as a `via` stop is softened evenly rather than drawn as a sawtooth.

23.3 Meshes

A rounded box and a triangle have a closed-form distance a shader can evaluate per fragment; an arbitrary path does not. Lines, curves, arcs and filled polygons (the canvas, Canvas) are drawn as `MESH` geometry instead: plain triangles, each vertex carrying only a position and a color, and the fragment shader returning the color interpolated across the triangle. `uv`, `sdf` and `blur` stay zero, so meshes need no new vertex fields and share the vertex array, the index array and the blending of every other mode.

A mesh antialiases its edges with geometry rather than a distance. The tessellator in `src/ui/canvas/tessellate.c3` gives every edge a **fringe**: a strip one pixel wide whose outer vertices have alpha 0 and whose inner vertices have the full color, so coverage ramps from 1 to 0 across the strip as the interpolated alpha falls, the same one-pixel ramp a `SHAPE` quad gets from its distance. NanoVG and Dear ImGui antialias the same way; no multisampling is needed.

- `DrawList.fill_polygon` fills a simple polygon, convex or concave: the inside is triangulated by ear clipping, and a ring of fringe triangles surrounds it, each vertex moved half the fringe out or in along the miter of its two edges.
- `DrawList.stroke_polyline` strokes an open or closed polyline with joins, caps and dashes, a fringe along both sides. A stroke's own triangles never overlap, so a translucent line is even; a line thinner than the fringe is drawn as wide as the fringe with its alpha scaled down, so it fades rather than breaks.

Both are built on three lower calls, which any paint function can use for geometry of its own:

```
list.mesh_begin(3);                                  // room for 3 vertices in one command
ushort a = list.mesh_vertex({ 10, 10 }, { 220, 40, 40, 255 });
ushort b = list.mesh_vertex({ 60, 10 }, { 40, 160, 60, 255 });
ushort c = list.mesh_vertex({ 35, 50 }, { 40, 80, 220, 255 });
list.mesh_triangle(a, b, c);                         // a gradient triangle, aliased edges
```

`DrawList.mesh_begin` picks the command the mesh goes into, starting a new command when the last one lacks room for the given number of vertices; `DrawList.mesh_vertex` returns an index relative to that command's `vertex_base`, and `DrawList.mesh_triangle` adds three of them. The clip must not change between `mesh_begin()` and the mesh's last triangle, since the shader interpolates the clip across a triangle, and one mesh has at most 65,536 vertices; the tessellator starts a new command itself when a long path would pass that. Triangles may be wound either way. Without a fringe of its own, a mesh triangle's edges are aliased: a pixel is drawn when its center is inside.

23.4 Drawing calls

The drawing calls are methods on the draw list. Kiln's core calls them while it turns Clay's render commands into the list, and widgets and your own paint functions call them with the box they were laid out in.

Call	Draws
`DrawList.rect(r, color, radius = 0)`	A filled rectangle with rounded corners.
`border(r, color, width, radius = 0)`	An outline `width` points wide, inside `r`.
`DrawList.circle(center, radius, color)`	A filled circle.

Call	Draws
`DrawList.shadow(r, color, blur, radius = 0, offset_y = 0)`	A soft shadow of `r`, spread `blur` points and moved down by `offset_y`.
`DrawList.triangle(r, toward, color)`	A filled triangle filling `r`, its point at the middle of the `toward` edge (a `Side`), as a popover's arrow (An arrow toward the anchor).
`DrawList.focus_ring(r, color, radius, t)`	A 2 pt ring 2 pt outside `r` at opacity `t`; nothing when `t` is near 0.
`DrawList.push_clip(rect)` / `DrawList.pop_clip()`	Restricts what follows to `rect`, intersected with the clip already in force.
`DrawList.glyph(r, uv, color, page)`	One glyph quad from an atlas page; the text code calls it (Text).
`DrawList.image(r, tex, color, radius, texture, pending)`	A picture: the `tex` part of a texture stretched over `r`, its corners rounded and antialiased; `Ui.image` calls it (Images).
`DrawList.quad(box, center, color, sdf, blur)`	The primitive under every shape call: one `SHAPE` quad with explicit parameters.
`DrawList.fill_polygon(points, color, fringe, scratch)`	A simple polygon, convex or concave, as a

Call	Draws
	mesh with an antialiased fringe `fringe` points wide (Meshes).
`DrawList.stroke_polyline(points, closed, color, style, fringe, scratch)`	A polyline stroked with the width, joins, caps and dashes of a `StrokeStyle`, as a mesh.
`DrawList.mesh_begin(vertices)`	Starts a `MESH` command with room for `vertices` vertices.
`DrawList.mesh_vertex(pos, color)`	Adds a mesh vertex and returns its index.
`DrawList.mesh_triangle(a, b, c)`	Adds a triangle between three indices `mesh_vertex()` returned.

A corner radius is clamped to half the shorter side, so any large radius gives a pill (`Corners.FULL` uses 9999). A color with zero alpha adds no geometry at all.

A paint function receives the draw list, its box after layout, and a data pointer. It takes `Color` values, not tokens (Rules), so resolve them with `Ui.c` in the frame function and pass them in the data, which lives in the frame arena (State and memory):

```
struct Meter { float value; Color track; Color fill; }

fn void paint_meter(DrawList* list, Rect r, void* data)
{
    Meter* m = data;
    list.rect(r, m.track, r.height / 2);
    list.rect({ r.x, r.y, r.width * m.value, r.height }, m.fill, r.height / 2);
}

// In the frame function:
Meter* m = ui.frame_new(Meter);
*m = { 0.6f, ui.c(TRACK), ui.c(ACCENT) };
ui.custom(ui::GROW, ui::px(8), &paint_meter, m);
```

23.5 From Clay to the draw list

Your containers and widgets declare boxes; `Ui.custom` declares a box drawn only by its paint function. Clay lays them out (Layout) and returns a list of render commands. In `Ui.end`, Kiln turns each render command into drawing calls:

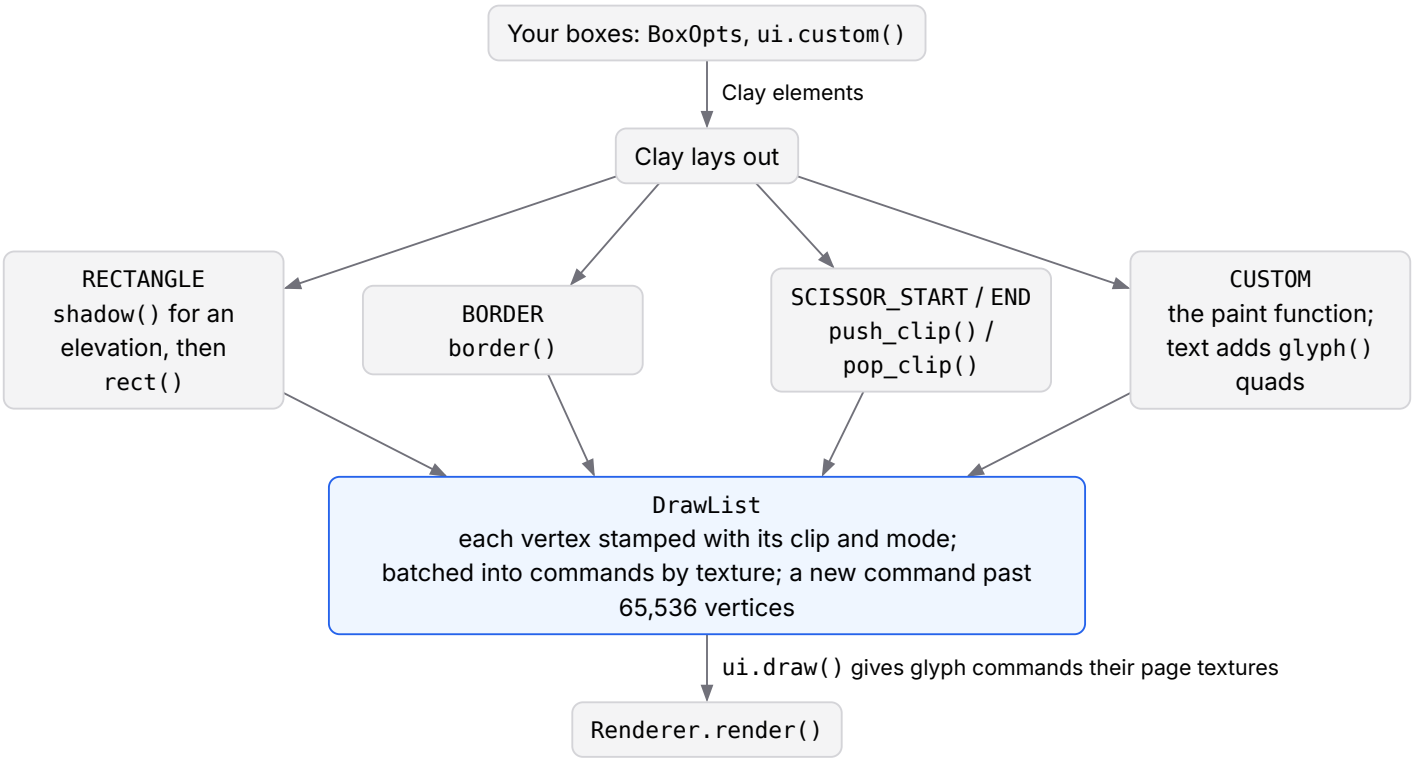


Figure 48: From boxes to the renderer. Each Clay render command becomes drawing calls on the draw list, which batches them into commands; `Ui.draw` gives the commands holding glyphs their atlas textures and hands the list to the renderer.

Render command	Drawing calls
`RECTANGLE`	`rect()` with the box's background and corner radius, after a `shadow()` when the box has an elevation and doesn't clip.
`BORDER`	`border()` with the border's color and width.
`SCISSOR_START`	`push_clip()` with the box's rectangle, for a box with `clip: true`; first the box's `shadow()`, when it has an elevation.
`SCISSOR_END`	`pop_clip()`.
`CUSTOM`	The box's paint function, with its final rectangle and data.

Kiln's boxes have the same radius on every corner and the same border width on every side, so it reads the top-left radius and the left width. Clay produces the commands in a fixed order. For each box on the way down the tree it emits `CUSTOM`, then `SCISSOR_START`, then `RECTANGLE`; then the box's children; then, on the way back up, `BORDER` and `SCISSOR_END`. Two consequences follow:

- A border is drawn over the box's children, so content never covers a card's outline.
- A paint function runs before the box's background rectangle, which would cover it. So a box with both a paint function and a `bg` or `gradient` gives Clay no rectangle: its `CUSTOM` command draws the shadow and the background first, then the paint, over the background and under the children. Kiln's own widgets mostly leave `bg` at `ui::NO_TOKEN` and paint their background themselves.

There is no depth buffer: the draw list is painted in order, and later geometry covers earlier geometry.

Text is not a Clay text element. `Ui.text` declares a `CUSTOM` box whose paint function places the shaped glyphs and calls `glyph()` for each, with the atlas page number in place of a texture and the glyph's pixel rectangle as its coordinates (Text). A picture is a `CUSTOM` box too: `Ui.image`'s paint function calls `image()` with a placeholder for the image and the level it draws, made by `ui::image_texture_ref`. Neither the glyph pages' textures nor the images' exist yet while the frame is built, so a command carries a `TexturePending` kind beside its texture, saying what the texture stands for:

```
enum TexturePending : char
{
    NONE,         // a real texture, or none
    ATLAS_PAGE,   // a glyph-atlas page, plus one
    IMAGE,        // an `ImageId` and one of its levels (`image_texture_ref`)
}
```

`Ui.draw` uploads the atlas, then resolves both kinds before calling the renderer. `Ui.resolve_text` swaps in each `ATLAS_PAGE` command's real page texture and divides the `uv` of its `TEXT` vertices by the page size, leaving the shapes and meshes in the same command untouched (Drawing). `Ui.resolve_images` swaps in each `IMAGE` command's level texture, uploading the level the first time it is drawn. An image freed between paint and draw resolves to no texture, and its quads turn transparent while the shapes beside them stay.

23.5.1 Floating layers

A floating layer (Layers) is a Clay floating element attached to the root, with a `zIndex` of its own: layers declared later get higher ones, and a modal layer's backdrop sits just below its layer. Every render command inside a floating element carries its root's `zIndex`, and Clay sorts the roots in ascending order, so the commands arrive page first (`zIndex` 0), then each layer above the ones below it. There is still no depth buffer: a layer is drawn after the page, so it covers it.

Before each command, Kiln sets the draw list's `fade` from the command's `zIndex`: 0 for the page, and for a layer, how far that layer has faded out. `DrawList.quad`, `DrawList.glyph` and `DrawList.mesh_vertex` scale every color's alpha by (Drawing).

- "fade" so a whole layer, its box, shadow, text and the paint functions of the widgets in it, appears and disappears as one without any widget knowing. Each shape is scaled on its own, not composited as a group, so while a layer fades, overlapping shapes inside it show faintly through one another. A modal backdrop fades by its own color instead: it is the `BACKDROP` token at the theme's `backdrop_opacity` times how far the layer is shown (Tokens).

23.5.2 Shadows and elevations

Clay has no shadows, so Kiln adds them. A box with a background and an elevation carries the elevation to its `RECTANGLE` command (as Clay's user data), and Kiln draws a `shadow()` just before the rectangle. Cards default to `SM`. A box that clips its children is the exception: Clay starts the clip before the box's background, so a shadow drawn with the background would be cut off at the box's own edges. Such a box carries its own Clay id along with the elevation, and Kiln draws its shadow at the `SCISSOR_START` with that id, before pushing the clip, so a clipped card or a scrolling layer keeps its shadow. Each elevation is one shadow quad in black:

Elevation	Blur	Offset down	Opacity
`SM`	3 pt	1 pt	8%

Elevation	Blur	Offset down	Opacity
`MD`	10 pt	4 pt	10%
`LG`	22 pt	10 pt	14%

In a dark theme the opacity doubles, since a shadow is hard to see on a dark surface. Widgets draw their own small shadows with the same call, such as under a slider's thumb or a switch's knob.

23.6 Batching

Every drawing call asks the list for the command its geometry belongs in; a mesh asks once, in `mesh_begin()`, for room for all its vertices. `cmd_for()` decides by texture alone, since clip and mode travel on the vertices. Shapes and meshes need no texture (texture 0), so they fit any command. Glyphs need their atlas page and pictures their image level: they fit a command with no texture yet, which then takes theirs, or one with the same texture and pending kind. Anything else starts a new command, as does geometry that would take the command past its vertex limit. Clip changes and the switches between shapes, paths, text and pictures start no command, so the command count grows only with the atlas pages and image levels a frame draws from: each screen of the widget gallery is one command, and one draw call.

Indices are 16 bits, so one command can address at most 65,536 vertices. Each command's indices count from its own `vertex_base`, and when the next quad or mesh would take a command past that limit, a new command starts at the current vertex. The draw list stays one vertex array and one index array however large it grows. Both GPU renderers point their vertex attributes at `vertex_base` whenever it changes, so they need no base-vertex draw call (Backends). The lists keep their capacity between frames (No allocations per frame).

24 Text

Kiln shapes, breaks and rasterizes text itself, so it looks the same on every backend and handles scripts that a simple left-to-right word wrapper can't. It is built as a pipeline from a UTF-8 string to textured quads in the draw list. Most of it lives in `kiln::text`; `Ui.text` and glyph drawing are in `kiln::ui` (`src/ui/text.c3`).

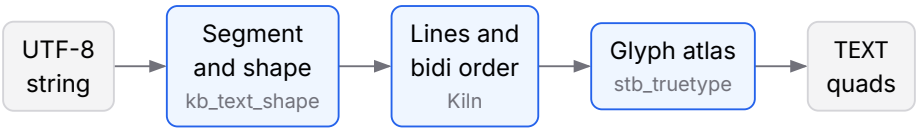


Figure 49: The text pipeline. Each stage is cached or reused across frames, so a steady frame repeats only the last step.

Stage	How
Fonts	**Inter** Regular and SemiBold, subset to Latin (with Vietnamese), Greek and Cyrillic plus punctuation, currency, arrows, the macOS key symbols for menu shortcuts (⌃ ⌘ ⌥, with ⇧ among the arrows) and a few UI symbols, embedded in the library (about 300 KB for both). Characters Inter lacks come from **system fallback fonts** (CJK, Arabic, Hebrew, Devanagari, Thai, Khmer, symbols, math and broad-coverage families), opened only when some text needs them; on the web, where a page can't reach the system's fonts, fallbacks the app lists are downloaded instead.
Segmentation and shaping	**kb_text_shape**, one context per weight: graphemes, line-break opportunities, direction and script runs, OpenType shaping (kerning, ligatures, contextual forms, marks, `tnum`). It picks a face per grapheme from the fonts it has been given.
Lines	Kiln's own: greedy breaking at Unicode line-break opportunities (so CJK breaks between ideographs), hard breaks at newlines, trailing spaces hanging past the width, words longer than a line broken between graphemes, `max_lines` cutting with "…".
Bidi	kb_text_shape returns runs in logical order, with right-to-left runs' glyphs already flipped. Kiln reorders each line's runs with rule L2 of the Unicode bidi algorithm.
Rasterization	**stb_truetype** into one-byte coverage pages, packed in rows, at the window's pixel scale.

24.1 Fonts

A `FontSet` says which fonts a `Ui` draws with: one main font per weight, fonts of your own to fall back on, whether to look through the system's fonts too, and the icon font (Lucide, which `Ui.icon` draws from).

```
struct FontSet
{
    char[] regular;
    char[] semibold;          // empty: use regular
    char[][] fallbacks;       // font files in memory, tried in order
```

```
    bool system_fallbacks;     // also look through the system's font directories
    char[] icons;              // the icon font (Lucide); empty: icons draw nothing
    WebFont[] web_fallbacks;   // the web: fallback fonts downloaded when text needs
them
}
```

`ui::create` takes one as its `fonts` argument, and so do `app::run` and `raylib::create_ui`, which pass it on. Left empty, it gets `text::default_fonts`: the embedded Inter in both weights, the embedded Lucide subset, and `system_fallbacks` on. A main font that doesn't parse stops the program in `create()`; a fallback that doesn't parse is skipped.

Each weight has its own kb_text_shape context, holding the main font of that weight first and every fallback after it. When the shaper finds no glyph for a character in any of them, Kiln opens the system fallbacks lazily. The first time it happens, it walks the system's font directories (the Windows and per-user font folders on Windows; `/System/Library/Fonts`, `/Library/Fonts` and `~/Library/Fonts` on macOS; elsewhere `~/.fonts` and the `fonts` folder of each XDG data directory) and lists the installed files of a fixed set of families, best first: Noto Sans CJK, Microsoft YaHei, Yu Gothic, Malgun Gothic, PingFang, Noto Sans Arabic, Noto Sans Hebrew, Noto Sans Devanagari, Noto Sans Thai, Noto Sans Khmer, Khmer UI, Noto Sans, Noto Sans Symbols, Noto Sans Symbols 2, Noto Sans Math, Noto Emoji, Segoe UI, Segoe UI Symbol, Segoe UI Emoji, Arial Unicode, Symbola and DejaVu Sans. It then opens the first listed file that has the missing character, adds it to every weight's context, and shapes the text again. Files that don't have the character stay closed and stay on the list for others. A character no font has is remembered, so it never sends Kiln through the files again.

Color emoji fonts are bitmaps that stb_truetype can't draw; a monochrome emoji font (Noto Emoji, Segoe UI Emoji) serves if one is installed.

24.1.1 Fallbacks on the web

A page can't reach the system's fonts, so in a browser there are no system fallbacks. Instead, a `FontSet`'s `web_fallbacks` lists `WebFont`s: fonts a web build downloads the first time some text needs one of their characters. Each is a URL, relative to the page, and the characters it covers. A page can't look inside a font before downloading it, so the characters are written down, as CSS `unicode-range` is: `"U+0600-06FF, U+0750-077F, U+4??"`, where each `?` is any hex digit. `text::unicode_ranges` reads that into `CodeRange`s. Other builds ignore web fallbacks.

```
struct WebFont
{
    String url;      // relative to the page, or absolute
    String ranges;
}
```

When shaping finds a character no face has, Kiln first tries the system fallbacks, then the web fallbacks: the first one whose ranges cover the character and that hasn't been asked for yet is fetched with the browser's `fetch()`. While it downloads, the character counts as pending rather than missing, so it isn't added to the set of characters no font has, and the text is drawn without it for now. At the start of each frame, `Ui.begin` takes the fonts that have arrived, before anything is shaped, and adds each with `Fonts.add_fallback`. That clears the shaped-text cache and the missing set, so text is shaped again with the new face: the text appears a frame after its font arrives. A font that fails to download, or doesn't parse, makes its characters missing.

The gallery downloads fonts this way. It passes `text::default_fonts()` with `web_fallbacks` set to `WEB_FONTS`:

```
    FontSet fonts = text::default_fonts();
    fonts.web_fallbacks = &WEB_FONTS;
    if (@ok(env::tget_var("KILN_GALLERY_REFERENCE")))
    {
        g.reference = true;
        g.theme = LIGHT;
        fonts.system_fallbacks = false;
    }
    // The app id a kiln-gallery.desktop file would match; the icon is the window's own.
    app::run("Kiln gallery", 1440, 900, &frame, &g, fonts: fonts, app_id:
"dev.kiln.Gallery", icon: &ICON,
```

`scripts/make_web_fonts.sh` makes the fonts and the list. It finds the installed Noto fonts with `fc-match` and subsets them with HarfBuzz's `hb-subset` into `resources/fonts/web/`, keeping every OpenType feature: whole Unicode blocks for Thai, Devanagari, Arabic, Hebrew and Khmer, and for Japanese the kana plus only the kanji the gallery uses, since all of them would be megabytes. It writes `examples/gallery/web_fonts.c3`, which lists each file with the ranges it covers, and `scripts/web.sh` copies the files beside the page, into `build/web/fonts/`. Each downloads only when its script is shown.

24.1.2 Adding a fallback at run time

`Fonts.add_fallback` adds a fallback after the others while the app runs, on any platform: `data` is a font file, which `Fonts` frees when `owned` is set. Every weight's context gets it, and the cache of shaped text and the missing set are cleared, so characters that had no font can come from it. Call it between frames, before any widget asks for text: shaped text from earlier in the frame doesn't survive it.

24.2 Shaping and lines

`Fonts.shape()` turns a string into `Glyph`s: a glyph index, the face it comes from, its advance and offset, the byte where its text starts, its direction, and flags for the start of a run, a grapheme, and a soft or hard line break before it. Positions are in ems, so one shaped text serves every size by a multiplication. Shaping also records the text's natural width and its widest unbreakable piece (its longest word), which is the least a wrapping text needs.

`text::layout()` breaks shaped text to a width, greedily: it remembers the last soft break opportunity and breaks there when the next glyph would overflow. A hard break (a newline) always ends a line. Spaces at the end of a line hang past the width rather than pushing the break earlier. A word with no break opportunity that is wider than the line breaks between graphemes. With `max_lines`, lines past the last are dropped and the last line is trimmed until "…" fits after it.

Each line is then put in visual order. The shaper's runs come in logical order with right-to-left glyphs already flipped, so Kiln only reorders runs: the paragraph takes the direction of its first run, at level 0 for left to right or 1 for right to left, and the other direction goes one level above it. Rule L2 then reverses every sequence of runs at or above each level, from the highest down to the lowest odd one. The ellipsis joins as a run in the paragraph's direction. Explicit bidi controls (U+202A–202E, U+2066–2069) aren't supported by the shaper and are ignored; they and other invisible characters (zero-width spaces and joiners, variation selectors) take no space.

24.3 Grapheme and word boundaries

Editing needs boundaries apart from shaping: a caret steps and deletes by graphemes, so a letter with combining marks, a Devanagari, Thai or Khmer cluster, a flag or an emoji with joiners is one step, and word jumps and double-clicks need word boundaries. `Breaks` holds both for one string, as flags per byte offset from 0 to the string's length, computed by kb_text_shape's break iterator (Unicode segmentation, UAX #29) through the shim's `kiln_breaks`. The start and end of the string are boundaries of both kinds.

`Breaks.update` recomputes the flags only when the text's hash or length changes, reusing their memory, so an editor can call it every frame. `Breaks.is_grapheme` and `Breaks.is_word` test one offset; `Breaks.next_grapheme` and `Breaks.prev_grapheme` step a caret.

The word moves follow each platform's convention. `Breaks.next_word_end` skips spaces and punctuation, then goes to the end of the word (Option+Right on macOS); `Breaks.next_word_start` goes past the current word and the spaces after it to the start of the next (Ctrl+Right on Windows and Linux); `Breaks.prev_word_start` goes back over spaces to the start of the previous word (Ctrl+Left, Option+Left). `Breaks.word_at` returns the byte range a double-click selects: the word around the offset, or the run of spaces and punctuation it is in. What counts as part of a word is simple: ASCII letters, digits, _ and `'`, and every non-ASCII character except Unicode's spaces and punctuation (the no-break space and Latin-1 signs, the General Punctuation block with its dashes, quotes and ellipsis, CJK punctuation with the ideographic space, and fullwidth punctuation). So words in other scripts, CJK and emoji are words, and the Unicode word boundaries then split them.

kb_text_shape joins only the first pair of an emoji ZWJ sequence, while Unicode keeps every pictograph after a zero-width joiner in the same grapheme (rule GB11). So `update()` removes a grapheme boundary wherever a pictograph (the U+1F000 plane, or the symbol and dingbat blocks U+2600–27BF) follows U+200D, and is one grapheme, not two.

Placing the caret on screen uses the shaped glyphs' byte offsets and each line's visual order instead; Text editing covers that and the editor that uses these boundaries.

24.4 Why not Clay's text

Clay wraps only at spaces and places words left to right, which breaks CJK wrapping and bidi. So `ui.text()` gives Clay a custom element with a size Kiln computes:

- A text with neither `wrap` nor `max_lines` is exactly as wide as its glyphs and one line tall, or, when it has newlines of its own, as wide as its widest line and as tall as its lines.
- A wrapping text grows to the width its parent allows, up to its natural width and never narrower than its longest word (like CSS's min-content). It is as tall as its lines, wrapped at the width it had **last frame**, since Clay hands out widths only after every element has asked for one. After a resize, its height catches up one frame later.
- A line cut with "…" grows the same way up to its natural width, but may shrink to nothing.

A text that needs last frame's width, or that is selectable, gets an id from `Ui.text_id` so its rect can be read back.

When Clay has placed the element, its paint function lays the text out again at the width it actually got and draws it.

24.5 Caches

Shaped text is cached by its bytes, weight and figure style (tabular or not), with glyph positions in ems, so one entry serves every size. Text that stays on screen is shaped once and kept. The cache is swept every 60 frames, and text unused for more than 300 frames (`CACHE_FRAMES`) is dropped; the

"…" used for cutting is kept for the life of the `Ui`. There is no cap on entries or bytes: the cache holds what the last few seconds of frames showed.

Atlas glyphs are keyed by face, glyph, pixel size (to a quarter pixel) and quarter-pixel offset. Pages are 1024 pixels wide, start 512 tall and double up to 2048; when no page has room the atlas grows a page, then adds one, up to four. At most that is four 2 MB pages.

```
const int PAGE_WIDTH = 1024;
const int PAGE_START_HEIGHT = 512;
const int PAGE_MAX_HEIGHT = 2048;
const int MAX_PAGES = 4;
const int PADDING = 1;
const int SUBPIXEL_BINS = 4;
```

With both caches warm, steady frames with text allocate nothing (No allocations per frame). `Ui.stats` reports the atlas's glyphs and pages.

24.5.1 Quarter-pixel positioning

Pen positions keep their fraction to a quarter pixel horizontally, and each quarter is rasterized on its own, so text doesn't shimmer as it animates. Baselines snap to whole pixels.

24.5.2 When the atlas fills

Glyphs that don't fit are skipped for the rest of that frame, and the whole atlas is cleared at the start of the next: every glyph is forgotten, every page zeroed and repacked, and visible text rasterizes again. Pages keep their size, so a cleared atlas doesn't need to grow again.

24.6 Drawing

Each glyph becomes a quad in the draw list in `TEXT` mode, with its atlas page as its texture. The atlas lives on the CPU, and pages become textures only when `Ui.draw` uploads them: a new page, or one that grew, is sent whole; otherwise only the rows that changed since the last upload. Until then, text commands carry the page number and pixel coordinates; after the upload, `draw()` swaps in each page's texture and normalized coordinates. Rendering covers the draw list and how `TEXT` quads are drawn.

24.7 The API

```
fn void Ui.text(&self, String text, TextSize size = BASE, Weight weight = REGULAR, Token color = TEXT,
    int max_lines = 0, bool wrap = false, bool tabular = false, Align align = START, float opacity = 1,
    bool selectable = false, bool ink = false)
```

`size` and `weight` pick from the theme's text scale (Scales), and `color` is a `Token`. `wrap` breaks lines to the parent's width, `tabular` turns on equal-width digits for numbers that change, `align` places each line within the text's box, and `selectable` lets the user select and copy the text (Selectable text). `max_lines` cuts with "…", by one rule that sizing and painting share, so the text's height and what it paints agree: with `wrap`, the text keeps at most `max_lines` lines (0 for no limit); without it, any `max_lines` above 0 means one line, cut with "…" when it doesn't fit the width the parent gives.

```
ui.text("Notifications", LG, SEMIBOLD);               // theme size and weight
ui.text(body, SM, color: TEXT_SECONDARY, wrap: true); // break to the parent's width
ui.text(title, max_lines: 1);                         // cut with "…"
ui.text(price, tabular: true, align: END);            // equal-width digits
ui.textf("Switched %d times", count);                 // formatted in the frame arena
float[<2>] size = ui.measure_text("Save", SM, SEMIBOLD);
```

`Ui.textf` formats into the frame arena and passes the result to `ui.text()` with the default style. `Ui.measure_text` returns the width of the text on one line and the line height of its size.

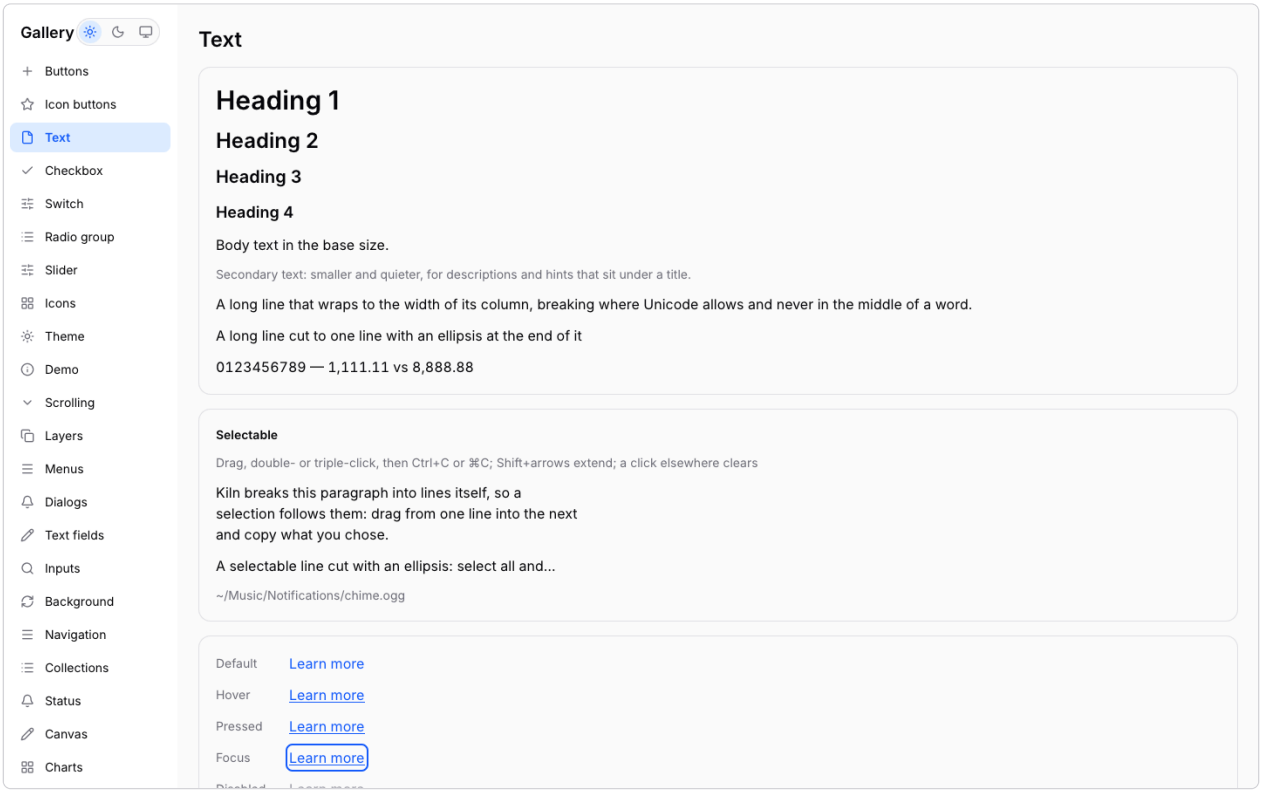


Figure 50: The gallery's text page in the light theme: headings in SemiBold at four sizes, body text in the base size, secondary text in a smaller size and `TEXT_SECONDARY`, a paragraph wrapping at a Unicode break opportunity, a line with `max_lines: 1` (wide enough here to fit without "..."), and tabular figures. Below, a text link in each state.

24.8 Selectable text

A text the user may want to copy, such as an error message, a path or an identifier, asks for it with `selectable: true`; `Ui.secondary` and `Ui.heading` pass the same argument on. Selection is opt-in, as it is for macOS and GTK labels, so a button's caption or a row's title doesn't catch drags meant for its widget.

```
ui.text(path, SM, selectable: true);
ui.secondary(error_message, selectable: true);
```

A selectable text always has an id, from `Ui.text_id`, even on one line. The id comes from the text and its call order within the enclosing id scope, so when the text changes it is a new widget and starts with no selection. Under that id, `Ui.@state_owning` keeps a `SelectableState`: the string's `Breaks` (Grapheme and word boundaries), updated each frame, and the selection's anchor and caret as byte offsets, clamped to the string's length.

`Ui.select_text` handles the pointer. Over the text the cursor is an I-beam. A press places the anchor and the caret at the nearest grapheme boundary, and a drag moves the caret; Shift+click extends the selection the text already holds. A double-click selects the word or the run of spaces and punctuation under the press (`Breaks.word_at`), and a triple-click selects everything. The press takes the keyboard from whatever had it, so the text can copy, select all and extend with Shift and the arrows, without being a focus stop; Escape, a press elsewhere or a change of focus lets it go. Pointers and layers counts the clicks, and Key routing covers how the keys reach the text and

which shortcuts give way to it. On Linux and the BSDs the selection becomes the primary selection once it is made, when a drag ends or the keys change it, so a middle click in another app pastes it; the text itself can't be pasted into (The primary selection).

A selectable text has its own menu, with Copy, disabled while nothing is selected, and Select All. It opens where the user asks for it, as any context menu does (Input and focus, Menus, popovers and tooltips): a secondary press, Ctrl+click with Mac keys, a long touch, the Menu key or Shift+F10. The request makes the text hold the keyboard, as a press does. While the menu is open the selection stays: `select_text()` sets `Ui.selected_menu`, so neither a press inside the menu nor the focus the menu takes clears it, and Copy copies what the user saw selected.

24.8.1 Positioned lines

A `text::Layout` says which glyphs each line holds, in visual order, but not where they are on screen or which bytes they show. `Ui.text_lines` adds both, into the frame arena, as a `TextLines`:

- Each `LineGlyph` has its pen `x` and `advance` in points, whether it is right to left, and the source bytes it shows as a range [`start`, `end`). A glyph's text ends where the next glyph in source order starts, found in one sorted pass over the glyph offsets, so a ligature spans every byte it joins and the glyphs of one cluster share its range. The "…" of a cut line is a glyph flagged `ellipsis`, showing no bytes.
- Each `TextLine` has its top `y` and height, its glyphs, where its last glyph ends, the lowest and the past-the-highest byte it shows, and `cut` when it ends with "…".

`TextLines.line_at` finds the line at a `y`, clamped: the first line above the text, the last below it. `TextLines.edge` gives the `x` of a byte inside a glyph: its leading edge, which is the right edge in a right-to-left run, with a ligature split evenly by the graphemes it holds. `TextLines.caret_x` gives a caret's `x` on a line, and `TextLines.byte_at` maps a point to the nearest grapheme boundary on the line at its `y`.

24.8.2 The selection

`Ui.selection_rects` turns a byte range into rects: one per run of selected glyphs on each line, joined where they touch. A range that crosses directions is contiguous in the source but not on screen, so it shows as several rects on one line. A line the selection continues past is filled to the text's right edge, as native selection does, except a cut line. The ellipsis is never selected: in a cut text only the glyphs shown highlight, while select all and copy take the whole string.

The selection is painted in `SELECTION` under the glyphs. The paint function computes the rects from the layout it has just made at the width the text got this frame, so the highlight always matches the glyphs. The pointer can't wait for that: `select_text()` runs while the text is built, so it lays the text out into `ui.select_layout` at the width it had last frame (Why not Clay's text), and maps the pointer on those lines.

25 Backends

raylib and SDL3 are **backends**, not dependencies of the core. The core never calls a windowing or graphics library itself: two small interfaces, `Platform` and `Renderer`, feed it input and turn its output into pixels. Two backends implement both for real windows: SDL3, the default for App style, with OpenGL by default or SDL_GPU on request; and raylib, which serves Game style, the browser, and App style for a target that chooses it (App style and Game style). The software renderer and the headless platform implement both interfaces for tests, with no window and no GPU.

25.1 Roles a backend library plays

raylib does three jobs in a typical C3 game; SDL3 does the first two. Kiln takes those two through its interfaces and keeps the third in the core:

Role	raylib provides	Kiln's home for it
Platform	window, input, timing, clipboard, DPI, cursor, event waiting	the `Platform` interface
Rendering	OpenGL context, textures, shaders, `rlgl`	the `Renderer` interface
Fonts	font loading (stb_truetype inside)	the core, which calls stb_truetype directly, so text is identical everywhere (Text)

25.2 The interfaces

Both interfaces live in `src/ui/backend.c3`. A renderer manages textures and draws a draw list (Rendering):

```
interface Renderer
{
    <* Starts a frame: clears the drawable to `clear`, `pixel_width` × `pixel_height` pixels. *>
    fn void begin_frame(Color clear, int pixel_width, int pixel_height);
    <* Presents the frame the renderer drew. *>
    fn void present();
    <* Reads the drawable just drawn, top row first, as RGBA8 into `out`. *>
    fn void read_pixels(char[] out, int width, int height);
    <* Creates a `width` × `height` texture in `format`, filled from `pixels` when it is not empty. *>
    fn TextureId texture_create(int width, int height, PixelFormat format, char[] pixels);
    <* Replaces the pixel rectangle `area` of `tex` with `pixels`, packed row by row. *>
    fn void texture_update(TextureId tex, Rect area, char[] pixels);
    <* Frees `tex`. *>
    fn void texture_destroy(TextureId tex);
    <* Whether render() can draw a live backdrop glass command. *>
    fn bool supports_glass();
    <* Draws `list` over a viewport `width` × `height` logical points, `scale` pixels per point. *>
    fn void render(DrawList* list, float width, float height, float scale);
    <* The GPU context is gone: forget every GPU handle without freeing it. *>
    fn void on_gpu_lost();
    <* Frees the renderer and every texture it still holds. *>
    fn void destroy();
}
```

`render()` takes the viewport in logical points and the scale in pixels per point. Textures are `R8` (coverage, for the glyph atlas) or `RGBA8`. `on_gpu_lost()` is for a context whose GPU objects are gone: the raylib and OpenGL renderers drop their handles there, and the software renderer, which has no GPU objects, does nothing.

Three calls frame the draw: `begin_frame(clear, pixel_width, pixel_height)` makes the drawable ready and clears it, `present()` shows the frame, and `read_pixels()` reads the drawable back, top row first, for screenshots. They sit on the interface so the app loop no longer touches OpenGL: `app::run` calls `begin_frame` before `ui.draw()`, then `present()`; a renderer that draws straight through, such as raylib's, leaves the first two empty, and the software renderer's `read_pixels` copies its framebuffer.

A platform fills each frame's `Input` (Input and focus) and answers the questions the core asks of the machine:

```
interface Platform
{
    <*
     Fills `input` with this frame's events. With `wait`, first sleeps until the
     next event, a `wake()`, or `until` (seconds on `time()`'s clock; 0 for no
     deadline), on a platform that `wakes()`; one that doesn't may ignore `until`.
    *>
    fn void poll(Input* input, bool wait, double until);
    <* Wakes a waiting poll from another thread. *>
    fn void wake();
    <*
     True when `wake()` wakes a waiting poll from another thread and `poll` sleeps
     only until `until`, so the `Ui` can let an App-style loop sleep through
     deadlines and running jobs. False where it can't (raylib): frames then keep
     coming until they pass.
    *>
    fn bool wakes();
    <* Seconds since the platform started. *>
    fn double time();
    <* Pixels per logical point on the current display. *>
    fn float dpi_scale();
    <* The screen edges that are covered or unsafe to draw controls under. *>
    fn Insets insets();
    <* Turns text entry (the on-screen keyboard, an input method) on or off for a field, saying what
it holds and where. *>
    fn void text_input(bool on, TextInput input);
    <* Moves a focused field's text area and caret while text entry is on: after typing, scrolling,
the window moving. *>
    fn void text_input_area(Rect area, float caret);
    <* Ends what an input method is composing, as when focus moves; the input method commits or drops
it. *>
    fn void end_composition();
    <* Tells the platform whether the app has an in-app back target (a layer or a page), so it can
register the OS's back gesture; ignored where there is none. *>
    fn void back_target(bool active);
    <* A copy of the clipboard's text, made with `allocator`; empty when there is none. *>
    fn String clipboard_get(Allocator allocator);
    <* Whether the clipboard holds text, asked without reading it where the OS can tell (Android would
show its clipboard preview on a read). *>
    fn bool clipboard_has();
    <* Puts `text` on the clipboard. *>
    fn void clipboard_set(String text);
    <*
     A copy of the primary selection, the text last selected in any app, made with
     `allocator`: what a middle click pastes on Linux and the BSDs. Empty where the
     OS has none.
    *>
    fn String primary_get(Allocator allocator);
    <* Makes `text` the primary selection; does nothing where the OS has none. *>
    fn void primary_set(String text);
    <* Shows `cursor` as the mouse cursor. *>
    fn void set_cursor(Cursor cursor);
```

```
    <* The OS's handle of the window (an HWND on Windows, an NSWindow on macOS, the X11 window or the
Wayland surface on Linux); null for none. *>
    fn void* native_window();
    <* Reads the whole file at `path` into memory made with `allocator`. *>
    fn char[]? load_asset(String path, Allocator allocator);
    <* Frees the platform. *>
    fn void destroy();
}
```

`poll()` fills pointers, keys, gamepad, wheel, time, window size and scale, and `events`, which says whether anything happened. Its `wait` argument asks it to sleep first, until the next event or until `until`, a time on `time()`'s clock (0 for no deadline), the earliest `Ui.wake_after` deadline; `wake()` ends such a sleep from another thread, and a background job calls it from its worker thread when it finishes (Background work). `wakes()` says whether the platform can do both: where it can't, a platform may ignore `until`, and the `Ui` keeps frames coming instead of letting the loop sleep through a deadline or a running job. `primary_get()` and `primary_set()` read and set the primary selection, the text last selected in any app, which a middle click pastes on Linux and the BSDs (The primary selection); a platform without one returns empty text and ignores the set. `insets()` reports safe areas (cutouts, gesture bars, the on-screen keyboard). The page lays out inside them, so a phone's system bars never cover its widgets; layers keep within the same area (Layers).

Three calls carry text entry, for an on-screen keyboard and an input method. A field calls `text_input(true, ...)` when it takes focus and `text_input(false, ...)` when it loses it, passing a `TextInput`: the field's text area in window points, the caret's x from its left, what the field holds (`InputKind`), whether it is a password, and whether Enter makes a new line, so the OS can offer the right keyboard. While the field keeps focus, `text_input_area()` follows its text and caret as typing, scrolling or the window move them, so the input method's candidate window stays by the caret and clear of the text. `end_composition()` ends what an input method is composing, when a press in the field would move the caret: the input method commits the text or drops it. What it is composing comes back in each poll's `Input.composition`, which the field draws at the caret, underlined.

25.2.1 How the core uses them

A `Ui` holds an optional `Platform` and an optional `Renderer`. `Ui.begin` without an `Input` polls the platform; `ui.begin(&input)` takes your input instead and leaves the platform alone, which is how tests script frames. `Ui.end` tells the platform which cursor to show, but only when it changed. `Ui.draw` first sends new and changed glyph atlas pages to the renderer, then hands it the frame's draw list. `Ui.destroy` destroys both. Figure 51 shows one frame.

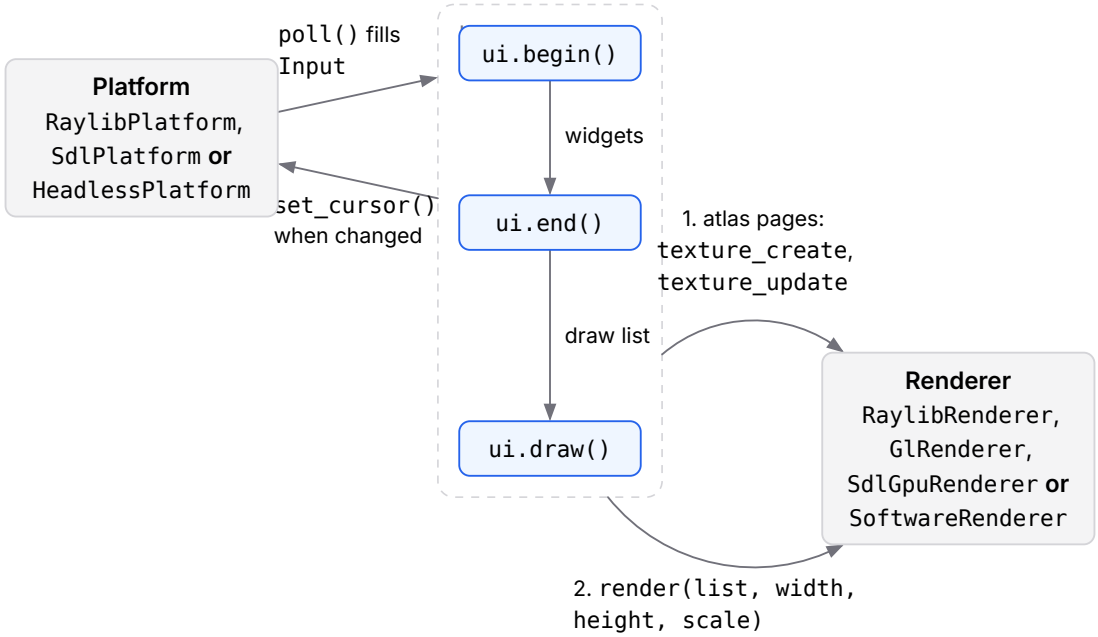


Figure 51: One frame through the two interfaces. Each interface has several implementations: raylib and SDL3 with OpenGL or SDL_GPU for windows, and the headless platform and software renderer for tests.

25.2.2 Implementing an interface

A backend is a struct that declares the interface it implements, with each method marked `@dynamic` so the interface can call it. The headless platform's `poll()` is a complete example: it plays the next scripted frame, or an empty one, at a fixed size and time step.

```
fn void HeadlessPlatform.poll(&self, Input* input, bool wait, double until) @dynamic
{
    *input = self.next < self.script.len ? self.script[self.next++] : (Input){};
    double step = wait && self.can_wake && until > self.now ? until - self.now :
self.dt;
    input.dt = (float)step;
    self.now += step;
    input.time = self.now;
    input.size = self.size;
    input.scale = self.scale;
}
```

Because the core only sees `Platform` and `Renderer`, a new backend (another windowing library, another graphics API) is one struct per interface, with no change to the core.

25.3 Choosing a backend

A backend is a **platform** and a **renderer**, and a target chooses the two separately. The current pairs are:

Platform	Renderer	Default	Used by
SDL3, `SdlPlatform`	OpenGL 3.3 core, `GlRenderer`	outside Linux	App style, and the fallback
SDL3, `SdlPlatform`	SDL_GPU, `SdlGpuRenderer` (Vulkan on Linux)	Linux	App style on Linux; `KILN_RENDERER=gpu` elsewhere

Platform	Renderer	Default	Used by
raylib, `RaylibPlatform`	raylib, `RaylibRenderer` (OpenGL, GLES or WebGL)	always	Game style, the browser, and App style on raylib
headless, `HeadlessPlatform`	the software rasterizer, `SoftwareRenderer`	always	tests and reference screenshots

On Linux SDL_GPU is the default. OpenGL remains the default on the other SDL3 platforms until MSL and DXIL exist, and it is the fallback everywhere; `KILN_RENDERER=gl` forces it. The roadmap records the measurements behind the Linux choice.

The pairs are not a per-OS rule: the core sees only the two interfaces, and a platform and a renderer mix freely where their contracts meet. The raylib renderer can't run in an SDL window because raylib owns its OpenGL context, so SDL3 pairs with its own renderers, OpenGL or SDL_GPU; the software renderer needs no window and can attach to a `Ui` with no platform at all. A new platform or a new renderer is one struct per interface (The interfaces).

A build target picks the pair by the directories it compiles. Each backend has an `app.c3` in module `kiln::app` with the same `app::run`, its loop and its `save_screen`; what both loops share, the `FrameFn` and `DrawFn` callbacks and the `KILN_WINDOW`, `KILN_SCREENSHOT` and `KILN_FRAME_TIMES` handling (`Screenshot`, `FrameTimes`), lives in `backends/app/common.c3`. An app's source is the same on either backend:

Backend	Compiles	Depends on
SDL3	`backends/sdl3`, `backends/sdlgpu`, `backends/gl`, `backends/app`	`sdl3`
raylib	`backends/raylib`, `backends/gl`, `backends/app`	`raylib6`

SDL3 is the default for App style. In `project.json`, the examples build on it under their plain names:

Targets	Backend	For
`hello`, `gallery`, `dashboard`, each with `-release`	SDL3, static from `build/sdl3`	App style
`hello-system`, `gallery-system`, `dashboard-system`	the OS's shared SDL3	App style, release builds (Linking the OS's SDL3)
`hello-raylib`, `gallery-raylib`, `dashboard-raylib`, each with `-release`	raylib	App style on raylib
`overlay`, `game`, each with `-release`	raylib	Game style
`gallery-web`, `dashboard-web`	raylib	the browser

A binary carries one backend: `scripts/check.sh` checks with `nm` that no raylib symbol is in `hello`, `gallery` or `dashboard`, and no SDL symbol in the `-raylib` builds, `overlay` or `game`.

The choice is made by directories, not by a feature flag (`@feat`). A flag can turn sources on and off, but not a target's dependencies, so every app would build and link both `raylib6` and `sdl3`. Nor would a flag save anything at run time: the core calls the `Platform` and `Renderer` interfaces a few times a frame, not once per widget, so there is no per-widget indirect call for a flag to remove.

`backends/gl/shader.c3`, module `kiln::glsl`, holds the signed-distance-field (SDF) shader the raylib and OpenGL renderers compile, in GLSL 330 with a GLSL 100 variant for OpenGL ES 2. The SDL_GPU renderer has its own Vulkan GLSL copy, compiled to the SPIR-V it loads (The SDL_GPU renderer).

25.4 Kiln as a library

A project of its own uses Kiln as C3 libraries rather than copying its source directories. `packages/` holds one `.c3l` directory per library, each with a `manifest.json` whose `sources` point back at the repository's directories, so the libraries and Kiln's own targets compile the same files:

Library	Compiles	Needs
`kiln`	`src/`, and `src/text/shim.c` from C	`clay`
`kiln_sdl3`	`backends/sdl3`, `backends/sdlgpu`, `backends/gl`, `backends/app`	`kiln`, `sdl3`
`kiln_raylib`	`backends/raylib`, `backends/gl`, `backends/app`	`kiln`, `raylib6`
`kiln_software`	`backends/software`	`kiln`

An app takes the core and one backend, as Kiln's own targets do (Choosing a backend); `kiln_sdl3` and `kiln_raylib` both hold `kiln::app` and `kiln::gl`, so they can't be listed together. Tests that draw without a window add `kiln_software`.

c3c loads a library's own dependencies only when the project lists them too, so a project names Kiln's: `clay` and the backend's `sdl3` or `raylib6`, which it finds in Kiln's `lib/`. With the repository copied or linked to `lib/kiln` (under any name not ending in `.c3l`, since the repository root has no manifest), an App-style project on SDL3 sets:

```
"dependency-search-paths": [ "lib", "lib/kiln/packages", "lib/kiln/lib" ],
"dependencies": [ "kiln", "kiln_sdl3", "clay", "sdl3" ],
"linker-search-paths": [ "lib/kiln/build/sdl3" ]
```

The native libraries are built once in Kiln's directory: `native/sdl3/build.sh` writes `build/sdl3/libSDL3.a` (The SDL3 backend), and a raylib project runs `native/raylib/build.sh` and links `build/raylib` instead. `scripts/setup.sh` runs both. A browser build also passes `--js-library` with Kiln's `src/os/web.js` in its link arguments, as `gallery-web` does, since a manifest can't name a path relative to the project that uses it.

The fonts Kiln embeds and the C shim's headers are found relative to their own files, so they resolve wherever the repository sits. `packages/check` is such a project, built only through the manifests: `check.sh` builds its SDL3, raylib and software targets, type-checks the SDL3 one for Windows and macOS, and runs the software one, which draws a frame without a window.

25.5 The raylib backend

`raylib::create_ui` makes a `Ui` with a raylib platform and a raylib renderer attached, for use inside your own raylib loop after `InitWindow()`; `app::run` uses the same pair when Kiln owns the loop on raylib (App style and Game style). A game that reads its own input makes the `Ui` with `ui::create`

instead, attaches a renderer with `raylib::attach_renderer` once the window is open, and calls `ui.begin(&input)` with an `Input` it fills itself; then it needs only `backends/raylib/renderer.c3` and `backends/gl/shader.c3`, not the raylib platform or the app loop, and it can make the `Ui` before the window exists, or with no window at all for its tests.

25.5.1 Renderer

`RaylibRenderer` (`backends/raylib/renderer.c3`) draws the draw list through `rlgl` with the shared SDF shader. It keeps one vertex buffer and one index buffer; when a frame needs more room, a buffer is made again at twice the size needed.

`render()` is built to sit in the middle of a Game-style frame:

- It first flushes raylib's own batch, so whatever the game drew before `ui.draw()` lands underneath the UI.
- It uploads the vertices and indices, sets the uniforms once (`viewport`, `framebufferSize`, `pixelScale`, `pointsPerPixel` and `texture0`), then draws each command. The shader reads seven vertex attributes: position, uv, color, `sdf`, `blur`, `clipRect` and `drawMode`. A draw command ends only where a different texture is needed or past 65,536 vertices, so a frame is about one draw per texture: the renderer binds a command's texture only when it has one and it differs from the bound texture, and shapes and meshes draw with whatever is bound. Each vertex carries its mode and its clip rectangle, in points; the fragment shader discards a pixel whose center lies outside the clip, with the bounds rounded outward to whole framebuffer pixels, so a clip keeps the same pixels a scissor would. Modes 0 to 2 are `SHAPE`, `TEXT` and `IMAGE`; mode 3, `MESH`, returns the interpolated vertex color as it is in both GLSL versions, its antialiasing already in the alpha of the mesh's fringe (Meshes).
- Before binding anything of its own, it sets its blending: colour by alpha, as raylib's alpha blending does, but alpha itself as "over" (one for the source, one minus its alpha for the destination), so the framebuffer stays opaque over an opaque background; a browser composites the canvas over the page by that alpha. The order matters: changing raylib's blend mode first draws raylib's pending batch with whatever is bound. It also turns depth testing and face culling off for the UI.
- At the end it hands raylib its state back: no shader bound (`rlDisableShader`), no texture, vertex arrays and buffers unbound, raylib's alpha blending mode set again, and face culling on, which is raylib's default: rlgl keeps no flag to read back, so a game that keeps culling off turns it off again after `ui.draw()`. Depth testing is left off, as raylib leaves it outside `BeginMode3D`. `scripts/raylib_state_check.c3`, run by `check.sh` where there is a display, checks the culling it finds after a frame.
- rlgl only draws 16-bit indices with no base vertex, so the renderer re-points the vertex attributes whenever a command's `vertex_base` changes.

`on_gpu_lost()` forgets the shader, vertex array, buffers and textures; the next `render()` creates the shader and vertex array again.

25.5.2 Platform

`RaylibPlatform` (`backends/raylib/platform.c3`) reads raylib's window and devices:

- **Pointer:** desktop builds read the mouse as pointer 0, with its left, right and middle buttons as `down`, `secondary` and `middle`. raylib also reports a held mouse as a touch point, so touch points are ignored there.
- **Keys:** raylib keys map to Kiln's `Key` enum, with key repeat counted as a press; Shift, Ctrl, Alt and Super fill the modifiers.
- **Gamepad:** the first gamepad's face buttons, shoulders, start and back and d-pad, with the left stick read as a d-pad.

- **Waiting:** `wait` turns raylib's event waiting on or off, which takes effect at the next `EndDrawing()`. raylib's wait has no timeout, so `until` is ignored.
- **Waking:** `wake()` does nothing, because raylib has no way to wake a waiting window from another thread, and `wakes()` is false. That is why the `Ui` counts as animating while any job is pending or a `Ui.wake_after` deadline is still to come: the loop keeps drawing frames and keeps event waiting off until they pass (Background work, App style: Kiln owns the loop).
- **The rest:** time, DPI scale, clipboard, cursor and asset loading come from the matching raylib calls. raylib has no primary selection, so `primary_get()` returns empty text and `primary_set()` does nothing. `insets()` reports none.
- **Text entry:** `text_input()`, `text_input_area()` and `end_composition()` do nothing. GLFW passes only committed characters, so there is never a composition to show; on X11 an input method draws what it is composing in a window of its own.

25.6 The SDL3 backend

`sdl::create_ui` opens a window and makes a `Ui` with an `SdlPlatform` and a renderer attached; `app::run` on SDL3 builds on it, and `sdl::platform_of` gives the platform back, for showing the frame and asking whether the window is closing. On Linux the renderer is SDL_GPU, on Vulkan; elsewhere it is OpenGL 3.3 core, which loads its functions at runtime, until Metal and Direct3D 12 shaders exist. `KILN_RENDERER=gl` forces OpenGL and `KILN_RENDERER=gpu` asks for SDL_GPU; an unknown value keeps OpenGL. The window is made hidden; when the Vulkan device, the window claim or the pipeline can't be made, it is replaced with an OpenGL window before either is shown, and the app says why. Without a display there is no window, so the app prints `kiln: no window could be opened` and exits with status 1, as on raylib. On Linux, Wayland finds the session's socket through `XDG_RUNTIME_DIR` even when `WAYLAND_DISPLAY` is unset.

When a screenshot is due, the SDL3 loop asks the renderer to read the frame with `read_pixels()` before `present()` swaps or submits it, because after that the drawable's contents are undefined, and saves the pixels with `SDL_SavePNG`.

25.6.1 The OpenGL renderer

`backends/gl/` is plain OpenGL 3.3 core and knows nothing of SDL. `gl.c3` looks every function it calls up at runtime with `gl::load`, through a `LoadFn` the platform passes (`SDL_GL_GetProcAddress` on SDL3), so nothing links libGL: the driver is found when the context is made. A function the driver lacks fails at startup, naming it, with `gl::MISSING_FUNCTION`.

`GlRenderer` (`backends/gl/renderer.c3`, made by `gl::new_renderer`) is a port of the raylib renderer: the same shader, the same buffers that grow to twice the size needed, the same blending, the same clipping in the shader. Owning the context, it differs where raylib's state no longer matters:

- It sets all of its state on every `render()`: blending, the viewport, depth testing, face culling and the scissor test off. There is no batch to flush and nothing to hand back.
- It binds a texture only when a command has one that differs from the bound texture, and re-points the vertex attributes when a command's `vertex_base` changes, as the raylib renderer does.
- `R8` atlas pages are `GL_R8` textures read through a (R, R, R, 1) swizzle, so coverage reads as opaque grey, as raylib's do. Every texture samples `NEAREST` and clamps to its edge.

`gl::clear_to` clears the drawable before a frame and `gl::read_pixels` reads it back top row first, for screenshots.

25.6.2 The SDL_GPU renderer

`backends/sdlgpu/`, module `kiln::sdlgpu`, is a second SDL3 renderer through SDL's GPU API. `SdlGpuRenderer` (`renderer.c3`, made by `sdlgpu::new_renderer`) implements the same `Renderer` as the others, so no widget or app changes. The first slice is Vulkan on Linux, and it is the default there;

OpenGL is the fallback, and the default on other platforms until MSL and DXIL exist. The window is created hidden and only shown once the device, the window claim and the pipeline are ready; a failure at any of them replaces it with an OpenGL window, still hidden, so the fallback has no visible flash. On Windows and macOS `KILN_RENDERER=gpu` falls back to OpenGL with a message, because their shader formats (DXIL and MSL) are not built yet; the roadmap tracks them.

`SdlPlatform` makes the `SDL_GPUDevice` before the window, so a machine without Vulkan can still have an OpenGL window, and claims the window for the device once it exists. The swapchain is `SDR` with vsync, or immediate when the driver refuses vsync. `present()` submits the frame; SDL_GPU's submission is the present, so the platform does not swap buffers.

The renderer keeps one vertex buffer and one index buffer, grown to twice what a frame needs, and a texture list with two shared samplers: nearest for `R8` atlas pages, linear for `RGBA8` pictures, both clamped to the edge. Each frame:

- `begin_frame()` acquires a command buffer and the swapchain texture. A minimized window gives no texture, and the frame is skipped.
- `render()` uploads the frame's vertices and indices through a transfer buffer and a copy pass, pushes one vertex uniform buffer (`viewport`, `framebufferSize`, `pixelScale`) and one fragment uniform buffer (`pointsPerPixel`), then opens one render pass that clears to the frame's color. It binds the pipeline and the two buffers once, and for each command binds its texture and sampler and calls one indexed draw. `vertex_base` is SDL_GPU's `vertex_offset`, so commands need no attribute re-pointing. Shapes and meshes draw with a 1×1 white texture.
- `present()` ends the pass and submits. The transfer buffer is released after the submit.
- `read_pixels()` ends the pass early, copies the swapchain into a download transfer buffer, submits with a fence and waits, so a screenshot is the frame that was just drawn. It swaps red and blue when the swapchain is BGRA.

`SdlPlatform.set_vsync(false)` turns vertical sync off for frame-time recording: it sets GL's swap interval to zero and asks SDL_GPU for `MAILBOX`, then `IMMEDIATE`, then leaves `VSYNC` when neither is supported. That makes the two renderers' unpaced frames comparable.

The shader is Vulkan GLSL 450 in `shader.vert` and `shader.frag`, with one vertex uniform buffer at set 1, the sampler at set 2 and the fragment uniform buffer at set 3 — the resource sets SDL_GPU assigns for SPIR-V. The vertex attributes are the seven fields of `Vertex`, and `shader.vert` flips y because Vulkan's framebuffer origin is top left where OpenGL's is bottom left; without the flip the UI is upside down. `scripts/make_sdlgpu_shaders.sh` compiles both sources with `glslangValidator -V` to `shader.vert.spv` and `shader.frag.spv`. The `.spv` files are checked in and embedded with `$embed`, so building Kiln needs no shader compiler and every machine gets the same bytes; rerun the script after changing the GLSL. MSL and DXIL will be generated from the same sources for Metal and Direct3D 12.

`app::run` keeps OpenGL when `draw_under` or `draw_over` is given: those callbacks draw with raw GL, which an SDL_GPU window has no context for. A `Ui` of your own that draws GL under or over the UI passes `needs_raw_gl: true` to `sdl::create_ui`. A renderer-neutral under/over layer is needed before SDL_GPU can support them.

25.6.3 Platform

`SdlPlatform` (`backends/sdl3/platform.c3`) reads SDL's events:

- **Points:** a point is a pixel divided by `SDL_GetWindowDisplayScale`. SDL's window coordinates are pixels on Windows and X11 and points on macOS and Wayland, so the platform converts each position by the window's pixel density over that scale, and resizes the window once after creating it, so it opens at the size asked for in points.

- **Pointer:** the mouse is pointer 0, with buttons 1, 3 and 2 (left, right and middle) as `down`, `secondary` and `middle`, and fingers are touch pointers. The mouse events SDL makes up from touches are ignored (the hint `SDL_TOUCH_MOUSE_EVENTS` is `0`), so a tap is not reported twice.
- **Keys:** letters, digits and the punctuation Kiln knows map by keycode, the character the layout gives the key, so Ctrl+Z is Z on any layout; the other keys map by scancode, their place on the keyboard. Android's back key is Escape.
- **Typed text:** only while a field has turned text entry on. `text_input()` starts SDL's text input with `SDL_StartTextInputWithProperties`: `SDL.textinput.type` is `TEXT`, `TEXT_NAME`, `TEXT_EMAIL`, `TEXT_USERNAME` or `NUMBER` by the field's `InputKind` (the `PASSWORD_HIDDEN` variant for a password), `SDL.textinput.multiline` says whether Enter makes a new line, and email addresses, user names and numbers ask for no capitalization and no autocorrection. Wayland passes the purpose and hints to the input method and the on-screen keyboard (GNOME's, KDE's); a Steam Deck in Game Mode, where Steam sets `SDL_ENABLE_STEAM_SCREEN_KEYBOARD`, opens its keyboard through `steam://open/keyboard` in numeric, email or multi-line mode; Android picks its input type.
- **Text area:** `text_input_area()` (and `text_input()` when it starts) calls `SDL_SetTextInputArea` with the field's text and the caret's offset, in SDL's window coordinates. Each system uses it its own way: Windows keeps the candidate list off the rectangle, X11's XIM puts its window at the caret's bottom-left, Wayland sends the compositor a caret rectangle, Steam places its keyboard clear of the rectangle, and macOS uses the whole rectangle.
- **The app id and icon:** given an `app_id`, `sdl::create_ui` calls `SDL_SetAppMetadata` before SDL starts, which SDL sends as the Wayland app id (`xdg_toplevel.set_app_id`) and the X11 `WM_CLASS`. An `icon` becomes an `SDL_Surface` over its decoded pixels for `SDL_SetWindowIcon`, which SDL turns into `_NET_WM_ICON` on X11 and the toplevel-icon protocol on Wayland compositors that have it (App style and Game style).
- **The title bar:** given `title_bar`, `sdl::create_ui` sets `SDL_VIDEO_WAYLAND_ALLOW_LIBDECOR` to `0` before SDL starts, opens a borderless window when the driver is Wayland, and gives SDL a hit test: the strip left of the window buttons is a drag region, the buttons stay normal, and six-point edges resize it. Kiln's SDL build patches the Wayland hit test so a press on a drag region waits for motion or a second press: a double-click maximizes or restores, and a drag starts the move after four points. `app::run` builds `Ui.title_bar` above the app's frame; other drivers keep the OS title bar (The title bar Kiln draws).
- **Composition:** `sdl::create_ui` sets `SDL_HINT_IME_IMPLEMENTED_UI` to `composition` before SDL starts. Wayland's text-input-v3 and macOS always hand the text being composed (the pre-edit) to the app as `SDL_EVENT_TEXT_EDITING` and never draw it; X11's input methods (XIM, IBus, Fcitx) and Windows' IMM draw it in a window of their own unless the app says it draws it. With the hint, Kiln draws it the same way everywhere, inline at the caret. The candidate list stays the input method's own window: Kiln doesn't ask for `SDL_EVENT_TEXT_EDITING_CANDIDATES`. SDL frees an event's text at the next poll and counts its `start` and `length` in characters, so the platform copies the text and turns them into byte offsets. SDL sends only changes, so the platform keeps the composition between events and reports it to `Input` on every poll; losing focus and `text_input(false, ...)` clear it.
- **Ending a composition:** `end_composition()` is `SDL_ClearComposition` on Windows. Elsewhere SDL can't clear a composition, so the platform stops text input and starts it again, which tells the input method focus moved, and it commits or drops the text.
- **Gamepad:** the first gamepad's buttons by position (south is A), shoulders, start and back and d-pad, with the left stick read as a d-pad past half its travel. The gamepad subsystem starts after the first frame, because looking for gamepads is slow; SDL then reports those already plugged in.

- **Waiting and waking:** an idle frame waits when the loop sets `Ui.wait_for_events`: in `SDL_WaitEvent` without a deadline, and in `SDL_WaitEventTimeout` until `until` with one. The timeout is rounded up to whole milliseconds, so the wait never ends just short of the deadline and comes straight back to wait again. `wake()` pushes a user event registered at start, which is safe from any thread, so a finished job wakes the loop (Background work); the wake event only ends the wait and doesn't set `events`. `wakes()` is true.
- **Showing the frame:** `SdlPlatform.present` swaps the GL buffers (with SDL_GPU the renderer's `present()` has already submitted the frame) and, unless told the frame is unpaced, returns no sooner than the display's refresh period less a tenth after the last frame (`SdlPlatform.refresh_period`, 60 Hz when SDL doesn't know the rate). Vsync normally does that wait, but SDL skips it on Wayland while the compositor isn't showing the window, and a driver setting can turn vsync off; an animation would then draw as fast as the CPU can. The loop presents unpaced only while it records frame times, which it takes back to back, as on raylib.
- **Primary selection:** on Linux and the BSDs (`ui::HAS_PRIMARY_SELECTION`), `primary_get()` and `primary_set()` are `SDL_GetPrimarySelectionText` and `SDL_SetPrimarySelectionText`, over X11 or Wayland. Elsewhere SDL would keep a primary selection inside the app, which isn't how those systems behave, so there is none.
- **The rest:** `insets()` come from `SDL_GetWindowSafeArea`; `native_window()` reads the window's properties (an `HWND` on Windows, an `NSWindow` on macOS, the Wayland surface or the X11 window's id on Linux); cursors are SDL's system cursors, made once and kept; the clipboard is `SDL_GetClipboardText` and `SDL_SetClipboardText`; assets load with `SDL_LoadFile`.

25.6.4 Building SDL3

`native/sdl3/build.sh`, with `native/sdl3/CMakeLists.txt`, builds SDL release 3.4.20 into `build/sdl3`, downloading it checked by its SHA-256 (`SDL3_DIR` builds a checkout instead). The library is static only, built for size (`MinSizeRel`) with each function and datum in its own section so the linker drops what no one calls, and trimmed to what Kiln uses:

Kept	Left out
video, GPU, OpenGL and OpenGL ES, Vulkan, X11 and Wayland (opened at runtime with `dlopen`), joystick and gamepad, D-Bus and IBus	audio, `SDL_Renderer`, camera, haptic, HIDAPI, sensor, power, dialog, tray, KMS/DRM, the offscreen and dummy video drivers, virtual joysticks,

Kept	Left out
	FriBidi, libthai, io_uring

`build.sh --full` builds every part into `build/sdl3-full`, for comparing sizes (Performance).

SDL keeps its **dynamic API**: every SDL call goes through a table of function pointers that SDL fills at startup. With `SDL3_DYNAMIC_API` set to a shared SDL3, such as `/usr/lib/libSDL3.so.0`, it fills the table from that library instead of its own copy, so a user can run a static app on a newer SDL without rebuilding it. Because the table refers to every SDL function, the linker can drop none of them.

25.6.5 Linking the OS's SDL3

`hello-system`, `gallery-system` and `dashboard-system` are release builds that don't search `build/sdl3`, so the linker finds the OS's shared `libSDL3.so` and the app carries no SDL of its own. This suits a Linux distribution's package, where SDL3 is already installed and shared with other programs. On Windows and macOS no SDL is installed, so an app shipped with its own shared SDL is larger than a static one: a shared library has to keep every function, since it can't know which ones an app calls.

The OS's SDL can be older than the one the binding matches. `sdl::create_ui` checks `SDL_GetVersion` before anything else and exits with `kiln: this app needs SDL 3.4 or newer` when it is, rather than failing later at a call the older SDL lacks; a static app pointed at an older library with `SDL3_DYNAMIC_API` gets the same message. `check.sh` builds the three examples and draws a frame with each when the C compiler can link `-lSDL3`, and skips them otherwise.

The binding, `lib/sdl3.c3l`, comes from the C3 vendor repository, with the changes its `README.md` lists. `scripts/sdl3_symbols.sh`, which `check.sh` runs, checks that every function it declares is in the built library.

25.7 The software renderer and the headless platform

`backends/software/` gives a `Ui` that needs no window and no GPU, for tests (Testing).

`SoftwareRenderer` (`backends/software/renderer.c3`) rasterizes the draw list on the CPU into an RGBA8 framebuffer, computing what the GPU shader computes: the rounded-box distance per pixel, the same edge antialiasing and soft shadows, glyphs sampled from the atlas, straight-alpha "source over" blending. It walks each command's indices, taking a primitive's clip and mode from its first vertex: a `MESH` vertex starts a triangle of three indices, any other a quad of six. The clip's box in framebuffer pixels, rounded outward as in the shader, is worked out again only when the clip changes. Shapes, glyphs and images are axis-aligned quads, so it fills each quad's pixel rectangle within that box directly, drawing a pixel when its center is inside, as a GPU does. Mesh triangles go through `fill_triangle()`, which rasterizes a triangle as a GPU does: it evaluates the three edge functions at each pixel center in the triangle's bounding box, draws the pixel when the center is inside all three edges, and interpolates the vertex colors by the barycentric weights those functions give, then blends with the same "source over". Triangles of either winding are accepted. A pixel center exactly on an edge belongs to the triangle only by the top-left rule (the edge is a top edge or a left edge), so a pixel on an edge two triangles share is drawn once, not twice or not at all. That exactness matters here: a translucent mesh blended twice along its internal edges would show seams, and the screenshot references are compared pixel for pixel. Textures are plain arrays in memory. Given the same fonts, it produces the same pixels on every machine.

`HeadlessPlatform` (`backends/software/headless.c3`) plays a script of `Input` frames and then empty ones, at a fixed size and a fixed time step of 1/60 s. The clipboard and the primary selection live in memory, assets come from the file system, and the cursor, insets and `wake()` calls do nothing. The text entry calls are recorded for tests: `text_input_on` and `text_entry` hold what the last `text_input()` asked for, with the area `text_input_area()` moved it to, and `text_input_calls`, `text_input_area_calls` and `compositions_ended` count the calls. Its `wakes()` is its `can_wake` field, false unless a test sets it: then it stands in for a platform that sleeps until a deadline, and a waiting poll with one moves the clock straight to the deadline instead of one step. `software::platform_of` gives a test the headless platform of a `Ui`.

A `Ui` made by `ui::create` and fed its own `Input` gets the renderer alone from `software::attach_renderer`, as a game's headless tests do. `software::create_ui` attaches both, and `software::draw` draws the last frame over the theme's `SURFACE` color:

```
Ui* ui = software::create_ui(390, 844, script: inputs);
ui.begin();                    // plays the next scripted frame
// ...widgets...
ui.end();
software::draw(ui);
char[] rgba = software::renderer_of(ui).framebuffer();
```

Since the OpenGL and SDL_GPU renderers compute what the software renderer computes, the software renderer's references check them too. Where there is a display, `check.sh`'s "GPU matches software" step draws the gallery's charts page with each GPU backend, with `KILN_GALLERY_REFERENCE=1` (the light theme, no system fallback fonts, no motion), and `scripts/screenshot_compare.c3` (target `screenshot_compare`) compares the frame with the software reference. It allows a small difference per channel and a small share of changed pixels, for GPUs that round edges a little differently.

25.8 One thread

The UI runs on one thread. raylib, SDL and OpenGL need the main thread, and a frame is too small to be worth splitting. While a frame runs, between `begin()` and `end()`, its `Ui` is the thread's current one (a thread-local), which is how springs find the frame's time step (Motion).

26 Glass

Glass is a material for the floating navigation dock, not a general background effect. The gallery pattern is a capsule with a sliding active pill. On SDL_GPU the capsule can be the Stage 1 glass material; elsewhere the same paint falls back to the solid surface. This chapter records what the material borrows from Apple's Liquid Glass, what a small WebGPU recreation gets right, and how Kiln renders the live backdrop.

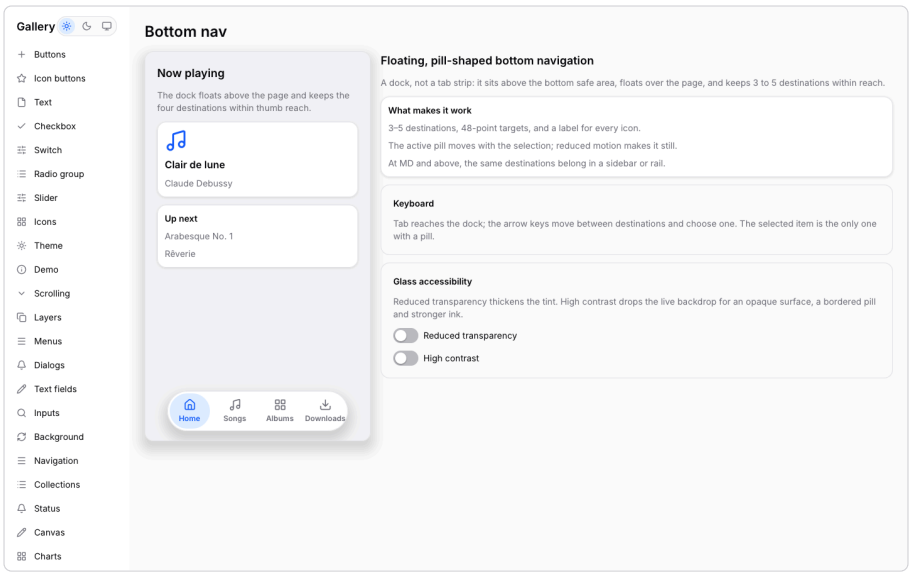


Figure 52: The Bottom nav gallery page. The software renderer shows the solid fallback; the SDL_GPU build can draw the same capsule as Stage 1 glass.

26.1 What Liquid Glass is

Apple describes Liquid Glass as a **functional layer** for controls and navigation that floats above content. It is not a blur applied to a panel. It bends and concentrates light (lensing), carries highlights and shadows that respond to geometry and motion, changes with its size and surroundings, and keeps text legible as content moves underneath it. Apple's Materials guidance and the Meet Liquid Glass session give the rules that matter here:

- Keep it in the navigation layer. Do not use Liquid Glass in the content layer; a table, a card or a page background is not a glass surface.
- Use it sparingly. It should bring attention to content, not compete with it.
- Do not put glass on glass. A control placed on glass uses fill, transparency or vibrancy instead of another glass layer.
- The **regular** variant adapts to the content behind it and flips light or dark for legibility. The **clear** variant is permanently more transparent, needs a dimming layer, and is only for media-rich backgrounds. The variants are not mixed.
- Tint only primary actions. A tinted surface still derives its tones from the content behind it.
- Reduced Transparency makes the material frostier; Increase Contrast gives it a strong border; Reduced Motion removes elastic behavior. These are accessibility settings, not decorations.

The material is live. Apple's tab bars shrink when the page scrolls and expand when it returns; scroll edge effects dissolve moving content under a floating bar; small elements flip their ink as the background changes. A static picture of the page cannot do any of that.

26.2 The WebGPU recreation

The small liquid-glass-webgpu project is a useful shader reference. It draws a pill-shaped lens over a web page and recreates the optical signature:

- a rounded-box signed distance field for the pill;
- coverage from the distance, and a lens bend that grows near the edge;
- a surface normal from the field's gradient;
- Schlick Fresnel for the rim;
- separate red, green and blue lookups for chromatic dispersion;
- an edge ring, a Blinn-Phong specular highlight and a rim term;
- a mask render target beside the color target.

It is not a foundation for Kiln. The project captures the page once with `html2canvas-pro` and samples that still image. Dragging works because the page behind is frozen. Scrolling, video, animated content and anything the capture library cannot render are outside it. It has no adaptive ink, no scroll edge effect, no accessibility settings and no performance budget. It is a lens shader, not a material system.

Kiln should borrow the math, not the capture. The backdrop must come from the live frame.

26.3 The render pipeline

The glass dock is a renderer feature, not a draw-list rectangle with a special paint callback.

1. **Scene pass.** Draw the page normally into an offscreen color texture.
2. **Backdrop pass.** Downsample and blur the dock's region into a smaller texture.
3. **Glass pass.** Draw the dock's rounded rectangle, sampling the blurred scene.
4. **UI pass.** Draw the icons, labels, active pill and focus ring on top.

The scene texture must be separate from the dock's target; sampling the target being written is a feedback loop. The blur samples only the dock's rectangle, so a full-screen glass pass is never needed.

A draw command can carry the material's parameters:

Field	Meaning
`rect`	the dock's rectangle
`radius`	the capsule's corner radius
`tint`	the regular variant's color, derived from the theme
`blur_radius`	the backdrop's blur, capped by the renderer
`refraction`	the SDF lens strength
`dispersion`	the edge's color spread
`rim`	the Fresnel and specular rim
`shadow`	the elevation's soft shadow
`luminance`	the backdrop's brightness, for ink selection

The draw list keeps its shapes, text and images. Only the dock's surface uses the new command.

26.4 Backends

SDL_GPU is implemented first: it already owns the render pass and the shader pipeline. The page before the dock is rendered to an offscreen scene texture; the scene is blitted to the swapchain,

and the glass command draws the capsule over it, sampling a blurred, lensed version of the scene. The scissor stays at the dock's bounds.

OpenGL follows with the same shader in GLSL and an FBO for the scene. If the FBO cannot be created, the dock uses the solid fallback.

The **software** renderer keeps a solid or lightly tinted capsule with the existing border and shadow. Screenshot tests therefore stay deterministic and do not depend on a GPU.

26.5 Material stages

Stage	Surface	Checks
1. Cheap glass	blur, tint, rim, shadow — implemented on SDL_GPU	legibility, cost, layout
2. Lensing	SDF bend, Fresnel, dispersion, specular rim — implemented on SDL_GPU	edge quality, motion, performance
3. Adaptive	ink from luminance, shadow from content, selective tint	contrast, reduced transparency, high contrast

Stage 1 is the version to ship first. It gives the dock depth without the cost or fragility of a full lens. Stage 2 adds the optical signature. Stage 3 makes the material behave like the platform's, rather than merely look like it.

26.6 Legibility and accessibility

A glass surface must remain readable over any content. The ink on the dock is chosen from the luminance behind it, not from the theme alone. The shadow grows over text and recedes over a solid light background. A clear variant, if it is ever offered, needs a dimming layer and a strong reason to exist.

The accessibility settings are part of the material, not a later patch:

- Reduced Transparency replaces the blur with a frosty, mostly opaque surface.
- Increase Contrast adds a strong border and high-contrast ink.
- Reduced Motion removes elastic lensing and makes the active pill settle at once.

The dock also keeps its 48-point targets, labels and focus ring. Glass does not change the hit area or the keyboard behavior.

26.7 Performance

The glass pass is scoped to the dock. The backdrop is downsampled before it is blurred, the blur has a fixed sample budget, and the pass is skipped when the dock is hidden. A full-screen lens, a per-frame page capture or a blur at device resolution is out of scope.

The first implementation should be profiled over a full-screen image, a dense text page and a scrolling list. If the material misses the frame budget, it falls back to the solid capsule rather than slowing the interface.

26.8 Integration

The gallery pattern remains the source of truth for layout and interaction. A `glass` option on the dock selects the material when the renderer reports support; otherwise the solid capsule is used. The C ABI is untouched while this stays a pattern. If the material is promoted, it belongs to the renderer, not to a new C widget: the dock's geometry, focus and selection already exist.

Status: Stages 1 to 3 are implemented on SDL_GPU. The dock paint emits a glass command; the renderer draws the page to a scene texture, blits it, then draws the capsule with a 9-tap blur, SDF lens, Fresnel rim, chromatic dispersion, specular highlight, tint and shadow. The tint and shadow follow the backdrop's luminance. Inactive dock labels and icons use an ink color chosen from the average backdrop under the capsule, so one dock keeps one ink; high contrast keeps the dock opaque with a bordered pill, and reduced transparency thickens its tint. OpenGL, raylib and the software renderer keep the solid fallback. Reduced transparency is an app setting until platforms expose one; the safe-area and scroll-edge behaviors remain.

Set `KILN_GALLERY_PHOTO` to a `;`-separated list, for example `examples/keithen-schwahn-mFjzQgNtXLU-unsplash.jpg;examples/vaclav-pechar-u-5wtZrq2rw-unsplash.jpg`, to see the dock over the Unsplash photographs used for local glass testing. Scroll the phone to move the stack under the dock; the glass samples the moving scene. The page credits Keithen Schwahn and Václav Pechar; the photos are not part of the screenshot references.

27 The operating system

Some things an app needs come from the operating system rather than from a window or a GPU: whether the user prefers a dark interface, their accent color, whether they asked for less motion, more contrast, larger text or scrollbars that stay shown, how far a notch of the mouse wheel scrolls, how long a tooltip or a submenu waits, how the text caret blinks, how quickly two clicks make a double-click, which character the user's region writes before a number's fraction, the order their language sorts text in, the system's own dialogs for choosing files, notifications, and opening links in the user's browser. `kiln::os` provides them. It is Kiln's own code, separate from the backends: it works the same with raylib, the software renderer or any other backend, and an app that never calls it doesn't carry it in an optimized build.

Typed text follows the same rule through the backend rather than `kiln::os`: an input method's candidate window and the on-screen keyboard are the OS's own. Kiln draws only the text being composed, inline at the caret, and tells the OS what each field holds and where it is (The interfaces).

27.1 One call, one implementation per OS

The calls are declared once, in `src/os/os.c3`, and each OS answers them in a file of its own: `linux.c3`, `windows.c3` and `macos.c3`, and `web.c3` in a browser (The web: the browser). Each file is a section of the same module marked with a feature, such as `module kiln::os @feat(LINUX);`, so a build compiles exactly one of them; `os::theme` picks it with a compile-time `$if`. There is no interface object and no virtual call, since a program only ever runs on the OS it was built for. On an OS with no implementation, every call fails with `os::UNSUPPORTED`.

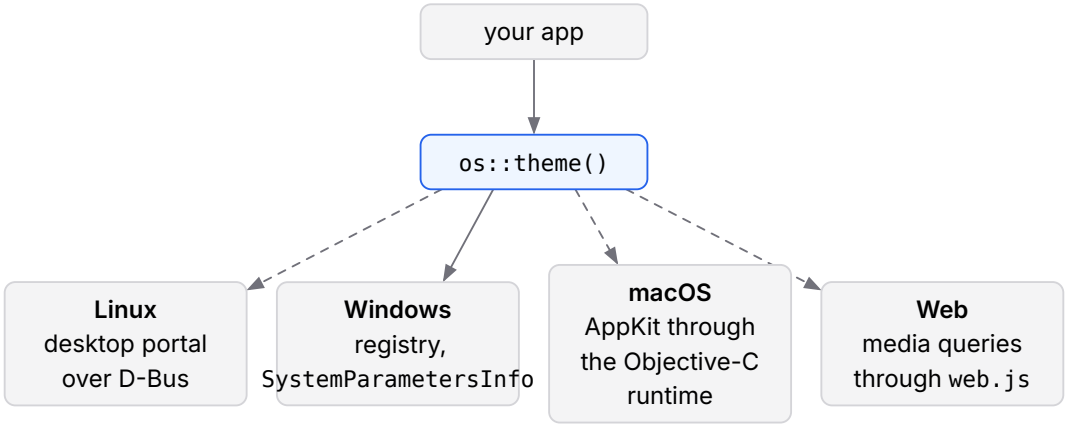


Figure 53: `os::theme()` and the four ways it is answered. The build compiles one of them; the app sees the same `SystemTheme` everywhere.

27.2 The system theme

`os::theme` returns a `SystemTheme`: the `ColorScheme` (dark, light, or no preference), the accent color as an `Oklch` when the OS has one, whether reduced motion and high contrast are on, whether scrollbars stay shown, how many lines one notch of the mouse wheel scrolls, how long the pointer rests before a tooltip shows or a submenu opens, how fast the text caret blinks (or that it shouldn't), how far apart two clicks may be to count as a double-click, how large the user wants text, and whether a middle click pastes the primary selection. Each part is read on its own, so a setting the OS doesn't have stays at its zero value (`has_accent` false, no preference) while the others are filled. The call fails only when the OS can't be asked at all.

Setting	Linux	Windows	macOS
Dark or light	portal `color-scheme`	registry `AppsUseLightTheme`	the `AppleInterfaceStyle` default
Accent	portal `accent-color`	registry `AccentColor` (DWM)	`NSColor`'s control accent color
Reduced motion	portal `reduced-motion`, else GNOME's `enable-animations`	the client-area animation setting	`NSWorkspace`'s reduce-motion setting
High contrast	portal `contrast`	the high-contrast setting	`NSWorkspace`'s increase-contrast setting
Scrollbars always shown	GNOME's Accessibility setting "Overlay Scrollbars" turned off (`overlay-scrolling`)	Settings > Accessibility > Visual effects, "Always show scrollbars" (registry `DynamicScrollbars` is 0)	System Settings > Appearance, "Show scroll bars" set to Always, or Automatically with a mouse (`NSScroller`'s preferred style is legacy)
Wheel step	no setting	Mouse settings, "Choose how many lines to scroll each time" (`SPI_GETWHEELSCROLLLINES`)	no setting
Tooltip delay	no setting (Kiln's 0.5 s)	Mouse settings' hover time (`SPI_GETMOUSEHOVERTIME`)	the `NSInitialToolTipDelay` default when it is set, else Kiln's 0.5 s
Submenu delay	no setting (Kiln's 0.2 s)	the menu show delay (`SPI_GETMENUSHOWDELAY`)	no setting (Kiln's 0.2 s)
Caret blink	GNOME's Accessibility setting "Cursor Blinking"	Keyboard settings' cursor blink rate	the `NSTextInsertionPointBlinkPer` default when it

Setting	Linux	Windows	macOS
	(`cursor-blink`) and its speed (`cursor-blink-time`, a whole on-off cycle, halved)	(`GetCaretBlinkTime`; `INFINITE` means no blinking)	is set, else Kiln's 0.5 s
Double-click time	GNOME's double-click delay (`org.gnome.desktop.peripherals.mouse double-click`)	Mouse settings' double-click speed (`GetDoubleClickTime`)	System Settings' double-click speed (`NSEvent`'s `doubleClickInterval`)
Decimal separator (`os::decimal_separator`)	the environment's `LC_NUMERIC` locale (`RADIXCHAR`)	Region settings' decimal symbol (`GetLocaleInfoEx`, `LOCALE_SDECIMAL`)	Language & Region's number format (`NSLocale`'s `decimalSeparator`)
Text size	GNOME's Accessibility setting "Large Text" or its scaling factor (`text-scaling-factor`)	Settings > Accessibility > Text size (registry `TextScaleFactor`)	no setting
Middle-click paste	GNOME Tweaks' Keyboard & Mouse setting "Middle Click Paste" (`gtk-enable-primary-paste`)	no setting (no primary selection)	no setting (no primary selection)

The wheel step is a number of lines, or "one screen at a time", which `SystemTheme.wheel_lines` reports as `uint::max`; it is 0 on an OS without the setting. The two delays are in seconds, also 0 without a setting. So are `caret_blink`, the time the caret stays on and then off, and `double_click_time`, the longest gap between the two clicks of a double-click; `caret_blink` is −1 when the user turned blinking off. `text_scale` is a factor, 1 for the OS's default size and 0 without a setting. `no_middle_click_paste` is true when the user turned off pasting the primary selection with a middle click (The primary selection); it is false without a setting. Accent colors arrive as sRGB and are converted to `Oklch`, the space Kiln's themes are written in (Theme), so a system accent can become a theme's `ACCENT` directly.

27.3 The decimal separator

`os::decimal_separator` returns the character the user's region writes between a number's whole and fractional parts, as a code point: `.` in English, `,` in German or French. It is 0 when the OS has no

setting. It is a call of its own rather than a field of `SystemTheme` because it comes from a different place: the locale, not the settings service that the theme's values come from, and it is read from the locale the app started with, not followed live. A user who changes their region sees the new separator the next time the app starts.

- **Linux:** `newlocale(LC_NUMERIC_MASK, "", …)` builds a locale object from the environment (`LC_ALL`, `LC_NUMERIC`, `LANG`) and `nl_langinfo_l(RADIXCHAR, …)` reads its separator. Kiln never calls `setlocale`, so the process's own locale, and with it `printf` and `strtod`, stays the C locale; a locale named in the environment but not installed gives 0.
- **Windows:** `GetLocaleInfoEx` with `LOCALE_SDECIMAL` for the user's default locale, which includes the decimal symbol the user chose in the region settings.
- **macOS:** `NSLocale`'s `currentLocale`, whose `decimalSeparator` reflects the number format chosen in Language & Region.
- **Web:** `Intl.NumberFormat` for the browser's language, the separator it writes in 1.1.

27.4 The calendar settings

`os::calendar_settings(allocator)` returns a `CalendarSettings`: the first day of the week (0 Sunday to 6 Saturday), whether the clock shows AM and PM, how a short date is written (its `DateOrder`, the mark between its parts and whether the day and month have a leading zero), the names of the months, the short month names and the short weekday names in the user's language, and how a time is written: the mark between its hour, minute and second, the words for AM and PM, and whether they come before the hour. The date picker and the time field read it (Dates, Times). Like the decimal separator, it comes from the locale the app started with and isn't followed live.

OS	First weekday	12-hour clock	Short date and names
Linux	`nl_langinfo_l` with glibc's `_NL_TIME_FIRST_WEEKDAY` and `_NL_TIME_WEEK_1STDAY`	`T_FMT` uses `%I`, `%l` or `%r`	`D_FMT`; `ALTMON_n` (else `MON_n`), `ABMON_n`, `ABDAY_n`
Windows	`LOCALE_IFIRSTDAYOFWEEK`	`LOCALE_STIMEFORMAT` uses `h`	`LOCALE_SSHORTDATE`; `LOCALE_SMONTHNAMEn`, `LOCALE_SABBREVMONTHNAMEn`, `LOCALE_SABBREVDAYNAMEn`
macOS	`NSCalendar`'s `firstWeekday`	`NSDateFormatter`'s format for the template `j` has `a`	the format for `yMd`; `standaloneMonthSymbols` and the short ones, `shortStandaloneWeekdaySymbols`
Android	`Calendar.getFirstDayOfWeek`	`DateFormat.is24HourFormat`	`getBestDateTimePattern` for `yMd`; ICU's standalone month names, `DateFormatSymbols`' short weekdays
Web	`Intl.Locale`'s week info	`Intl.DateTimeFormat`'s `hourCycle`	`formatToParts` of a date; `Intl.DateTimeFormat` names

OS	Time pattern	AM and PM
Linux	`T_FMT_AMPM` on a 12-hour clock, else `T_FMT`	`AM_STR`, `PM_STR`
Windows	`LOCALE_STIMEFORMAT`	`LOCALE_S1159`, `LOCALE_S2359`

OS	Time pattern	AM and PM
macOS	`NSDateFormatter`'s format for the template `jms`	`AMSymbol`, `PMSymbol`
Android	`getBestDateTimePattern` for `hms` or `Hms`	`DateFormatSymbols.getAmPmStrings`
Web	`formatToParts` of a time, as letters	the `dayPeriod` part of a morning and an evening time

The time pattern is read for two things only: the first `:` or `.` after the hour, and whether the day period (`%p`, `a`, `tt`) comes before the hour, as it does in Korean (`오전 3:30`). Letters in a strftime pattern are literal text, so `%Hh%M` has no day period.

On Linux, as for the decimal separator, `newlocale(LC_TIME_MASK, "", …)` reads the environment's `LC_TIME` without touching the process's locale. glibc gives the first weekday as a day counted from the date `_NL_TIME_WEEK_1STDAY` names (30 November 1997, a Sunday, in most locales); musl has neither item, so the fallback applies. The month names are the standalone forms where the OS has them, the nominative a calendar's title uses, not the genitive a date uses in languages such as Russian or Polish. What the OS doesn't give falls back to English names, Monday first, a 24-hour clock, `:`, `AM` and `PM` after the time, and `2026-11-25`.

The settings' strings live in one block in the allocator given, so `CalendarSettings.free` is a single free. `os::english_calendar` builds fixed English settings, which tests pin with `Ui.set_calendar`.

27.5 Sorting text

`os::collate` compares two strings the way the user's language sorts them, as the platform's file manager orders names: runs of digits compare as numbers, so "track 2" comes before "track 10", and case is ignored. Like `strcmp`, it returns a negative number when `a` goes first, zero when the two are equal, and a positive number otherwise. A table sorts its text columns through it (Tables).

- **Windows:** `CompareStringEx` with the user's default locale and `SORT_DIGITSASNUMBERS | LINGUISTIC_IGNORECASE`, the order of Explorer.
- **macOS:** `NSString`'s `localizedStandardCompare:`, the order of Finder.
- **Web:** `Intl.Collator` for the browser's language with `numeric: true`, so digit runs compare as numbers.
- **Linux and other systems:** the call fails with `os::UNSUPPORTED`. No desktop service compares strings, and glibc's collation (`strcoll`) compares digits one character at a time, so it would put "track 10" before "track 2".

Where `collate` fails, `Ui.compare_text` falls back to `text::natural_compare`, Kiln's own natural order: digit runs as numbers, case folded, anything else by code point, and finally the bytes, so two different strings never compare equal. It falls back the same way when the OS calls two different strings equal, such as "File" and "file", which keeps a sort stable from one frame to the next. Which comparison a `Ui` uses is `Ui.text_order`: `TextOrder` `SYSTEM` asks the OS first, `NATURAL` goes straight to Kiln's order. `ui::system_text_order` gives `SYSTEM` where `collate` works, and the `Ui` follows it while the theme follows the system (Following the system); `ui::create` keeps `NATURAL`, so a sorted table looks the same in a screenshot on every machine.

27.6 Linux: the desktop portal

On Linux the settings come from the XDG desktop portal, `org.freedesktop.portal.Settings`, which GNOME, KDE and other desktops provide over the D-Bus session bus. Kiln talks to it through libdbus, the reference D-Bus library, but doesn't link it: the first call loads `libdbus-1.so.3` with `dlopen` and looks up the few functions it uses. An app therefore starts on any Linux system, and

`os::theme` fails with `os::UNSUPPORTED` when libdbus, the session bus or the portal is missing, such as in a container or over SSH. Kiln never starts a session bus itself, and tells libdbus not to end the process if the bus goes away.

Each setting is one `ReadOne` call with a short timeout, falling back to the older `Read` on portals that lack it. The settings come from three namespaces: `org.freedesktop.appearance` for the portable ones, and GNOME's `org.gnome.desktop.interface` and `org.gnome.desktop.peripherals.mouse`, which the portal passes through from GNOME's own settings. Kiln asks the bus for the `SettingChanged` signals of all three, so a change to any of them arrives while the app runs.

27.7 Windows and macOS

Windows reads four registry values under the current user, among them `TextScaleFactor` for the text size, and asks `SystemParametersInfoW` for the animation, wheel, hover, menu and contrast settings (`SPI_GETCLIENTAREAANIMATION`, `SPI_GETWHEELSCROLLLINES`, `SPI_GETMOUSEHOVERTIME`, `SPI_GETMENUSHOWDELAY`, `SPI_GETHIGHCONTRAST`), and `GetCaretBlinkTime` and `GetDoubleClickTime` for the caret and the double-click. macOS reads the `NSInitialToolTipDelay` and `NSTextInsertionPointBlinkPeriodOn` user defaults (in milliseconds; macOS has no such settings in System Settings, but a user can set them with `defaults write`), `NSEvent`'s `doubleClickInterval`, `NSColor`'s `controlAccentColor` in the sRGB color space, `NSWorkspace`'s `accessibilityDisplayShouldReduceMotion` and `accessibilityDisplayShouldIncreaseContrast`, and `NSScroller`'s `preferredScrollerStyle`, in which macOS has already resolved Automatically for the attached mouse or trackpad. It calls AppKit through the Objective-C runtime, with `objc_msgSend` cast to a typed function pointer for each message, inside an autorelease pool; it needs no `NSApplication`, so it works in a Game-style program that owns its own loop.

`scripts/check.sh` type-checks both for their OS on every change; they run on those systems only.

27.8 The web: the browser

In a browser (On the web) the page stands in for the OS. `web.c3` asks it through one JavaScript library, `src/os/web.js`, which web builds link with emcc's `--js-library`. The direction is always the same: C3 asks, once a frame, and JavaScript never calls back into WebAssembly. An answer that comes later, such as a chosen file or a changed media query, waits in the library until C3 asks for it. Outside a page, as when the test suite runs as WebAssembly under node, every call fails with `os::UNSUPPORTED` instead of throwing.

The theme comes from media queries, which most browsers take from the desktop's settings:

Setting	Web
Dark or light	`prefers-color-scheme`
Accent	none: browsers report a fixed `AccentColor` rather than the user's, so a page can't use it to fingerprint them
Reduced motion	`prefers-reduced-motion`
High contrast	`prefers-contrast: more` or `forced-colors: active`
Decimal separator	the browser language's number format (`Intl.NumberFormat`)
Text size	the root element's font size over 16 pixels, the size CSS `rem` stands for, which the browser's font size setting changes
Others	no setting, so Kiln's defaults

The first call listens for each query's `change` event, so `os::theme_changed` answers at once after a change. The font size has no event, so the theme is also read again at most once a second and compared, as on Windows and macOS.

The other calls use the browser's own services: `Intl.Collator` for the text order (Sorting text), an `<input type="file">` or a save picker for file dialogs (File dialogs), the Notification API, `window.open` and the page title (Notifications, Progress on the taskbar or Dock, Opening a URL).

A browser opens a file picker, a new window or a notification prompt only during, or just after, a click or a key press: a user gesture. Kiln handles a press in the frame after it arrives, well within the time browsers allow, so a button that opens a dialog or a link works as on the desktop. A call made at any other time, such as from a timer, may be blocked.

The clipboard is the platform's, not `kiln::os`'s, but the browser limits it the same way. A page reads the system clipboard only in a paste event, which Ctrl+V sends, or asynchronously behind a prompt. So the raylib backend's web platform keeps the last text the page copied or pasted: Ctrl+V brings the system clipboard's text, and a context menu's Paste brings the last text copied or pasted in the page. Copying writes to the system clipboard with `navigator.clipboard.writeText` and keeps the text too.

27.9 Following the system

An app follows the OS unless it chooses otherwise. `themes::SYSTEM` is the default theme of `app::run` and `raylib::create_ui`, so an app that passes no theme starts in the user's dark or light setting with their accent color (`themes::from_os`). Fifteen behaviors follow the platform along with the theme:

- **Reduced motion:** with the setting on, springs jump to their targets instead of animating (Motion).
- **High contrast:** with the setting on, `Ui.high_contrast` is true, and status widgets such as badges, alerts and progress bars draw a strong outline that doesn't rely on their tint (Status).
- **Scrollbars:** with "always show scrollbars" on, a scroll area keeps its scrollbar visible whenever its content overflows; otherwise the scrollbar fades in while the area scrolls or the pointer is over it.
- **Wheel step:** one notch of the mouse wheel scrolls the user's number of lines (16 points each, `ui::WHEEL_LINE`), or a whole view when they chose a screen at a time (`ui::WHEEL_PAGE`). Without an OS setting, a notch scrolls `ui::WHEEL_STEP`.
- **Tooltip delay:** a tooltip shows once the pointer has rested on its widget for the user's hover time. Without an OS setting, it waits `ui::DEFAULT_TOOLTIP_DELAY` (0.5 s).
- **Submenu delay:** a submenu opens once the pointer has rested on its row for the user's menu delay (Menus, popovers and tooltips). Without an OS setting, it waits `ui::DEFAULT_SUBMENU_DELAY` (0.2 s).
- **Dialog order:** a dialog's confirming button comes first on Windows and KDE, and last on macOS, GNOME and most other Linux desktops, Android and the web, as each platform's own dialogs place it (Dialogs, sheets and toasts). This is a fixed convention, not a user setting, so `Ui.dialog_confirm_first` comes from the build target, and on Linux from `XDG_CURRENT_DESKTOP`, rather than from `os::theme`: it holds whatever the theme, and no OS change moves it.
- **Caret blink:** a focused text field's caret stays on, then off, for `Ui.caret_blink` seconds each, the user's blink rate; when they turned blinking off, or reduced motion is on, the caret stays steady (Text editing). Without an OS setting, each phase lasts `ui::DEFAULT_CARET_BLINK` (0.5 s).
- **Double-click time:** two clicks make a double-click, which selects a word in a text field (Text editing), when the second comes within `Ui.double_click_time` seconds of the first. Without an OS setting, the gap is `ui::DEFAULT_DOUBLE_CLICK` (0.4 s).
- **Text keys:** a text field answers to the platform's editing keys, `Ui.text_keys` (Text editing): `TextKeys` `MAC` on macOS, with Option and Command, and `PC` elsewhere, with Ctrl, Home and End. Like the dialog order, this comes from the build target, not from `os::theme`.

- **Decimal separator:** a number field (Inputs) shows and accepts numbers with `Ui.decimal_separator`, the user's region's separator from `os::decimal_separator`. Without an OS setting, it is `.`.
- **List placement:** a select (Inputs) opens its list where the platform's own does, `Ui.list_placement`: `ListPlacement` `OVER` on macOS, where a pop-up button opens its menu over itself with the chosen item under the pointer, and `BELOW`, under the button, elsewhere. Like the dialog order, this is a convention of the build target (`ui::platform_list_placement`), not a user setting.
- **Text order:** a table (Tables) sorts text with `Ui.compare_text`, in `Ui.text_order`: `TextOrder` `SYSTEM`, the order of `os::collate`, where the OS compares text, and `NATURAL` elsewhere (`ui::system_text_order`).
- **Text scale:** the whole UI grows or shrinks with the user's text size, `Ui.text_scale`, the `SystemTheme.text_scale` the OS reports. Text, controls and spacing scale together, as sizes in CSS `rem` do with a browser's font size, so larger text never overflows the control around it; the window just holds less. `Ui.begin` applies it as a zoom: it multiplies the pixels per point by the scale and divides the window size and the pointers' positions by it, so widgets lay out and hit-test in the same points at any scale. The scale is kept between `ui::MIN_TEXT_SCALE` (0.5) and `ui::MAX_TEXT_SCALE` (3). Without an OS setting, it is 1.
- **Middle-click paste:** on Linux and the BSDs a middle click in a text field pastes the primary selection (The primary selection), as `Ui.middle_click_paste` says: on unless the user turned it off (`SystemTheme.no_middle_click_paste`). Elsewhere there is no primary selection, and it is off.

The colors and the settings follow separately. Passing another theme, such as `themes::DARK`, or an app's own palette, stops only the colors following; the settings go on following the OS, since a user who asked for less motion or larger text asked it of every app, whatever its colors. Only pinning a setting (Overriding a setting) or `ui.follow_os_settings(false)` stops one. `ui.set_theme(themes::SYSTEM)` makes the colors follow again. `ui::create` and `software::create_ui` start in `themes::LIGHT` and follow nothing, at a text scale of 1, so tests and screenshots don't depend on the machine they run on; `themes::SYSTEM`, `Ui.set_themes` or `Ui.follow_os_settings` turns following the settings on.

An app with its own look usually has a light and a dark palette. `Ui.set_themes(light, dark, os_accent)` follows the OS with them: it applies the one that matches the user's light or dark setting, and the other when the user switches, as `themes::SYSTEM` switches between `LIGHT` and `DARK`. With `os_accent: true` the user's accent color lies over the palette's, its companions derived from it (`themes::with_os_accent`); without it, the app keeps its brand's accent.

```
ui.set_themes(glaze::LIGHT, glaze::DARK);   // the app's palettes, following the OS's light or dark
```

While anything follows the system, the `Ui` asks `os::theme_changed` at the start of every frame. The call doesn't block: on Linux it reads the portal's `SettingChanged` signals, which arrive on the D-Bus connection; on Windows and macOS it reads the settings again at most once a second and compares; in a browser it reads the media queries' `change` events, and the font size once a second (The web: the browser). After a change the `Ui` applies the new theme, reduced motion, high contrast, scrollbars, wheel step, delays, caret blink, double-click time, text scale and middle-click paste before the frame is built. The decimal separator is not followed this way: it comes from the locale the app started with. Nor is the text order: whether the OS can compare text doesn't change while the app runs. An App-style app that is asleep waiting for input updates on its next frame, as soon as the pointer moves or a key is pressed.

27.10 Overriding a setting

An app can pin any of the fifteen to its own value, whatever the platform says:

```
ui.set_high_contrast(true);            // a kiosk in bright sunlight, outlined whatever
the OS says
ui.set_always_show_scrollbars(true);   // a dense editor where scroll position matters
ui.set_wheel_step(ui::WHEEL_PAGE);     // a slide viewer: one notch, one slide
ui.set_tooltip_delay(0.2);             // a toolbar of unlabeled icons: hints come
quickly
ui.set_dialog_confirm_first(false);    // a game whose dialogs look the same on every
platform
ui.set_caret_blink(-1);                // a terminal-style console with a steady caret
ui.set_decimal_separator('.');         // a CAD tool whose numbers match the files it
exchanges
ui.set_text_order(NATURAL);           // a list of part numbers that sorts the same on
every machine
ui.set_text_scale(1);                  // a fixed-size kiosk screen laid out to the
pixel
ui.follow_os(SCROLLBARS);              // back to the user's setting
```

`Ui.set_reduced_motion`, `Ui.set_high_contrast`, `Ui.set_always_show_scrollbars`, `Ui.set_wheel_step`, `Ui.set_tooltip_delay`, `Ui.set_submenu_delay`, `Ui.set_dialog_confirm_first`, `Ui.set_caret_blink`, `Ui.set_double_click_time`, `Ui.set_text_keys`, `Ui.set_decimal_separator`, `Ui.set_list_placement`, `Ui.set_text_order`, `Ui.set_text_scale` and `Ui.set_middle_click_paste` each pin one setting, and `Ui.follow_os` with an `OsSetting` lets go of it, so it follows the OS again at once; `ui.follow_os(DIALOG_ORDER)`, `ui.follow_os(TEXT_KEYS)` and `ui.follow_os(LIST_PLACEMENT)` put the platform's convention back, `ui.follow_os(TEXT_ORDER)` puts `NATURAL` back, `ui.follow_os(HIGH_CONTRAST)` puts `false` back and `ui.follow_os(TEXT_SCALE)` puts 1 back while the theme doesn't follow the system. The value comes from the first of these that has one:

1. the app's choice, set with a pin;
2. the OS setting, while the theme follows the system;
3. Kiln's default: motion on, the usual contrast, scrollbars that fade in on use, `ui::WHEEL_STEP` per notch, the default tooltip and submenu delays, caret blink and double-click time, `.` between a number's whole and fractional parts, Kiln's natural text order (`TextOrder` `NATURAL`), a text scale of 1, middle-click paste on Linux and the BSDs, and the build target's dialog order, text keys and list placement.

A pin survives every OS change and every `set_theme`: the designer's choice always wins. Reduced motion, high contrast, the text scale and always-shown scrollbars are accessibility settings, though, and a user who turned them on usually needs them. Before pinning one off, prefer adapting the design: a calmer transition that still works without motion, colors that hold up with an outline, a layout that leaves room for a visible scrollbar, or one that scrolls when larger text makes it taller.

27.11 File dialogs

`os::open_file_dialog`, `os::save_file_dialog` and `os::choose_folder_dialog` show the system's own dialogs: open one or more files (with `FileFilter`s by name pattern), choose where to save (with a suggested name), or choose a folder. They don't block. Each returns a `FileDialog`, a small value the app keeps in its state and asks every frame:

```
if (ui.icon_button(FOLDER, "choose a sound"))
{
```

```
    app.dialog = os::open_file_dialog("Choose a sound", { { "Sounds", { "*.wav",
"*.ogg" } } });
}
if (try paths = app.dialog.result())   // once, when the user has chosen
{
    use(paths[0]);
    os::free_paths(paths);
}
```

`FileDialog.result` reports `os::PENDING` while the dialog is open, then the chosen paths exactly once, after which the handle is empty and reports `os::NO_DIALOG` a dismissed dialog reports `os::CANCELLED` once. The paths are file-system paths, copied into the allocator you pass and freed with `os::free_paths`. `FileDialog.is_open` says whether the dialog is still waiting, so a button can ignore clicks meanwhile, and `FileDialog.close` lets go of a dialog the app no longer wants, freeing its place. Because nothing blocks, the app keeps drawing and animating while the user browses; an app that stopped drawing for several seconds would get the desktop's "not responding" prompt on GNOME.

Each OS shows its native dialog:

- **Linux:** the desktop portal's `FileChooser` (`OpenFile` and `SaveFile`), which shows GNOME's, KDE's or another desktop's dialog. The call returns at once with a request object; the answer arrives later as that request's `Response` signal, with the chosen files as `file://` URIs, which Kiln decodes to paths. Kiln adds the signal's match rule before the call, from the request path the portal will use, so an answer can't arrive unwatched. One message pump reads the D-Bus connection for both the theme and the dialogs, so neither swallows the other's messages. The dialog isn't attached to the app's window.
- **Windows:** the Common Item Dialog (`IFileOpenDialog`, `IFileSaveDialog`) through COM, with folder picking as an option of the open dialog.
- **macOS:** `NSOpenPanel` and `NSSavePanel`, filtered by the patterns' extensions.
- **Web:** the browser's own pickers. Opening clicks a hidden `<input type="file">` (its `accept` built from the filters) and copies each chosen file into Emscripten's in-memory file system, at `/kiln/open/<id>/<name>`, so code that reads a path works unchanged; the input's `cancel` event gives `os::CANCELLED`. Saving uses `showSaveFilePicker` where the browser has it (Chromium-based browsers), and the user picks a file then; elsewhere the dialog answers at once with the suggested name, and the file becomes a download. A page can't see the user's folders, so `os::choose_folder_dialog` reports `os::UNSUPPORTED`.

On Windows and macOS the dialogs are modal: they run inside the call that opens them, so the app's frame loop waits until the user is done, and the first `result()` already has the answer. The same code works on every platform.

A page can't write to the user's disk, so the path a save dialog returns is only a name there. `os::save_file` writes data to such a path on every platform: on the desktop it is a plain file write (`file::save`); on the web it writes to the file the user picked, or starts a download named after the path, without waiting for either. An app saves through it rather than writing the file itself, as the gallery's SVG export and the dashboard's exports do:

```
if (try paths = app.export.result())
{
    if (@catch(os::save_file(paths[0], svg))) ui.toast("Couldn't save the chart.");
    os::free_paths(paths);
}
```

27.12 Notifications

`os::notify` shows a system notification with a title and a body. It returns once the OS has accepted it, without waiting for it to appear:

```
if (ui.button("Save", PRIMARY)) { save(); (void)os::notify("Saved", "Your changes are on disk."); }
```

- **Linux:** `org.freedesktop.Notifications`, which every desktop's notification server provides, called over the same D-Bus connection. Kiln waits for the server's reply, up to a second: GNOME Shell looks up the sender before it shows anything, and drops the notification when the program has already exited.
- **Windows:** a tray icon's balloon, which Windows 10 and 11 show as a toast in the corner and keep in the notification center. Kiln adds the icon, owned by a small hidden window, the first time an app notifies, and removes it when the app exits.
- **macOS:** `UNUserNotificationCenter`. macOS asks the user for permission the first time, and only lets an app in an app bundle notify: a program run on its own gets `os::UNSUPPORTED`.
- **Web:** the Notification API. The first notification asks the user for permission and shows once it is granted; when the user has denied it, the call fails with `os::UNSUPPORTED`.

A toast (Dialogs, sheets and toasts) can go out the same way. `Ui.toast` with `notify_if_unfocused: true` also sends its text through `os::notify` when it is raised while the window doesn't have the keyboard focus (`Input.unfocused`, The input of a frame), so a message raised while the user is in another app isn't missed. It does so only when the app asks: most toasts, such as "Saved", matter only to someone looking at the window.

27.13 Progress on the taskbar or Dock

`os::set_progress` shows how far a long task has come on the app's taskbar button or Dock icon, where the user sees it while working in another app. The value runs from 0 to 1; `os::PROGRESS_UNKNOWN` shows work whose end isn't known, and `os::PROGRESS_NONE` takes the progress away. It takes the window's handle, which the backend gives through `Platform.native_window()`: an `HWND` on Windows, an `NSWindow` on macOS, under SDL3 on Linux the Wayland surface or the X11 window's id, and null when there is no window. Linux's progress doesn't use the handle, but a handle tells the `Ui` there is a window to show progress for. An app rarely calls `os::set_progress` directly. `Ui.set_progress` takes the value each frame, rounds it to a percent and tells the OS only when that changes, and does nothing on a platform with no window, such as the headless one, so headless tests run unchanged. The hello example's Save drives it from its background job:

```
bool saving = demo.save_run != demo.save_done;
ui.set_progress(saving ? demo.save_progress : os::PROGRESS_NONE);
```

- **Windows:** the taskbar's `ITaskbarList3`, created through COM the first time. `SetProgressState` switches between no progress, indeterminate and normal, and `SetProgressValue` sets the value out of 1000.
- **macOS:** the Dock tile. Kiln sets its content view to an `NSImageView` of the app's icon with an `NSProgressIndicator` bar across the bottom and asks the tile to `display`; `PROGRESS_NONE` puts the plain icon back.
- **Linux:** the `Update` signal of `com.canonical.Unity.LauncherEntry`, sent over the same D-Bus connection, which KDE Plasma, Ubuntu's dock and Dash to Dock show on the app's launcher. The signal names the app by its desktop file, `application://<id>.desktop` (`os::launcher_uri`),

and carries `progress` and `progress-visible`. The id is the one given to `os::set_desktop_id`, else the executable's name, so an app installed as `org.example.Player.desktop` names it at startup. The signal has no indeterminate state, so unknown progress shows as none, and a desktop without such a dock ignores it. SDL3 sends the same signal for its own window progress, which Kiln doesn't use; unasked, SDL sends it only to reset the progress when it creates the window, so it never undoes Kiln's.

- **Web:** the page title, with the percentage in front of it, such as "45% Gallery", or "…" for unknown progress; `PROGRESS_NONE` puts the title back. The tab shows it while the user is on another one.
- **Other systems:** the call fails with `os::UNSUPPORTED`.

27.14 Opening a URL

`os::open_url` hands a URL to the user's default handler: the browser for `http` and `https`, the mail app for `mailto`, the file manager or a viewer for `file`. Only those four schemes are accepted, in any case; anything else fails with `os::UNSUPPORTED`, so text from a document or a server can't make the app launch some other handler. Linux uses the desktop portal's `OpenURI` and waits for its reply, for the same reason as notifications; Windows uses `ShellExecuteW`, macOS `NSWorkspace`, and the web `window.open`, in a new tab. None waits for the browser to start.

27.15 Trying it

`c3c run os_services -- notify "Title" "Body"` shows a notification and `c3c run os_services -- url https://c3-lang.org` opens a link. `c3c run os_dialogs -- open` (or `open-many`, `save`, `folder`) shows a dialog and prints what you chose. `c3c run os_theme` prints what the OS reports about its theme, such as the lines below; `c3c run os_theme -- --watch` keeps running and prints them again on every change.

```
color scheme:    DARK
accent:          oklch(0.614 0.164 255.3), #3584e4
reduced motion:  no
high contrast:   no
scrollbars:      shown on use
```

28 Testing

Three pieces show what Kiln looks like, and the tests check that it keeps looking that way. Each piece has one job:

Piece	Inspired by	Purpose
`Ui.show_demo`	Dear ImGui	A built-in panel with every widget, live, and a **theme editor** for tuning the running app's theme. Easy to put behind a debug key in an app or a game.
Gallery (`examples/gallery`)	GTK widget factory, libadwaita demo	Every component in every state, light and dark, at desktop and phone widths. The source of the **screenshot tests**.
Game example (`examples/game`)	CUI, Kirigami Gallery	A game's menus and HUD, built with real code over a raylib loop.

28.1 The built-in demo

One call shows a panel with every widget and a theme editor, in any app or game:

```
if (ui.shortcut(F1)) show_demo = !show_demo;
if (show_demo) ui.show_demo(&show_demo);   // the panel's close button clears the flag
```

The panel is a card, placed wherever the call sits in the layout. Without the `open` argument it has no close button. A radio group switches between two sections:

- **Widgets:** buttons (primary, default, ghost and an icon button) that count clicks, a checkbox, a switch, a radio group, a slider, text, an icon and a link.
- **Theme:** edits the running app's theme as you drag: the accent's OKLCH hue, chroma and lightness, light or dark, the spacing base, the control and panel radii, and reduced motion. The accent's companions (hover, pressed, subtle, on-accent, focus ring) are derived again from the new accent (Theme). Turning Dark on or off switches to the built-in light or dark theme with the edited accent, spacing and radii; switching back to the starting mode returns to the starting theme. Below the controls, the accent, spacing and radii are shown as `Theme` fields, ready to paste into a theme of your own. Reset puts the accent, light or dark, spacing, radii and reduced motion back to their starting values, those of the theme in use when the demo first appeared; the accent companions are derived again.

The demo keeps its state under its own id and keeps that state alive itself, so the edits survive frames in which the panel is hidden. Unoptimized builds keep its code even when an app never calls it; optimized builds drop it.

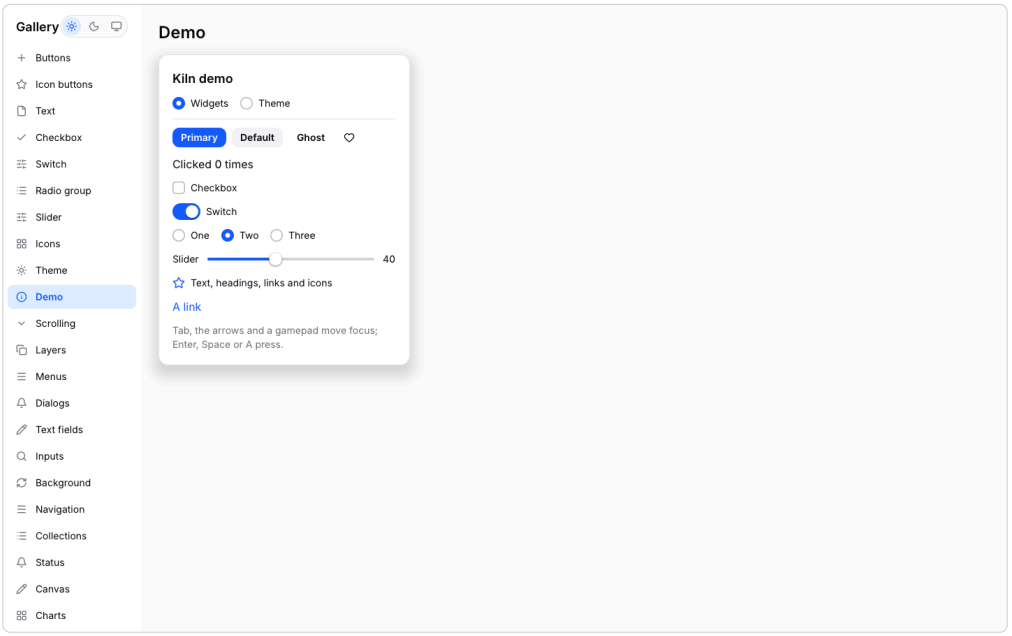


Figure 54: The demo page of the gallery: `ui.show_demo()` showing its Widgets section. The radio group at the top of the panel switches to the theme editor.

28.2 The game example

`c3c run game` is a whole small game in Game style (App style and Game style): your raylib loop runs a top-down arena (collect coins and dodge the red squares until the timer runs out), and Kiln draws the main menu, settings, pause menu, HUD and game-over screen over it. Every screen works with the mouse, the keyboard and a gamepad:

- Each menu is a **focus scope**, so focus lands on its first button when it appears and Escape or gamepad B leaves it; the buttons sit in a vertical focus group for the d-pad (Input and focus).
- The game reads its movement keys only when `Ui.wants_keyboard` is false, so keys that move focus in a panel don't also move the player.
- F1 opens `ui.show_demo()` beside the arena; its theme editor restyles the menus while the game runs.

`KILN_GAME_SCREEN=menu|play|pause|settings|over|demo` opens a screen at once, for screenshots. Every screen except `menu` starts a round and plays it for a moment, so the HUD and the arena have something to show; `demo` is a round with the demo panel open.

28.3 The gallery

`c3c run gallery` opens one page per component: buttons, icon buttons, text, checkbox, switch, radio group, slider, icons, theme, demo, scrolling, layers, menus and dialogs. Ctrl+1 to Ctrl+9 and Ctrl+0 (Cmd on macOS) switch to the first ten pages, in order. `KILN_GALLERY_PAGE=<name>` opens a page by its name (`buttons`, `icon_buttons`, `text`, `checkbox`, `switch`, `radio`, `slider`, `icons`, `theme`, `demo`, `scrolling`, `layers`, `menus`, `dialogs`, in any case); the pages live in `examples/gallery/pages.c3`, without a backend, so the screenshot tests can draw them headlessly. `c3c run gallery` builds it on SDL3, and `c3c run gallery-raylib` opens the same gallery on raylib (App style: Kiln owns the loop).

The layers page shows the floating layers of Layers: a menu (Menus, popovers and tooltips) anchored under its button, a popover asked to open to the right of a button at the window's right edge, which flips to the left and shifts to stay inside, a hint that takes no focus, and a dialog (`Ui.@dialog`, Dialogs, sheets and toasts) whose color button opens a second menu above it. Each layer's `open` flag is a field of the `Gallery` struct, so a test opens the menu or the dialog by setting `menu` or `dialog` before the first frame.

The menus page shows the widgets of Menus, popovers and tooltips: a File menu with shortcut labels, a disabled row, two checkable rows and a submenu with a submenu of its own, beside a menu button and an icon menu button; a popover with an arrow and one without; and icon buttons with tooltips. A line under the menus names the last row chosen. Its File menu and its arrow popover open the same way, through the `file` and `arrow_popover` fields.

The dialogs page shows the dialogs and toasts of Dialogs, sheets and toasts: buttons that open a plain dialog, a destructive one, one with a single button and one holding a switch and a slider, with a line naming how the last dialog was answered; and buttons that raise a short toast, one with an Undo action and one long enough to wrap. The gallery draws the toasts with `Ui.toasts` after every page, so they stay on screen when the page changes. Its destructive dialog opens through the `remove` field.

Each page shows one panel in the current theme. A segmented switcher in the sidebar's header picks it: Light, Dark, or System, which follows the OS (Following the system) and is where the gallery starts. The switcher is a custom widget built as Widgets describes, in `examples/gallery/switcher.c3`: one id and one `interact` per segment, springs for its hover and focus ring, and a paint function for the pill and its sliding highlight. Tab enters at the chosen segment and the arrows choose, as in a radio group. The choice applies with `Ui.set_theme` at the start of the next frame, so every frame is built in one theme.

A widget page is a **state matrix**: a row per state. The rows come from one table:

```
const StateRow[5] STATES = {
    { "Default", {}, false },
    { "Hover", { .hovered = true }, false },
    { "Pressed", { .hovered = true, .held = true }, false },
    { "Focus", { .focused = true, .ring = true }, false },
    { "Disabled", {}, true },
};
```

The switch figure in The five-step pattern shows the matrix at desktop width. Below the MD breakpoint the sidebar moves above the page, for phone widths (Layout).

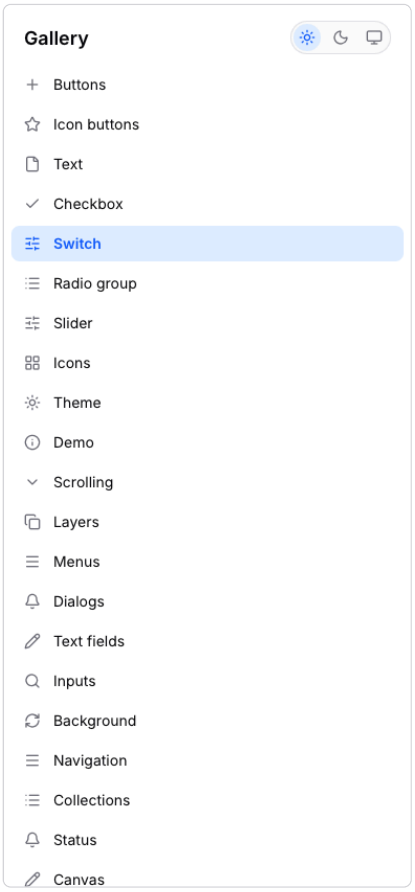


Figure 55: The switch page at 390 × 844: the sidebar, with the theme switcher in its header, sits above the page.

The same frame code builds both layouts. When a page is taller than the screen, it scrolls. The screenshot tests draw a few pages at phone width too.

28.3.1 Showing states without input

The scope `Ui.@show_state` makes the matrix possible without a pointer or a keyboard:

```
ui.@show_state({ .hovered = true, .held = true })   // shown hovered and pressed
{
    ui.button("Save", PRIMARY);
};
```

It ORs `hovered`, `held`, `focused` and `ring` into what `Ui.interact` returns (Widgets) and sets `Interact.shown`. Nothing fires: no clicks or activations come of it, real focus and pointer capture stay as they are, and a disabled widget stays disabled, since `interact()` returns nothing for it before the shown state applies. Widgets don't change the cursor for a hover that is only shown. Scopes nest; the innermost one counts.

28.4 Tests

Tests drive the core headlessly, with no window and no GPU. `Ui.begin` takes an `Input` you fill in (pointers, keys, gamepad, time step), so a test can script a click frame by frame. `software::create_ui` builds a `Ui` with no window that plays those frames back (Backends). A test sets `Input.unfocused` to play a window in the background, and replaces `Ui.notify_os`, the `NotifyFn` a toast sends system notifications through (`os::notify` by default, Notifications), with a function that counts them, so it checks a toast's notification without showing one on the desktop.

28.4.1 Screenshot tests

Screenshot tests check how things look. They draw with the software renderer, which gives the same pixels on every machine given the same fonts (Backends), so the tests turn off the system's fallback fonts (the gallery shows only text Inter has) and turn on reduced motion, so springs are at rest at once (Motion).

Each case is a gallery page in a theme at a size: every page in light and in dark at desktop size (1440 × 900), and a few at phone width (390 × 844). The layers, menus and dialogs pages are drawn only with a layer open. The layers page has its menu open, and its dialog open, in both themes, and its menu open at phone width; the menus page has its File menu open, and its popover with an arrow open, in both themes; the dialogs page has its destructive dialog open, and two toasts showing, in both themes. A case picks its theme by setting the gallery's switcher to Light or Dark, never to System, so the pictures don't depend on the machine. The test opens the page, runs ten frames, draws the last one, and compares the framebuffer with its reference PNG, `test/screenshots/<page>-<theme>-<width>x<height>.png`, such as `switch-dark-1440x900.png`. A case with a layer open adds the layer to the page's name: `<page>-menu-<theme>-<width>x<height>.png` or, with the dialog or the popover open, `<page>-popup-…`, such as `layers-popup-light-1440x900.png`. The inputs page names its open color picker `inputs-color-…`. The dialogs page names its destructive dialog `dialogs-menu-…` and its toasts `dialogs-popup-…`; the toasts are raised after the first frame, as an app raises them from a button.

A further test drives the switcher itself from the keyboard: Tab, then Right, chooses Dark, and the dark theme applies from the next frame.

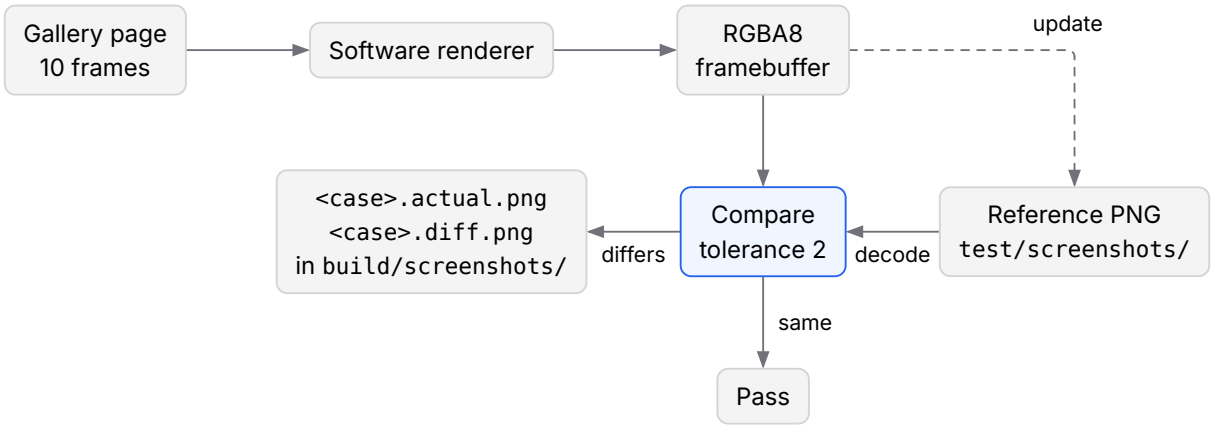


Figure 56: How one screenshot case runs. With `KILN_UPDATE_SCREENSHOTS=1` set, the dashed path replaces the comparison: the framebuffer is written as the reference.

- **Comparison:** both images are RGBA8. A pixel has changed when any of its four channels differs by more than 2 (out of 255). A case fails when any pixel changed or the sizes differ, or when its reference is missing.
- **On failure:** the test reports the number of changed pixels and the largest difference, and writes what it drew and a diff image to `build/screenshots/`. The diff image is the reference faded to light grey with every changed pixel in solid red. When the sizes differ there is no diff image; the test reports both sizes.
- **The references are approved images:** the dashed path in the figure writes them, and a person looks at them before committing them (CONTRIBUTING.md gives the steps). A reported change is either a bug (fix the code, and the old references pass again) or intended (regenerate, review the changed PNGs, and commit them with the change).

```
struct Comparison
{
    bool same_size;
    usz changed;       // pixels where some channel differs by more than the tolerance
    int max_delta;     // the largest channel difference anywhere
}
```

What each page costs is checked the same way, in numbers rather than pixels: `test/screenshot/counts_test.c3` records, per gallery and dashboard page, the allocations of its first ten frames and one steady frame's vertices, draw commands, Clay elements, shaped texts and atlas glyphs, and compares them exactly with `test/counts.csv`. They don't depend on the machine, so a change in them is a change in the code; `KILN_UPDATE_COUNTS=1` rewrites the file, to be reviewed and committed like reference images. Times do depend on the machine, so they live in the benchmarks instead (Performance).

The tests compare the software renderer with its own earlier output, never with a GPU: GPU output differs between drivers and needs a window. An App-style app's screenshots (`KILN_SCREENSHOT=<file.png>`, taken after a few frames, on either backend) are for people, such as `KILN_GALLERY_PAGE=buttons KILN_SCREENSHOT=out.png c3c run gallery`, and for the check below.

28.4.2 GPU renderers against the software renderer

The references are the software renderer's, and the software renderer computes what the shader does (One shape, one quad), so a GPU frame of the same page should match them to within rounding. `scripts/check.sh` holds each GPU renderer to that where there is a display: it runs `gallery` (SDL3, SDL_GPU on Linux), `gallery-raylib` and, when the OS has SDL3, `gallery-system` on the charts page at 1440 by 900 and compares each screenshot with `test/screenshots/charts-light-1440x900.png`. It runs the same page through OpenGL with `KILN_RENDERER=gl`, and through SDL_GPU where a Vulkan device exists (the default on Linux). It also points the Vulkan library and the ICD variables at nothing and checks that the hidden SDL_GPU window is replaced by an OpenGL one that still draws the same page.

`KILN_GALLERY_REFERENCE=1` makes the gallery draw as the screenshot tests do: the light theme, no system fallback fonts and no motion, so the window shows what the reference shows. The check sets it with `KILN_GALLERY_PAGE=charts`, `KILN_WINDOW=1440x900` and `KILN_SCREENSHOT`, then hands both images to `build/screenshot_compare` (`scripts/screenshot_compare.c3`). Its comparison is looser than the tests': a pixel has changed when a channel differs by more than 4, and the images match when at most one pixel in a thousand changed, slack for GPUs that round edges a little differently. It writes a `.diff.png` beside the screenshot, in the tests' red-on-grey style, and exits 1 when they don't match.

The same script checks the backends in three more ways: without a display, `hello` and `hello-raylib` end with a message and exit code 1 rather than a crash; each app carries one backend, so `nm` finds no raylib symbols in the SDL3 builds (`hello`, `gallery`, `dashboard`) and no SDL symbols in the raylib ones (the `-raylib` builds, `overlay` and `game`); and `scripts/sdl3_symbols.sh` checks that the built SDL3 library defines every function `lib/sdl3.c3l` declares, which would otherwise compile and then fail to link.

28.5 Frame statistics

`Ui.stats` reports the last finished frame. The `Ui` wraps the allocator given to `ui::create` in a counter, so it sees every allocation the core makes through its allocator (the backends and the C font code's own mallocs are not counted):

```
struct Stats
{
    usz allocations;    // acquires and resizes in the frame
    usz bytes;          // bytes they asked for
    usz frees;          // releases in the frame
    usz shaped_texts;   // entries in the shaped-text cache
    usz atlas_glyphs;   // glyphs in the atlas
    usz atlas_pages;    // glyph atlas textures
    usz vertices;       // in the draw list
    usz commands;       // draw commands in the draw list
    int elements;       // Clay elements opened
    // Where the frame's time went, in milliseconds:
    float build_ms;     // from `Ui.begin` to `Ui.end`: the app's widget code
    float layout_ms;    // Clay's layout, and reading the rects back
    float paint_ms;     // painting the draw list, canvases tessellated
    float end_ms;       // the rest of `Ui.end`: scrolling, focus, layers, collecting
state
    float draw_ms;      // `Ui.draw`: the atlas upload and the renderer (CPU time; the
GPU works on after)
    uint draw_calls;    // `Ui.draw`: the renderer's draw calls (a GPU renderer's; 0 for
the software one)
    uint gl_calls;      // and every graphics call it made, draws included
}
```

After warm-up, a steady frame makes no allocations (No allocations per frame). A test holds the library to it: it opens every gallery page at desktop size in both themes, runs ten frames, then checks that each of the next thirty frames reports zero allocations.

29 Performance

Kiln measures what a frame costs in three ways, each with its own job:

Measure	Where	Purpose
`Ui.stats`	Every app, every frame	Where the last frame's time went, phase by phase, with its allocations and draw list sizes. Always on, for an overlay or a log.
Benchmarks (`scripts/bench.sh`)	A developer's machine	Times for every gallery and dashboard page, a few focused cases, real GPU frames and binary sizes, compared with a committed baseline.
Counts test (`test/screenshot/counts_test.c3`)	The test suite	Numbers that don't depend on the machine (allocations, vertices, draw commands, elements, shaped texts, glyphs) compared exactly, so a change in what a page costs shows up as a diff.

Times depend on the machine and vary from run to run, so they live in the benchmarks, which report and never fail. The counts are the same everywhere, so the tests hold them exactly.

29.1 Where a frame's time goes

A frame runs from `Ui.begin` to `Ui.end`, then `Ui.draw` hands it to the renderer (The frame). The `Ui` times it in five phases with a monotonic clock: each phase ends with one clock read, which allocates nothing, so the timing is always on and costs a few reads a frame.

Phase	From and to
`build_ms`	From `Ui.begin` to the start of layout in `Ui.end`: the app's widget code, with the `Ui`'s own housekeeping at the start of the frame.
`layout_ms`	Clay's layout (Layout), reading the rects of the frame's elements back, and laying out layers (Layers) and scroll areas.
`paint_ms`	Painting Clay's commands into the draw list (Rendering), canvases tessellated (Canvas).

Phase	From and to
`end_ms`	The rest of `Ui.end`: scrolling, focus and keyboard navigation, dismissing layers, collecting unused state.
`draw_ms`	`Ui.draw`: uploading new glyphs to the atlas, resolving text, and the renderer's work. This is CPU time; with a GPU renderer the GPU works on after it returns.

The first four are Kiln's own work, independent of the renderer; their sum is what the benchmarks call **ui**. They sit in `Stats` beside the frame's allocations and the sizes of its draw list and caches (Frame statistics):

```
struct Stats
{
    usz allocations;    // acquires and resizes in the frame
    usz bytes;          // bytes they asked for
    usz frees;          // releases in the frame
    usz shaped_texts;   // entries in the shaped-text cache
    usz atlas_glyphs;   // glyphs in the atlas
    usz atlas_pages;    // glyph atlas textures
    usz vertices;       // in the draw list
    usz commands;       // draw commands in the draw list
    int elements;       // Clay elements opened
    // Where the frame's time went, in milliseconds:
    float build_ms;     // from `Ui.begin` to `Ui.end`: the app's widget code
    float layout_ms;    // Clay's layout, and reading the rects back
    float paint_ms;     // painting the draw list, canvases tessellated
    float end_ms;       // the rest of `Ui.end`: scrolling, focus, layers, collecting
state
    float draw_ms;      // `Ui.draw`: the atlas upload and the renderer (CPU time; the
GPU works on after)
    uint draw_calls;    // `Ui.draw`: the renderer's draw calls (a GPU renderer's; 0 for
the software one)
    uint gl_calls;      // and every graphics call it made, draws included
}
```

`draw_calls` and `gl_calls` count the renderer's work in `Ui.draw`: the draw calls a GPU renderer issues, and every graphics call it makes, draws included. Each draw call has a fixed cost on the driver whatever it draws, so the count says how well the renderer batches. The software renderer makes no graphics calls and leaves both at zero.

29.1.1 Reading the statistics in an app

`Ui.stats` returns the last finished frame, its draw included: `Ui.end` fills in the four phases, and `Ui.draw` adds `draw_ms` and the call counts after it. Called while a frame is being built, it describes the frame before, so an app can show its own cost in a corner without measuring itself:

```
Stats s = ui.stats();
float own = s.build_ms + s.layout_ms + s.paint_ms + s.end_ms;
ui.textf("%.2f ms (build %.2f, layout %.2f, paint %.2f) + draw %.2f ms, %d vertices, %d
allocations",
    own, s.build_ms, s.layout_ms, s.paint_ms, s.draw_ms, s.vertices, s.allocations);
```

A frame of a gallery page takes a tenth of a millisecond or so of Kiln's work on a desktop machine (Steady frames), so the numbers jump around in the second decimal from frame to frame; an overlay reads better with a spring or a running average over the last second than with the raw

value. Allocations above zero in a steady frame point at something that grows each frame (No allocations per frame).

The numbers are only as steady as the frames: in App style (App style: Kiln owns the loop) an app that is not changing sleeps, and the first frame after a sleep is often slower, with cold caches. For timing a page, both App-style backends draw frames back to back on request (Real frames on a GPU).

29.2 Benchmarks

`scripts/bench.sh` builds the `bench` target of `project.json` (`bench/main.c3`) and runs it. Everything it measures is built as an app ships: the `bench` target and each example's `-release` target (`hello-release`, `gallery-release` and so on) are optimized at O3 without panic messages, while development builds and the tests stay at O0, where times say little. Nothing is built above O3: O5's fast maths folds `is_nan` to false, and charts use NaN for their gaps (Charts).

The program runs every gallery and dashboard page headlessly, and the dashboard's overview a second time over its longest period, a year, where its charts hold the most points. It runs them at 1440 × 900 with the software renderer (Backends), Kiln's own fonts and no system fallbacks, reduced motion and background jobs done inline (Background work), so a run repeats itself from machine to machine as closely as times can. For each page it measures:

- **A cold first frame:** a new `Ui` and new app state, the page built and drawn once. This frame shapes all its text and fills the atlas.
- **Steady frames:** after 20 frames of warm-up, 200 measured frames, reported as the median and the 95th percentile of each phase, of **ui** (build, layout, paint and end together) and of **draw**.

draw here is the software renderer, which fills every pixel on the CPU so that the screenshot tests get the same pixels everywhere (Screenshot tests). It is far slower than a GPU and dwarfs Kiln's own work, so the steady numbers worth watching are the phases; the real cost of drawing comes from the GPU frames below.

Focused cases follow, each for one thing that can grow with the data rather than with the page:

- **Shaping** a paragraph of about 650 characters that no cache has seen (Caches): Latin, Greek and Cyrillic, with kerning and ligatures.
- **A long list** of 10,000 rows (Lists), scrolled a notch every frame, so new rows come into view each time.
- **Charts** (Charts), tessellated every frame: a line of 5,000 points; the same line zoomed to a fixed x range of 250 points, a twentieth of it; a line of 50,000 points, far more than the chart has pixels across; and area charts of 365 and 5,000 points, filled down to the axis.
- **A long text field** holding 2,000 characters, focused, either typing a character and deleting it on alternate frames, or selecting by dragging the pointer back and forth.
- **500 checkboxes** in a column: many small widgets, each with a label.
- **12,000 widgets at once**, for what a whole frame costs when an app builds a large UI in full rather than through a long list: a settings page of 3,000 rows in a scroll area, each a label, a text field, a checkbox and a button, mostly off screen; and a grid of 120 × 100 small text cells, all in view. Each prints its phases beside Clay's element count and the steady frames' allocations, which stay at none.
- **Making and destroying a `Ui`:** fonts, atlas, Clay and the theme.
- **The first frame of mixed scripts:** a new `Ui` with the system's fallback fonts on (Fonts), showing one line in Hebrew, Arabic, Chinese, Japanese, Korean, Thai, Devanagari and symbols. The time covers finding, loading and placing the fallback fonts, so unlike every other case it depends on the fonts the machine has.

The program prints a table and writes every number to `build/benchmarks/latest.csv`, one row per case and metric: `case,metric,median,p95,unit`. The script then adds two kinds of rows the headless program cannot measure:

- **Real GPU frames**, where there is a display: the gallery's charts, collections and text pages and the dashboard's overview, each run from its `-release` build on both backends with `KILN_FRAME_TIMES` (Real frames on a GPU): `gallery` and `dashboard` on SDL3, `gallery-raylib` and `dashboard-raylib` on raylib. The script keeps the median of each column, the startup time, and the draw and GL calls per frame, as `gpu/<app>/<page>` rows in the unit `calls`.
- **Peak memory**, where there is a display: the dashboard's peak resident memory (`VmHWM`) after 300 frames and after 3,000, on each backend, as `memory/<app> <frames> frames` rows in MB, for `dashboard` and `dashboard-raylib`. The two stay within a few hundred kilobytes; if the second is more than 5 MB higher, something a frame allocates is never freed, and the script says so.
- **Sizes:** each example from its `-release` build, stripped, as `size/<app>`: `hello`, `gallery` and `dashboard` on SDL3, `hello-raylib` and `gallery-raylib` on raylib, `game` and `overlay`, and `hello-system` and `gallery-system`, which link the OS's SDL3 and are built as they are, without a `-release` target; and what it ships, as `size/<app> shipped`: the executable plus raylib's shared library when it links one, both stripped. Then the web gallery's `.wasm` and `.js` together, as a browser downloads them (On the web): gzipped at level 9 (`size/web gallery`) and with Brotli at level 11 (`size/web gallery br`). The web sizes need Emscripten, and the Brotli one the `brotli` tool; without them those rows are left out.

Last, it compares the medians with `bench/baseline.csv`, case by case, and marks each change of more than 10% with a star. The baseline's first lines name the machine, the compiler and the commit it was measured on; a run on another machine says so above the comparison, since its numbers can't be compared one for one. The script never fails: a star is a prompt to look, not a verdict.

```
scripts/bench.sh          # run and compare with the baseline
scripts/bench.sh --save   # run, then make the results the new baseline, and add them to
the history
```

Save a baseline after a deliberate change to what a frame costs, a faster layout or a heavier widget, measured on the same machine as the baseline it replaces, and commit it with the change. A baseline from another machine makes every comparison after it say less.

Each save also adds its results to `bench/history.csv`, one row per case and metric, with the date, the machine and the commit (`git rev-parse --short HEAD`). A baseline is usually saved just before the change is committed, so its row names the commit the change was made on top of and says it had uncommitted changes; the next commit is the one that holds them. The history only grows, so how a case's cost moved over time is one `grep` away:

```
grep 'line chart of 5000 points,ui' bench/history.csv
```

29.3 The baseline

The charts in this section are drawn from the committed `bench/baseline.csv` as the book is built, so they show whatever baseline is in the repository.

29.3.1 Steady frames

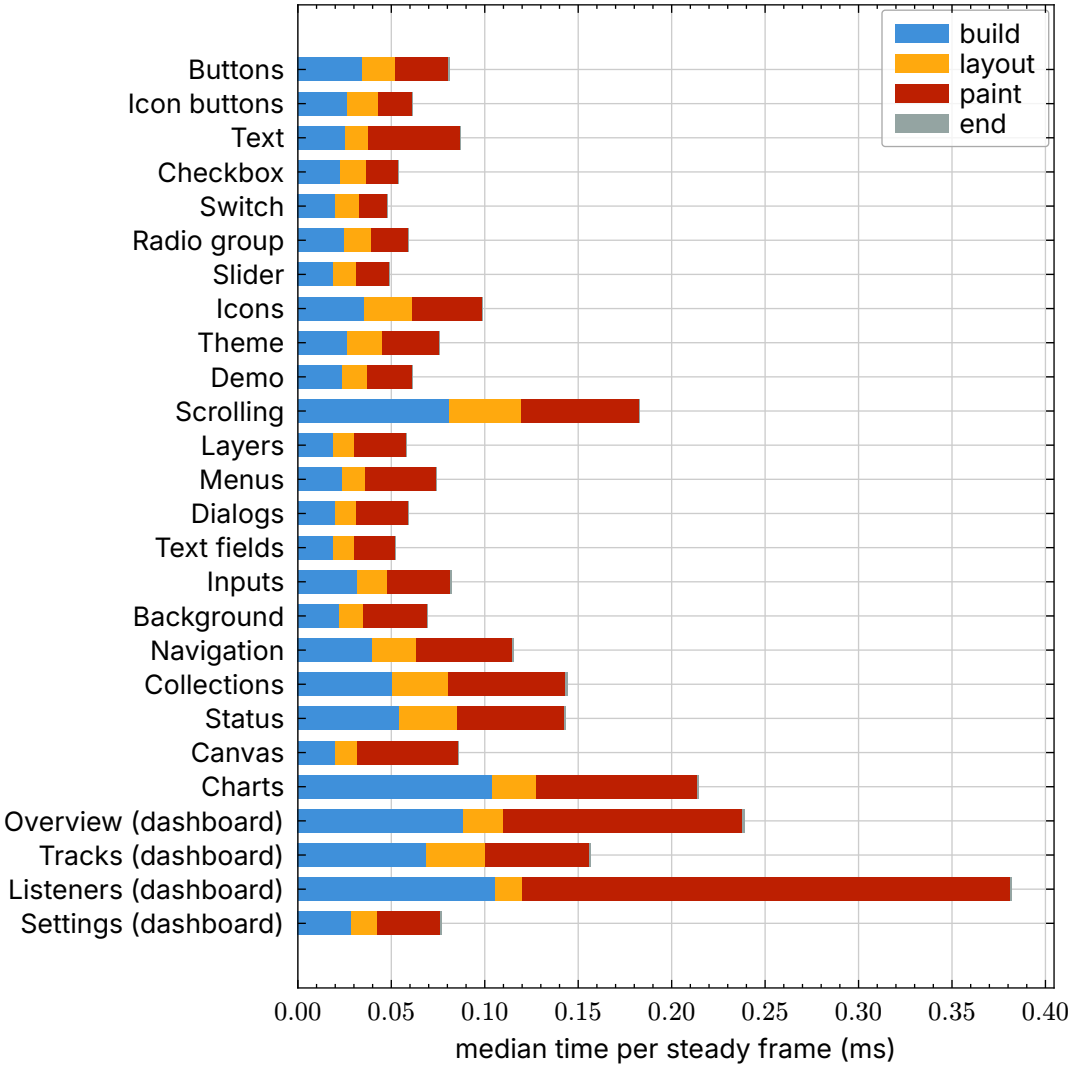


Figure 57: Kiln's own work in a steady frame of each gallery and dashboard page at 1440 × 900, split by phase: the median of each phase over 200 frames, measured on an AMD Ryzen 5 3600X 6-Core Processor, release build at O3. Drawing is left out. end is a few microseconds, too thin to see.

Most pages cost under a fifth of a millisecond; the dashboard's pages and the gallery's charts, with their charts and tables, cost the most. Building and painting take most of each frame, and painting leads on pages heavy with text, charts and tables, since it turns every element into vertices; layout grows with the number of elements and stays small.

The focused cases, from the same baseline. Steady cases give Kiln's own work per frame (**ui**); the others time one operation:

Case	Measure	Median
Shaping 650 characters, uncached	time	0.26 ms
A 10,000-row long list, scrolled each frame	ui	0.04 ms
The dashboard's overview over a year	ui	1.16 ms
A 5,000-point line chart	ui	0.37 ms
The same line zoomed to 250 points	ui	0.08 ms
A 50,000-point line chart	ui	1.57 ms

Case	Measure	Median
A 365-point area chart	ui	0.10 ms
A 5,000-point area chart	ui	1.08 ms
Typing in a 2,000-character text field	ui	0.19 ms
Dragging a selection in a 2,000-character text field	ui	0.20 ms
500 checkboxes	ui	0.54 ms
Making and destroying a `Ui`	time	1.26 ms
The first frame of mixed scripts, with system fallback fonts	time	29.49 ms

A long list costs about what a small page does whatever its length, since it builds only the rows in view, and a chart zoomed to part of its line costs a fraction of the whole. A chart's cost grows with its points, though more slowly than they do, and an area chart, which fills down to the axis, costs more than a line of the same length. Shaping is the costly step in text, which is why shaped text is cached (Caches): a steady frame shapes nothing. The first frame of mixed scripts is the costliest single operation, and the one that varies most between machines, since it opens and reads the system's fonts.

29.3.2 The first frame

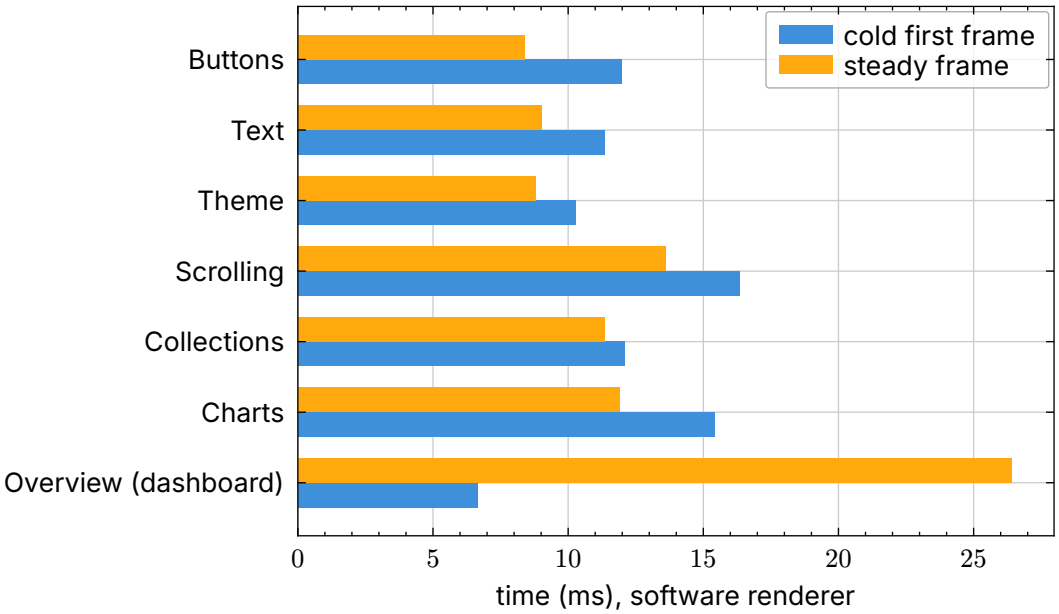


Figure 58: A new `Ui`'s first frame against a steady frame (the median of 200), both built and drawn with the software renderer, measured on an AMD Ryzen 5 3600X 6-Core Processor, release build at O3. The cold frame includes making the app's state, shaping every text and filling the atlas, but not making the `Ui` itself.

Both bars include the software draw, which is most of each, so the difference between them is what the first frame adds. On most gallery pages that is one to a few milliseconds: shaping every text and rasterizing its glyphs, which later frames find in their caches. The cold frame is a single sample against the median of 200, so on pages where the difference is small, such as Collections, it says little, and it can come out below the steady frame. The dashboard's first frame is much cheaper than its steady frames, because its data comes from a background job (Background work): the first frame shows skeleton cards, and the charts and tables appear on later frames. Making the `Ui` before the first frame, and destroying it after, takes about 1.3 ms more.

29.3.3 Real frames on a GPU

Both App-style backends record frames when asked through the environment. With `KILN_FRAME_TIMES=<file.csv>`, `app::run` draws `KILN_FRAME_TIMES_FRAMES` frames (300 by default) back to back, with no idle sleep, no pacing and no vsync, then writes the file and exits. For each frame it records Kiln's own work, from `Stats`, the draw (`Ui.draw` with the backend's renderer, `GlRenderer` or `SdlGpuRenderer` on SDL3), and the present (`SdlPlatform.present`, which swaps the GL buffers or submits the SDL_GPU frame, and raylib's `EndDrawing`), and the draw's draw calls and graphics calls (Where a frame's time goes); it also records the startup, from the start of `app::run` to the end of the first presented frame, window and fonts included:

```
KILN_FRAME_TIMES=frames.csv KILN_GALLERY_PAGE=charts build/gallery-release
```

The file starts with a `startup_ms` line, then one row per frame under the header `frame,build_ms,draw_ms,present_ms,draw_calls,gl_calls`, where `build_ms` is Kiln's own work, the four phases together.

Page	ui	draw	present	draws	GL calls	startup
gallery/charts	0.23	0.04	0.18	131	2141	102
gallery/collections	0.14	0.04	0.14	109	1716	99
gallery/text	0.11	0.03	0.13	62	986	125
dashboard/overview	0.22	0.04	0.15	90	1501	93

Table 1: Real frames on a GPU: the medians of 300 frames, in milliseconds and in draw and GL calls per frame, and the startup in milliseconds, measured on an AMD Ryzen 5 3600X 6-Core Processor, release build at O3. The baseline's `gallery` and `dashboard` rows measure the raylib builds.

With a GPU, drawing takes a fraction of Kiln's own work on the CPU: the renderer uploads the draw list's vertices once and issues a draw call per draw command, and the GPU does the rest after `Ui.draw` returns. A draw command ends only where a different texture is needed, since each vertex carries its own clip and mode, so a frame is about one draw per texture, and most of the GL calls set the frame's state once: blending, the shader's uniforms, the buffers and the vertex attributes. Presenting includes waiting on the driver for the buffer swap. The startup includes opening the window and everything up to the first frame on screen.

29.3.4 Sizes

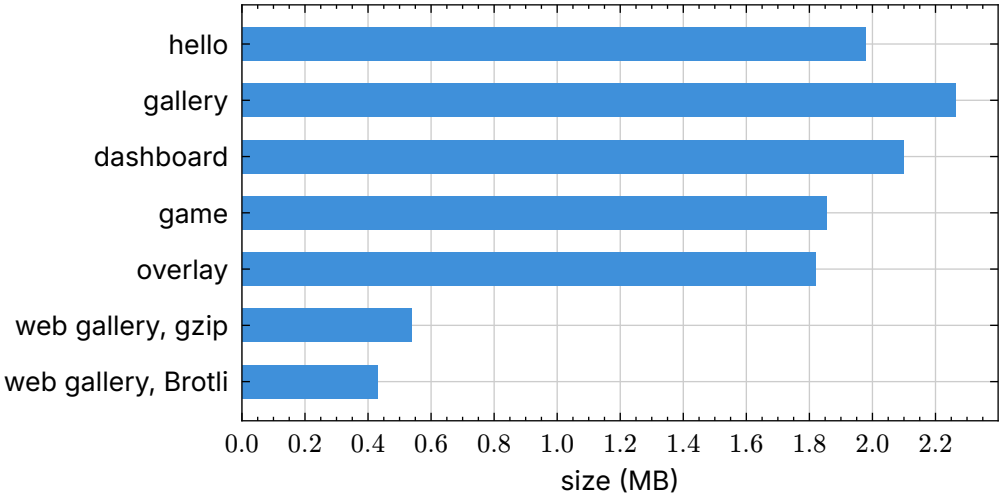


Figure 59: Each example's release build, stripped, and the web gallery's `.wasm` and `.js` together, gzipped at level 9 and with Brotli at level 11, measured on an AMD Ryzen 5 3600X 6-Core Processor, release build at O3. The baseline's `hello`, `gallery` and `dashboard` rows measure the raylib builds.

Every example embeds Kiln's own fonts (Fonts) and carries Kiln, Clay and the font code, so most of each binary is shared, and the examples differ by little more than the app and the backend. What an example ships, its `shipped` row, adds raylib's shared library where the example links one; on Linux raylib is linked statically, with unused sections dropped, so the executable is all there is and the two rows are the same.

The web gallery's download, compressed, is a fraction of a desktop executable, and Brotli takes a fifth or so off gzip. `scripts/web.sh` builds it small: the C sources at `-Oz`, the JavaScript minified with Closure, and no emulated file system (`-sFILESYSTEM=0`). It writes a `.gz` and a `.br` copy beside each file, so a server that supports precompressed files sends them as they are.

29.3.5 What an app carries

An app carries only the widgets it calls. c3c compiles only the functions reachable from `main` (`--strip-unused`, on by default), and Kiln's C code is compiled with a section per function, so the linker drops what nothing calls. An app that draws no chart has no chart code, and one that never opens a date picker has neither the calendar nor the code that reads the region's calendar settings.

That holds only while nothing every app runs reaches a widget: `Ui.begin` and `Ui.end`, the theme's set-up, `Ui.destroy`, or a table of function pointers that lists widgets. A widget keeps its state in its own `@state`, adds nothing to `Theme`, and reads OS settings it alone needs the first time it draws, as the date picker does (Dates). `scripts/check.sh` holds Kiln to it: its step "unused widgets stay out of hello" lists the symbols of `hello`, which calls no picker, chart or canvas, and fails if any of theirs is there. The same names must be found in the gallery, which calls them all, so a renamed function can't make the check pass by matching nothing.

29.4 The counts test

Times can't be held to exact values, but much of what a frame costs can. `test/screenshot/counts_test.c3` opens each gallery and dashboard page in a new `Ui` the way the benchmarks do, and writes one line per page:

- the allocations of its first ten frames together, the warm-up (No allocations per frame);
- after those, one steady frame's vertices and draw commands, its Clay elements, the shaped texts in the cache and the glyphs in the atlas.

The lines must match `test/counts.csv` exactly. A change that makes a page draw more vertices, open more elements or allocate more while it warms up fails the test, and when the change is intended, `KILN_UPDATE_COUNTS=1` rewrites the file, so the new costs are a diff reviewed with the code, as the screenshots are (Screenshot tests). The steady frame's allocations are held to zero by a test of their own (Frame statistics).

Under WebAssembly the test leaves the allocations out: with 32-bit sizes, lists grow at other moments and the counts differ by one or two. It skips the gallery's Background page there, which shows the web's own text.

30 C3 features in Kiln

Kiln's API is plain C3: a widget is a method on `Ui`, its options are named arguments, and a container is a macro whose body is ordinary code. There are no class strings to parse, no callbacks to register and no retained widget objects to keep in sync. That shape comes from a handful of C3 features. Named and default arguments replace option strings and builder chains; trailing-body macros give containers a block of code instead of a closure; `defer` makes every push meet its pop; `$Type` parameters give typed widget state without generics on `Ui`. This chapter maps each feature to the place it carries weight, with a real example from the source, and points to the chapter that explains the design around it.

30.1 The map

C3 feature	Used for, with an example	See
Methods with named and default arguments	Widget options. `Ui.button` takes `variant`, `size`, `disabled`, `icon`, `icon_end` and `full`, all with defaults (`src/ui/widgets/button.c3`), so a call names only what differs: `ui.button("Save", PRIMARY, icon: CHECK)`.	Widgets
Trailing-body macros	Containers. `Ui.@row`, `Ui.@column`, `Ui.@stack` and `Ui.@card` forward to `Ui.@box_ex`, which opens a Clay element, runs the body and closes it (`src/ui/layout.c3`). The same shape scopes ids (`Ui.@id`), themes (`Ui.@theme`), focus (`Ui.@focus_scope`, `Ui.@focus_group`) and shown states (`Ui.@show_state`).	Layout
`defer`	Scoped push and pop inside those macros: `close_box()` in `@box_ex`, `Ui.pop_id` in `@id`, `close_scope()` in `@focus_scope`, the group stack in `@focus_group`, and restoring the outer theme in `@theme`. `Fonts.init` uses `defer catch self.free()` to undo a half-built font set. Clip regions (`DrawList.push_clip`/ `DrawList.pop_clip`) and token overrides (`Ui.push_token`/ `Ui.pop_token`) are paired calls, not `defer`.	Layout, Input and focus
Macros with a `$Type` parameter	Typed widget state: `Ui.@state` called with `SliderState` and an id returns a `SliderState*`, zeroed the first time the id asks (`src/ui/state.c3`). `Ui.frame_new` allocates a paint struct in the frame arena the same way.	State and memory
Arena allocator	The frame arena. `Ui.frame_arena` is a `DynamicArenaAllocator`, reset by `Ui.begin` (`src/ui/ui.c3`). Paint data (`frame_new(ButtonPaint)`) and formatted strings (`string::format(&ui.frame_arena, ...)`, `Ui.textf`) live there until the next frame.	State and memory
Distinct types (`typedef`)	`typedef Id = inline ulong;` (`src/ui/id.c3`): an id is its own type, so a plain `ulong` can't be passed where an id is expected, while `inline` keeps it usable as a number for hashing.	Widgets
Enums	Closed sets of choices: `Variant` and `ControlSize` (button), `Dir`, `Align`, `Elevation`, `Corners` (layout), `Radius`, `TextSize`, `Breakpoint` (theme). `Motion` carries associated values: each motion token holds its `duration` and `bounce` (`src/ui/spring.c3`).	Widgets, Motion

C3 feature	Used for, with an example	See
`constdef`	Color tokens. `Token` is a `constdef` over `ushort` (`src/ui/theme.c3`), not an enum, so `Ui.add_token` can return new tokens beyond the built-in ones.	Theme
`bitstruct`	Packed flags: `Interact` (`hovered`, `pressed`, `clicked`, `focused`, `ring`, ...) in a `ushort`, `Gamepad` buttons in a `ushort`, `Mods` in a `char`, and `SliderResult` (`changed`, `released`).	Widgets, Input and focus
Optionals and faults	Things that can fail outside the program's control. `Fonts.init` returns `void?` and fails with `io::INVALID_FORMAT` when the main font doesn't parse (`src/text/fonts.c3`); `Platform.load_asset` returns `char[]?`; the system font scan skips directories `path::ls` can't read; `Ui.rect_of` returns `Rect?` for a widget not laid out yet.	Text, Backends
Contracts (`<* @require *>`)	API misuse, checked in a safe build (C3's default debug build, where `env::COMPILER_SAFE_MODE` is true): `Ui.end` without `begin()`, a container left open, `pop_id` without `Ui.push_id`, `pop_token` without `push_token`, a slider whose `min` is above its `max`, `@state` asked for another type on the same id.	The frame
`$if` on safe mode	Reports that only safe builds compile in, under `$if env::COMPILER_SAFE_MODE`: duplicate ids (`Ui.make_id`, `src/ui/id.c3`), duplicate shortcuts (`Ui.shortcut`, `src/ui/shortcut.c3`), and Clay's errors other than running out of capacity or an unbalanced open and close, and the message when Clay's budget grows after an overflow (`src/ui/layout.c3`), all printed to stderr.	Widgets, Input and focus, Layout
Interfaces with `@dynamic`	The `Renderer` and `Platform` interfaces (`src/ui/backend.c3`), implemented by `RaylibRenderer`, `RaylibPlatform`, `SoftwareRenderer` and `HeadlessPlatform`. `CountingAllocator` (`src/ui/stats.c3`) implements the standard `Allocator` interface the same way.	Backends
Vectors (`float[<2>]`)	Positions and sizes: `Pointer.pos`, `Input.size`, `Input.wheel`, draw-list vertices (`pos`, `uv`), and directional focus, which subtracts rect centers and projects them on the arrow's direction (`Ui.navigate_direction`, `src/ui/focus.c3`).	Input and focus, Rendering
Generic modules	`IdMap{Value}` (`kiln::ui::idmap`), a hash map keyed by `Id` that allocates only when it grows, holds seen ids, rects, widget state and atlas glyphs; the standard `List{T}` holds the stacks.	State and memory
`$embed`	The built-in fonts: Inter Regular, Inter SemiBold and the Lucide icon subset are compiled into the binary (`src/text/embedded.c3`).	Text

30.2 Containers: a macro around Clay's open and close

Every container ends in `Ui.@box_ex`. It opens a Clay element, defers the close, and runs the body in between:

```
macro void Ui.@box_ex(&self, BoxOpts opts; @body())
{
    self.open_box(&opts);
    defer self.close_box();
    @body();
}
```

`@row`, `@column`, `@stack` and `@card` are thin macros that take named options with defaults and pass a `BoxOpts` to `@box_ex`. The body is a plain block, so children are ordinary statements and can use the caller's locals and loops without a closure:

```
ui.@row(gap: 3, align: CENTER)
{
    if (ui.button("Reset", GHOST)) volume = 0.5;
    ui.slider("Volume", &volume);
};
```

Because the close sits in a `defer`, it runs when the body ends, and the element tree Clay builds always matches the code. Layout explains the containers and their options.

30.3 Typed state from a `$Type` parameter

Widget state lives in a table keyed by id. The `@state` macro turns that untyped storage into a typed pointer, and its contract catches one id asking for two different types:

```
<*
 A pointer to `id`'s state of type `$Type`, zeroed the first time `id` asks.
 The state lives across frames and is freed after `ui::STATE_DATA_SECONDS`
 without a call, unless `Ui.keep_state` holds it.

 @require !self.has_state(id) || self.state_type(id) == $Type::typeid : "this id already
has state of another type"
*>
macro Ui.@state(&self, $Type, Id id) => ($Type*)self.state_ptr(id, $Type::typeid,
$Type::size);
```

The slider asks for its springs in one line: `SliderState* st = self.@state(SliderState, id);` (`src/ui/widgets/slider.c3`). State and memory covers how long the state lives and when it's collected.

30.4 Scoped ids: `defer` and `$if` together

`@id` scopes the ids of the widgets in its body. It accepts a string or a loop index, picking the hash at compile time with `$if`, and pops the id stack with `defer`:

```
macro void Ui.@id(&self, key; @body())
{
    $if types::is_int($Typeof(key)):
        self.push_id_index((long)key);
    $else
        self.push_id(key);
    $endif
    defer self.pop_id();
    @body();
}
```