ChatGPT

Fountain Coach SDLKit – Current Implementation Status (Pre-Alpha)

Features

- **Core SDL Wrappers:** SDLKit provides Swift wrappers for SDL3's window and rendering systems. A window wrapper (`SDLWindow`) and renderer wrapper (`SDLRenderer`) are implemented and wired to actual SDL calls when SDL3 is available [1] [2]. This allows creating windows, rendering content, and interfacing with SDL in a Swift-friendly way. In headless or CI builds (where SDL isn't present), these calls safely no-op or throw an `sdlUnavailable` error instead of failing [3].
- **Window Management:** All basic window operations are supported. You can open and close windows (each with a unique `window_id`), and control window state via functions to show/hide, resize, move, maximize, minimize, or restore windows [4] [5]. The API also supports setting window titles, opacity, and "always on top" flags, as well as querying window info (position, size, title) [6] [7]. These are exposed as JSON tools under endpoints like `/agent/gui/window/...` for external use [8] [9].
- **Drawing Primitives:** SDLKit implements basic 2D drawing primitives. You can clear the window with a color, draw filled rectangles, draw lines, and draw filled circles. Internally, the renderer wrapper handles these operations by calling SDL3's rendering functions (or custom routines). For example, `drawRectangle` uses SDL's fill-rect API [10], `drawLine` uses a simple pixel loop (or SDL's line draw for batch operations) [11] [12], and `drawCircleFilled` is implemented with a midpoint circle algorithm filling horizontal spans [13] [14]. The JSON agent provides endpoints for these primitives (e.g. `/agent/gui/clear`, `/agent/gui/drawLine`, etc.) which map to the internal functions [15] [16]. All color parameters can be given as ARGB hex integers or named/color-code strings, which are parsed by the library [17] [18].
- **Text Rendering:** Text drawing is supported via an optional SDL_ttf integration. The `drawText` tool is defined and will render text to the window if SDL_ttf is available at runtime [19]. SDLKit has a module `SDLKitTTF` that, when included, links SDL_ttf; otherwise, calling `drawText` will return a `notImplemented` error if text support isn't present [20]. The implementation resolves font specifications (including a special `"system:default"` font alias) and uses SDL_ttf to render text onto an SDL texture for drawing [21] [22]. This is currently basic but functional – font caching is in place and common system font paths are attempted on Linux tests [23] [24]. Without SDL_ttf, the function simply reports not implemented.
- **Texture Handling:** The library allows loading images as textures and drawing them. `textureLoad` can import an image from disk into an SDL texture (using SDL_image for formats beyond BMP) [25] [26]. There are corresponding methods to draw a texture at coordinates (with optional scaling) and to draw a rotated texture [27] [28]. You can also free textures by ID to release memory [29]. Tiled drawing is supported at the JSON layer by repeatedly drawing a texture in a grid (the `/agent/gui/texture/drawTiled` endpoint computes tiling in the router and calls the draw API multiple times) [30].
- **Input and Events:** SDLKit provides basic input event capture and state queries. The `captureEvent` call will poll for or wait for an SDL event (key, mouse, quit, etc.) and return the event data in a structured JSON format [31] [32]. This allows retrieving user input events from a window. Additionally, there are endpoints to get the current keyboard modifier keys state and mouse cursor position/buttons without waiting for events (via `getKeyboardState` and

`getMouseState`) [33] [34]. These leverage small utilities (`SDLInput.getKeyboardModifiers()` and `.getMouseState()`) that call SDL to retrieve the instantaneous input state [35] [36]. (Note that SDLKit does not yet simulate or inject input events – it only reads input.)

- **Clipboard and Display Info:** The library includes support for clipboard text and display metrics. The `/agent/gui/clipboard/get` and `/agent/gui/clipboard/set` tools allow reading or writing the system clipboard text [37] [38]. Internally this uses SDL's clipboard APIs (e.g. `SDL_GetClipboardText`) and is functional when a window is open [39] [40]. For display info, endpoints exist to list displays and get a display's bounds. SDLKit will enumerate video displays (monitors) and return their identifiers and names, and can return a display's pixel bounds [41] [42]. This can be useful for positioning windows or determining screen size.
- **JSON Agent and API:** A significant feature of SDLKit is the JSON GUI Agent (`SDLKitGUIAgent` and its `SDLKitJSONAgent` router). All the functionality above is exposed through a JSON API contract (`sdlkit.gui.v1`) that Fountain's AI planners or external clients can call [43] [44]. The JSON router maps HTTP requests to the appropriate agent methods, handles JSON encoding/decoding, and returns JSON responses or error codes [45] [46]. For example, a POST to `/agent/gui/window/open` with JSON body `{"title":"Demo","width":640,"height":480}` will invoke `openWindow` and return `{"window_id": 1}` [47] [48]. Every tool (open, close, draw, present, etc.) has a corresponding endpoint defined. The agent enforces argument validation (returning `invalid_argument` errors for bad inputs) and uses canonical error codes (`window_not_found`, `sdl_unavailable`, `not_implemented`, etc.) as defined in the contract [49]. An OpenAPI 3.1 specification is included in the repo to document all endpoints and schemas [50] [51], ensuring the API is well-defined.
- **Examples and Demo:** The repository includes a demo program and an example server. The `SDLKitDemo` executable target can be run to open a window and draw some shapes (clear screen, a rectangle, a line, a circle, and attempt text) to showcase the library [52]. There is also an `Examples/SDLKitJSONServer` which is a lightweight Swift NIO HTTP server that uses `SDLKitJSONAgent` to handle incoming requests [53] [54]. This server serves the OpenAPI spec at `/openapi.yaml/json` and implements the agent endpoints, demonstrating how SDLKit could be embedded in a service. These examples are rudimentary but confirm that the pieces (agent + HTTP + SDL) work together. (On macOS, a window will actually appear when running the demo or server, whereas in headless CI these calls are no-ops.)

Overall, most planned features for a v1 appear to be sketched out and in place, but a number of them are still in a preliminary or partial state (throwing `notImplemented` in some build modes or lacking polish). The core window and rendering functionality is **implemented but labeled alpha** – basic operations work, yet more advanced aspects (performance tuning, extensive error handling) may need refinement. The JSON agent covers a broad surface area (window management, drawing, input, etc.), essentially fulfilling the intended "bounded GUI tools" contract, though some tools only function when optional SDL components are present. All interactions are confined to the main thread for safety (the agent is marked `@MainActor` to ensure SDL calls happen on the main thread) in line with SDL's requirements [55] [56].

Test Coverage

The current test suite is limited in scope, focusing on non-UI logic and API correctness rather than graphics output. There are a handful of unit tests (`Tests/SDLKitTests`) covering configuration and utility functions, JSON routing, and OpenAPI consistency:

- **Configuration and Utilities:** Tests verify that default config values and environment overrides work (e.g. `SDLKitConfig.presentPolicy` defaulting to `.explicit`, and `SDLKIT_GUI_ENABLED` env var enabling/disabling GUI) [57] [58]. Color parsing is tested thoroughly: various color string formats like `#RRGGBB` or named colors are decoded to the correct ARGB hex, and invalid strings throw errors [23] [59]. These ensure the helper `SDLColor.parse` behaves as documented. There is also a Linux-only test that checks the `SDLFontResolver` can find a system font file for `"system:default"` – it searches common font paths and expects at least one readable `.ttf` to exist, skipping the test if none is found [24] [60]. This validates the fallback logic for font selection on Linux systems.
- **JSON API and OpenAPI:** A significant portion of the tests focuses on the JSON agent and OpenAPI spec, ensuring the JSON responses and error codes conform to the contract. For example, `JSONRouterTests` send requests to unknown or invalid endpoints and assert that the agent returns the proper error JSON (`not_implemented` for an unrecognized `/agent/gui/*` path, and `invalid_endpoint` for a non-agent path) [61] [62]. There are tests that load the embedded OpenAPI YAML file and compare it against what the agent serves at runtime, to guarantee no divergence [63]. Another test converts the YAML spec to JSON (using Yams when enabled) and deep-compares it to the agent's `/openapi.json` output, ensuring the conversion logic is correct and consistent [64] [65]. Caching behavior is also verified: setting the `SDLKIT_OPENAPI_PATH` env var to an external spec file, the tests check that repeated requests only trigger one conversion (caching the result) [66] [67], and that removing or changing the external file properly falls back to the internal spec [68] [69]. These tests give confidence that the API documentation endpoints and version reporting work reliably in various scenarios.
- **Screenshot and Error Handling:** Because rendering output can't easily be checked in an automated test, the library uses mock-based tests to verify screenshot functionality and error propagation. A `SDLKitScreenshotTests` class defines a dummy `MockScreenshotAgent` (subclassing the GUI agent) that overrides `screenshotRaw` and `screenshotPNG` to return predetermined data or throw errors [70] [71]. Using this, tests call the `/agent/gui/screenshot/capture` endpoint with format "raw" or "png" and assert that the JSON response contains the expected base64 strings and metadata [72] [73]. They also force a `notImplemented` error for PNG (simulating a case where SDL_image is unavailable) and confirm the error JSON contains code `"not_implemented"` and the message suggesting to retry with raw format [74] [75]. This approach ensures the JSON agent correctly wraps the screenshot data and error in the expected schema without needing a real window or SDL_image.

Overall, the tests cover configuration, parsing, and the JSON interface thoroughly, but do not yet exercise the actual GUI operations in a live SDL environment. There is **minimal integration or end-to-end testing of rendering** – for instance, no automated test opens a real window and draws a shape (since in CI the code runs in headless mode by design [76]). The project's guidelines note that if SDL3 is missing, tests should skip rather than fail [77], which is reflected in how the suite is written (e.g. the font test skips if no font, and presumably any attempt to use a GUI without SDL would be skipped or stubbed). As a result, current coverage ensures the *logic and API contract* are sound, but **does not yet guarantee that drawing a rectangle on-screen truly works on all platforms**. Manual testing or additional integration tests (perhaps on a machine with graphics) will be needed to validate the actual rendering and window management behaviors. There's also no reported coverage percentage, but given the limited number of tests focusing on specific components, certain modules (like the internals

of `SDLWindow` / `SDLRenderer`) are likely not exercised by the test suite at all. Increasing test coverage – especially for critical path functions like window open/close and rendering calls – is an important step toward production readiness.

Documentation

The SDLKit project includes both **developer-oriented documentation** and **user-facing API documentation**, though some areas could be expanded as the project matures:

- **README:** The repository's README.md provides a solid high-level overview. It describes the package purpose and design (a Swift wrapper around SDL3 for use in Fountain's AI ecosystem), and lists status information and project structure [78] [79]. The Status section explicitly labels the project as "Alpha (Pre-Alpha)" and summarizes which parts are wired up (core window+renderer, JSON agent tools) [80]. Build and usage instructions are given, including how to install SDL3 (via Homebrew, apt, vcpkg depending on OS) and how to compile and run tests [81] [82]. There's guidance on running in headless mode for CI and enabling text rendering by including the `SDLKitTTF` product [83] [20]. The README also presents a Quick Start code snippet and a note on a demo, which helps new users get an initial window open (with a caveat that some draw calls still throw `notImplemented` in this early version) [84]. Overall the README is informative, though the quick start example hints at incomplete features (e.g. noting `drawText/drawRectangle/present` are not implemented yet) which may be slightly outdated given recent code – this could be updated as those functions come online.
- **Agent Contract & Internal Docs:** For more detailed documentation, the project relies on an **AGENTS.md** file. This serves as both a contributor guide and a specification of the GUI agent's contract [85] [86]. It outlines coding conventions, thread-safety requirements, testing guidelines, and security considerations for the repo [87] [88]. Importantly, it defines the agent's JSON API in prose: listing each tool endpoint, expected request/response JSON, and error codes [89] [19]. The agent capabilities are enumerated (openWindow, closeWindow, drawText, etc.), matching what's in the OpenAPI spec, along with notes (e.g. that `drawText` may be a no-op if SDL_ttf is unavailable, or that `clear` / `drawLine` / `drawCircleFilled` were newly added) [19] [90]. It also describes the event schema and threading model, and environment variables like `SDLKIT_GUI_ENABLED` and `SDLKIT_PRESENT_POLICY` that control runtime behavior [91]. For developers, AGENTS.md includes a "Contributor Workflow" section explaining how to extend the agent (e.g. steps to add a new tool: update schema, implement method, add tests, etc.) [92] [93]. This file is a valuable reference for both maintainers and integrators (AI agents) to understand the intended behavior of the GUI agent.
- **API Reference (OpenAPI Spec):** The project maintains an **OpenAPI 3.1** specification file (`sdlkit.gui.v1.yaml`) that is considered the source of truth for the JSON API [50]. This spec documents every endpoint under `/agent/gui/` as well as the health and version endpoints [94] [95]. Each operation has a summary, description, and schema for its request and response. For example, the spec defines the schema for a `OpenWindowRequest` and `OpenWindowResponse`, error responses, and so on [96] [97]. The OpenAPI file is used at runtime too – the agent serves it directly via `GET /openapi.yaml` or converted to JSON via `GET /openapi.json` [98] [99]. This means clients can retrieve the live API documentation from the running agent, ensuring they always get the correct version. The presence of this spec is a strong point in documentation, as it will enable auto-generated client code and help ensure forward/backward compatibility rules are followed (the spec version is tracked separately and the agent's `/version` call includes both agent protocol and spec version numbers [100] [101]).
- **Code Comments:** At this stage (pre-alpha), inline code documentation is sparse. Public classes and methods do not yet have extensive doc comments (e.g. many `SDLKitGUIAgent` methods

and core functions lack Swift documentation comments). Instead, the high-level behavior is documented in README/AGENTS.md, and the code is relatively straightforward to read. In the future, adding doc comments for each public API (for use with Swift documentation generation) would be beneficial. The project guidelines do encourage documenting public APIs and keeping docs updated with code changes [92], so we can expect more comments to be added as the code stabilizes.

- **Additional Docs:** The repository doesn't appear to have a separate website or wiki yet. However, the included markdown files (`README.md`, `AGENTS.md`) and the OpenAPI spec together cover both usage and development. There is also a brief `NEWS.md` or `CHANGELOG` placeholder referenced in search results, but it's not clear if it's maintained – likely as the project is pre-release, formal release notes don't exist yet. Similarly, there's a `TODO` or roadmap embedded in the README and issues might be tracked on GitHub for missing pieces.

In summary, **the documentation is good for an alpha project**: a new user can read the README to get started and understand what's done vs. not, and a developer can consult AGENTS.md and the OpenAPI spec for detailed behavior of each API call. As SDLKit moves toward production, documentation will need to evolve from "what we plan to do" to "what is fully supported," updating any outdated notes (for example, removing mentions of unimplemented functions once they are implemented). Additionally, more example usage (beyond the quick start snippet) and higher-level guides (like how to integrate SDLKit into an AI planning system or troubleshooting common setup issues) would enhance the user-facing docs.

Path to Full Implementation and Production Readiness

While SDLKit has made significant progress in its implementation, it is not yet production-ready. The roadmap and current status highlight several areas to complete or improve before a stable release:

- **Complete Remaining Features:** All major planned features have been outlined, but a few are partially implemented:
- *Audio and Advanced Features:* (If in scope) There's no mention of audio or more complex rendering (3D, shaders) – SDLKit focuses on 2D GUI primitives. This may be fine for the intended use (AI planners controlling simple GUIs), but it should be clarified if features like audio output or 3D rendering are out of scope or slated for future versions.
- *Error Handling and Edge Cases:* The code should be reviewed for robust error handling. For instance, the environment variable `SDLKIT_MAX_WINDOWS` exists (default 8 windows) [102] but currently the code does not enforce this limit – opening more windows than the soft cap might be allowed until SDL fails. Implementing such limits and handling failures (like out-of-memory when creating surfaces, or inability to open a window on headless systems) with user-friendly errors is important for production.
- *Resource Management:* Ensure that all resources are properly freed. The current implementation relies on OS cleanup for some things (for example, it calls `SDL_DestroyWindow` and `SDL_DestroyTexture` where needed, but does not call a global `SDL_Quit` – which is acceptable but worth documenting). Font resources are cached; a strategy to free them on shutdown or on memory pressure might be needed to avoid leaks if an application dynamically loads many fonts. These considerations, while minor, contribute to long-term stability.
- *Cross-Platform polish:* So far, development and testing have mostly been on macOS and Linux (as indicated by instructions and CI). Windows support needs verification – e.g., the vcpkg installation path and library discovery logic should be tested on Windows. The code uses `#if os(Linux)` in at least one test and hard-codes Unix font paths [103], so there may be platform-specific adjustments needed. Ensuring that the library works consistently on Windows (window

creation, event capture, etc.) and addressing any Windows-specific SDL quirks (such as the need to pump events differently) will be necessary for full production readiness.

- **Expand Testing (Integration & CI):** As noted, test coverage needs to grow. Before release, we should have:
- *Integration Tests:* Create tests or sample applications that actually open a window and perform a sequence of drawing operations, then perhaps read back pixels (using the screenshot function) to verify that the expected pixels were drawn. This could be done in a headless framebuffer or with a virtual display if running in CI is difficult. Even without pixel-perfect checks, running the code on a real display and ensuring no crashes or obvious errors (and that closing windows works without memory leaks) is crucial. Automated UI tests could be challenging, but a controlled environment (like using Xvfb on Linux to simulate an X display) might allow some level of automated integration test for drawing.
- *Cross-Platform CI:* At the moment, CI is likely only running in headless mode on Linux. Enabling a macOS runner for CI (the README suggests setting up a self-hosted Mac runner for validation [104]) would help catch Mac-specific issues. Similarly, running tests on Windows (perhaps on a GitHub Actions Windows runner) would ensure the code builds and the stub logic or actual logic behaves on that platform. Even if GUI tests on Windows are non-interactive, simply linking and running the basic logic tests there would increase confidence.

- *Additional Unit Tests:* Write tests for any logic that can be tested in isolation. For example, tests for the color parsing already exist, but we could add tests for the geometry functions (e.g., ensure that drawing a line via `drawLine` sets the correct pixels – perhaps by intercepting calls or by using the pixel buffer capture on a small surface). Testing the font resolution on macOS (looking up "Arial" or system fonts) would complement the Linux font test. Also, tests to ensure that calling window operations in invalid sequences (like calling `present` after window is closed) yields the expected `window_not_found` error would be helpful.

- **Performance and Stability:** As the code transitions from alpha to beta, it should be optimized and hardened:

- *Rendering Performance:* The current implementations of `drawLine` and `drawCircleFilled` are straightforward but not highly optimized (using per-pixel loops). SDL's renderer might have more optimized functions (e.g., `SDL_RenderLine` which SDL3 introduced) – in fact, the shim wraps `SDL_RenderLine` and `SDL_RenderPoints` and the code uses them for batched operations [12] [105], but single-line draws use a custom loop. We should evaluate if performance is sufficient for the expected use (small drawings for an AI UI) or if these should leverage SDL's primitives. Similarly, if many shapes are drawn, ensuring that using an explicit present policy vs. auto present is well-documented so users can batch draws efficiently is important.
- *Concurrency and Threading:* The agent is main-thread-bound, which simplifies a lot. We should ensure that calling the JSON agent from background threads doesn't cause issues (the code uses `@MainActor`, so Swift will hop threads as needed – this is good). It would be wise to test a scenario with multiple concurrent JSON requests to verify that the main actor effectively serializes them and no data races occur with the `windows` dictionary. Since Swift's concurrency is used, this is likely fine, but a stress test or two might be considered.

- *Stability under Load:* Open and close many windows in succession, load many textures, etc., to see if any crashes or resource exhaustion occurs. Because there is a soft limit of windows and caching of fonts/textures, the system seems designed for at most a handful of windows (default 8). If an AI agent attempted to open dozens of windows or very large textures, how does SDLKit respond? These scenarios should either be documented as out-of-scope or handled gracefully (e.g., by returning an error if limits are exceeded).

- **Documentation Improvements:** As we reach production readiness, documentation should be updated and expanded:

  - *Update Status and Examples:* The README's status notes and quick start should be revised to reflect the latest state (for example, if `drawRectangle` and `present` are now implemented, the comment that they throw `notImplemented` should be removed to avoid confusion). The quick start could be expanded to a slightly more complete example once more features work (perhaps showing drawing a shape and calling present). Providing a screenshot or GIF of the demo output in the README could also be helpful for users to see what SDLKit can do.
  - *API Documentation:* Generating a reference from the OpenAPI spec (HTML pages or similar) might be useful for users who don't read YAML. Alternatively, a brief section in the README listing the key endpoints and their purpose (with a pointer that full details are in the OpenAPI spec) can guide new users.
  - *In-Code Docs:* Adding Swift documentation comments for public APIs will help developers who use IDEs like Xcode get quick help. For instance, each method in `SDLKitGUIAgent` could have a short description of what it does and what errors it throws. This will also ensure that if the project is published as a Swift Package, the documentation appears in DocC or other Swift doc generators.

  - *Guides and Tutorials:* Consider writing a short "Getting Started with SDLKit" guide separate from the reference. This could explain how to integrate SDLKit into an existing Swift project, how to run the JSON server and call it (with curl examples or a sample client), and how to compile against the Fountain-Coach SDL fork. Right now, some of this info is in the README, but as adoption grows, more user-friendly guides will reduce friction.

- **Finalize and Harden the API:** Since production readiness implies a stable API, the team should review the toolset for completeness and consistency:

  - Ensure naming is consistent (the OpenAPI uses certain endpoint naming; the Swift API uses camelCase – this seems fine).
  - If any planned tool is missing (the roadmap in the README mentioned possibly "OpenAPI wiring samples and end-to-end examples" and all those have been started [106]), make sure they are done or defer them explicitly. For example, if multi-draw batching (drawPoints/drawLines/drawRects) was a "nice-to-have," it's actually implemented now. If something like an `input/textInput` tool (for typing text) was considered, it's not present – if it's out of scope, that's fine but clarify it.
  - Version the API appropriately: it's labeled v1, and the guidelines state that only additive changes will be made moving forward [107] [108]. Before a 1.0 release, double-check that all error codes and responses align with how the AI planner expects them, so that we won't need breaking changes. Any adjustment now (during pre-alpha) is easier than after public release.
  - Security and sandboxing: Since this toolkit allows GUI operations potentially triggered by an AI, consider if any additional safeguards are needed (the docs mention GUI features are off by default on server platforms [109]). Make sure that if `SDLKIT_GUI_ENABLED=false`, *all* GUI-affecting calls truly become no-ops or throw `sdlUnavailable` – this seems to be handled via checks in code (e.g., `guiEnabled` gating `openWindow` [110]). This is important for production deployments where one might run the agent in a headless server environment safely.

In conclusion, **to reach production readiness, SDLKit needs to go from a "minimal, compilable skeleton" [111] to a fully tested, documented, and hardened library**. The good news is the foundations are largely in place: the feature list is comprehensive and aligned with the design goals, and the basic system works in practice for simple demos. The focus now should be on filling in any

remaining gaps (both in code and in docs), testing extensively across environments, and iterating on feedback. By completing the roadmap tasks – wiring up any loose ends, improving test coverage, and polishing documentation – Fountain Coach SDLKit will be well on its way to a 1.0 release that is robust and trustworthy for production use.

Sources:

- Fountain-Coach SDLKit README (Project status, features, roadmap) [78] [106]
- Fountain-Coach SDLKit AGENTS.md (Agent API contract and development guidelines) [19] [49]
- SDLKit source code (selected implementations of features and tests) – e.g. SDLWindow/SDLRenderer.swift and JSONTools.swift [112] [15]
- SDLKit Tests (verifying configuration, color parsing, JSON responses, etc.) [23] [61]

---

[1] [2] [3] [17] [18] [20] [43] [47] [52] [76] [78] [79] [80] [81] [82] [83] [84] [104] [106] README.md
https://github.com/Fountain-Coach/SDLKit/blob/3fe78905784a4ed05a8a45cd895604aeb4a9b9c5/README.md

[4] [5] [6] [21] [31] [32] [110] SDLKitGUIAgent.swift
https://github.com/Fountain-Coach/SDLKit/blob/3fe78905784a4ed05a8a45cd895604aeb4a9b9c5/Sources/SDLKit/Agent/SDLKitGUIAgent.swift

[7] SDLWindow.swift
https://github.com/Fountain-Coach/SDLKit/blob/3fe78905784a4ed05a8a45cd895604aeb4a9b9c5/Sources/SDLKit/Core/SDLWindow.swift

[8] [9] [15] [16] [30] [33] [34] [37] [38] [45] [46] [48] [99] [100] [101] JSONTools.swift
https://github.com/Fountain-Coach/SDLKit/blob/3fe78905784a4ed05a8a45cd895604aeb4a9b9c5/Sources/SDLKit/Agent/JSONTools.swift

[10] [11] [12] [13] [14] [22] [25] [26] [27] [28] [29] [105] [112] SDLRenderer.swift
https://github.com/Fountain-Coach/SDLKit/blob/3fe78905784a4ed05a8a45cd895604aeb4a9b9c5/Sources/SDLKit/Core/SDLRenderer.swift

[19] [44] [49] [50] [51] [55] [56] [77] [85] [86] [87] [88] [89] [90] [91] [92] [93] [98] [102] [107] [108] [109] [111] AGENTS.md
https://github.com/Fountain-Coach/SDLKit/blob/3fe78905784a4ed05a8a45cd895604aeb4a9b9c5/AGENTS.md

[23] [57] [58] [59] SDLKitTests.swift
https://github.com/Fountain-Coach/SDLKit/blob/3fe78905784a4ed05a8a45cd895604aeb4a9b9c5/Tests/SDLKitTests/SDLKitTests.swift

[24] [60] [103] FontResolverTests.swift
https://github.com/Fountain-Coach/SDLKit/blob/3fe78905784a4ed05a8a45cd895604aeb4a9b9c5/Tests/SDLKitTests/FontResolverTests.swift

[35] [36] SDLInput.swift
https://github.com/Fountain-Coach/SDLKit/blob/3fe78905784a4ed05a8a45cd895604aeb4a9b9c5/Sources/SDLKit/Core/SDLInput.swift

[39] [40] SDLClipboard.swift
https://github.com/Fountain-Coach/SDLKit/blob/3fe78905784a4ed05a8a45cd895604aeb4a9b9c5/Sources/SDLKit/Core/SDLClipboard.swift

[41] [42] SDLDisplay.swift
https://github.com/Fountain-Coach/SDLKit/blob/3fe78905784a4ed05a8a45cd895604aeb4a9b9c5/Sources/SDLKit/Core/SDLDisplay.swift

53 54 main.swift

https://github.com/Fountain-Coach/SDLKit/blob/3fe78905784a4ed05a8a45cd895604aeb4a9b9c5/Examples/SDLKitJSONServer/Sources/main.swift

61 62 68 69 JSONRouterTests.swift

https://github.com/Fountain-Coach/SDLKit/blob/3fe78905784a4ed05a8a45cd895604aeb4a9b9c5/Tests/SDLKitTests/JSONRouterTests.swift

63 64 65 66 67 OpenAPIConversionTests.swift

https://github.com/Fountain-Coach/SDLKit/blob/3fe78905784a4ed05a8a45cd895604aeb4a9b9c5/Tests/SDLKitTests/OpenAPIConversionTests.swift

70 71 72 73 74 75 SDLKitScreenshotTests.swift

https://github.com/Fountain-Coach/SDLKit/blob/3fe78905784a4ed05a8a45cd895604aeb4a9b9c5/Tests/SDLKitTests/SDLKitScreenshotTests.swift

94 95 96 97 sdlkit.gui.v1.yaml

https://github.com/Fountain-Coach/SDLKit/blob/3fe78905784a4ed05a8a45cd895604aeb4a9b9c5/sdlkit.gui.v1.yaml