ChatGPT

SDLKit – Swift SDL3 Wrapper Architecture and Design

Introduction and Rationale for SDLKit as a Standalone Module

Why a separate SDLKit? In the Fountain-Coach ecosystem, keeping modules focused and decoupled is a core principle. SDLKit is conceived as an independent Swift Package wrapping SDL3 (Simple DirectMedia Layer v3) instead of embedding SDL logic directly into TeatroKit (the GUI module). The primary rationale is **modularity and reuse**: by isolating all SDL-specific code into its own package, multiple projects can leverage it (e.g. FountainKit's AI control flows, TeatroKit's GUI, or other tools) without forcing a heavy GUI dependency on each. This follows the same successful modularization approach used in FountainKit and TeatroKit, where components are split into focused modules for maintainability [1] [2].

Decoupling GUI from core logic: Embedding SDL in TeatroKit's monolith would violate the strict separation of concerns. TeatroCore (the rendering engine) should remain free of C library dependencies like SDL [3], and even TeatroGUI (the interactive preview) should ideally not hard-code a specific backend. By creating SDLKit as a standalone, **TeatroGUI can depend on SDLKit** for actual windowing and rendering, but other parts (like headless rendering or audio-only functionality) remain unaffected. This decoupling prevents unwieldy growth of any single module and lets the SDL-based GUI evolve on its own timeline [1]. New contributors can focus on the SDLKit module separately, and headless or server environments can simply omit SDLKit if a GUI isn't needed (keeping headless usage lightweight [1]).

Cross-project consistency: A dedicated SDLKit also promotes consistency across the organization. It can serve as the unified low-level graphics layer for all FountainAI applications (current and future), rather than having each project roll its own SDL integration. As noted in the Teatro refactoring plan, the goal is for the SDL-powered GUI to become a **unified rendering backend** for FountainAI apps. Having SDLKit as a shared module makes that possible: FountainKit's agents, the Teatro preview, and even potential future UI tools can all call into the same SDLKit API. In short, **SDLKit encapsulates all platform-specific media handling (windows, rendering, input, etc.)** in one place, keeping higher-level modules clean and platform-agnostic.

SDLKit Architecture Overview

Module structure: SDLKit will be implemented as a Swift Package Manager module with clear internal separation between the **C SDL library bindings** and the **Swift-friendly API**. We plan to include a SwiftPM *system library target* (e.g. `CSDL3`) that links against the installed SDL3 native library on each platform. On top of that, the main `SDLKit` target will provide the pure Swift interface. This layering ensures that low-level C interop is contained and can be evolved or replaced easily (for example, if switching from dynamic linking to static, or adding SDL extensions like TTF). The Swift package will declare a dependency on the SDL3 system libraries (via pkg-config on Linux, Homebrew on macOS, etc.), but **SDLKit itself will expose a high-level API** rather than C symbols.

System library linkage: SDLKit relies on SDL3 being available on the system or development environment. On macOS and Linux, this likely means the developer installs SDL3 (e.g. via Homebrew or apt). The `CSDL3` target will use SwiftPM's `pkgConfig` or `linkedLibrary` to find `libSDL3` at compile/link time. On Windows, SDL3 can be handled by providing the appropriate `.dll` and headers (possibly via a Vcpkg integration or manually). The architecture must account for platform differences in locating SDL – for instance, conditional compile settings or build scripts might be used to help find SDL3 on Windows. By encapsulating this in the system library target, **the higher Swift code remains identical across platforms**, and only the package manifest or build settings handle the differences. All three major platforms (macOS, Linux, Windows) are intended to be supported from the start or via staged rollout, with testing on each [4] .

Abstraction layers: SDLKit's design introduces a **Swift-friendly abstraction layer** above the raw SDL C API. Instead of exposing C pointers or requiring manual SDL function calls, SDLKit provides Swift `class` es/ `struct` s that manage SDL resources and state. For example, there will be an `SDLWindow` class to represent a window, an `SDLRenderer` for drawing, perhaps an `SDLTexture` or `SDLSurface` for images, and so on. Internally, these might hold unsafe pointers to the corresponding SDL structures (like `SDL_Window *`), but the pointers are never exposed publicly [5] . This encapsulation follows the principle already used in Teatro's stubbed SDL backend: **SDL specifics are hidden behind Swift types** [5] . The Swift layer will handle converting Swift data types to C (and vice versa) as needed, and manage resource lifecycles (ensuring every `SDL_Create*` is paired with the appropriate `SDL_Destroy*` at the right time).

Core subsystems separation: Within SDLKit's Swift code, we delineate the major subsystems of SDL to keep code organized and allow optional features:

- **Windowing & Graphics:** This includes window creation, context management, and 2D drawing. Likely types: `SDLWindow` and `SDLRenderer`. `SDLWindow` handles opening an OS window, setting up an OpenGL/Metal/Vulkan context if needed (SDL3's default renderer backends), and high-level window operations (title, size, fullscreen toggle, etc.). `SDLRenderer` provides drawing APIs – e.g., methods to clear the screen, draw shapes or images, and present the frame. Under the hood it wraps an `SDL_Renderer` (if using SDL's rendering API) or possibly uses SDL surfaces for simple software rendering. This separation mirrors the stub design where `SDLWindow` and `SDLRenderer` are distinct [6] , and ensures that rendering logic (which might be GPU accelerated) is managed apart from window event logic.

- **Input Events:** SDLKit will capture input devices and user events via an `SDLEvent` abstraction. In the stub version, `SDLEvent` is an enum of cases like `.quit`, `.keyDown`, `.mouseMove` etc. [7] . The real SDLKit will map actual `SDL_Event` types to this Swift enum (or a similar Swift structure). There will likely be an event polling mechanism (e.g., an `SDLEventPump` or simply a method on `SDLWindow` or an `SDLApp` class) to retrieve events from SDL's queue. We keep event polling **non-blocking and Swift-y** – e.g., `window.pollEvent()` returns an `SDLEvent?` as in the stub [8] . Keyboard keys and mouse coordinates will be translated to Swift types (e.g., key codes can be Int or a custom `KeyCode` enum). By abstracting events, **the higher-level code (like FountainKit or Teatro controllers) doesn't depend on SDL's C event structs**, just on the Swift enum or callback.

- **Audio Output (Optional):** SDL also provides audio device management and playback capabilities. While primary audio synthesis and music in Fountain-Coach is handled by specialized modules (e.g. TeatroAudio with Csound/FluidSynth [9]), SDLKit can optionally expose basic audio output for simpler use cases or cross-platform sound effects. The architecture will treat audio as a separate concern (perhaps an `SDLAudio` singleton or `SDLAudioDevice` object). SDL's audio API could be wrapped so that you can open the default audio device and play raw audio buffers or perhaps simple WAV files. However, **this subsystem will be optional and minimal** – many FountainAI apps will continue to use the dedicated audio engine. SDLKit's audio integration plan is mainly to have a fallback or simple sound support (for example, to beep or play a short sound without needing Csound). This might be rolled out after windowing and input are stable, and possibly behind a feature flag or separate target if it introduces extra dependencies (like linking SDL_mixer or other extension libraries).

- **Font/Text Rendering:** A notable extension of SDL for GUI apps is text rendering via SDL_ttf (the TrueType font library). SDLKit's architecture anticipates a **text rendering module** so that the AI or GUI can draw text to the window (e.g., subtitles, labels, or debugging info). Initially, SDLKit might not include font rendering (perhaps using simple shapes or bitmapped text), but the plan is to integrate SDL_ttf for proper font support. This will likely involve an `SDLFont` or `SDLTextRenderer` class that can load fonts and create text textures. We keep this separate from core rendering to avoid forcing font dependencies on all users. In the **progressive rollout**, font support comes when needed, aligning with the Teatro roadmap which plans to expand GUI fidelity to include fonts and rich text [10].

Memory and resource management: Each SDLKit Swift object will take responsibility for the underlying SDL resource. For example, `SDLWindow` when opened will call `SDL_CreateWindow`, and when closed (or on deinit) will call `SDL_DestroyWindow`. Similarly, `SDLRenderer` will manage the SDL renderer/context. This RAII-style management ensures that we don't leak OS resources. We will also convert SDL's error codes or null-pointer returns into Swift `Error` throwings or optionals, making error handling more idiomatic. Initialization of SDL (the `SDL_Init` call) will be handled behind the scenes: likely the first time an SDLKit object is created, the library will ensure `SDL_Init` is called for the relevant subsystems (video, audio, etc.) if not already done. We might provide an explicit `SDLKit.initialize()` for clarity, but it can be optional – the library can lazy-init on first use. SDLKit will also ensure thread-safe usage where appropriate: for instance, most SDL calls should happen on the main thread (especially window creation on macOS). We may enforce this by documenting that certain calls be made from the main thread, or even internally dispatch to main. The design is such that **concurrency is respected** – the stub classes are already marked `Sendable` where possible and use locking for event queues [11] [8]; the real implementation will continue that approach (e.g., protecting event queue access and ensuring no race conditions around window state).

Swift-Friendly Public API Overview

The public API of SDLKit is designed to feel like a natural Swift framework rather than a C binding. Here we outline the expected key types and how they abstract away SDL's C pointers and functions:

- `SDLWindow` – Represents a window or screen. This class wraps an `SDL_Window` pointer internally but exposes high-level properties and methods. For example:
- Initializer: `SDLWindow(config: SDLWindowConfig)` to create a window object with a given configuration (title, width, height, vsync, etc.). There is a config struct (`SDLWindowConfig`) to bundle common window parameters [12].
- `open()` method: actually opens the window on the screen (calls SDL_CreateWindow and related setup). If unsuccessful, it throws an error (encapsulating SDL_GetError message). On success, it sets up an associated rendering context (e.g., SDLRenderer).
- `close()` method: destroys the window (SDL_DestroyWindow) and cleans up. After closing, the window object can either be discarded or potentially reopened.

- Properties like `isOpen` (or an `openState` Bool as in stub [13]) to check if the window is currently open, `size` (width/height), `title`, etc., with getters/setters that call SDL functions (e.g., SDL_SetWindowTitle).

Importantly, `SDLWindow` **hides the pointer** – consumers never call `SDL_Init` or `SDL_CreateWindow` themselves; they just use this class. This is already foreshadowed by the stub: the stub `SDLWindow` provides an identical interface without exposing any SDL types [14]. SDLKit will implement those methods for real. The class will be `Sendable` (likely using locking or ensuring it's mostly used on one thread) so that it can be safely orchestrated by concurrent code (like an AI agent sending a command to open a window).

- `SDLRenderer` – Manages drawing operations for a window. Typically, SDL's model is to create a renderer tied to a window (SDLKit might create one automatically inside `SDLWindow.open()`). In the stub design, `SDLRenderer` was initialized with a width and height and maintained an off-screen pixel buffer for testing [15]. In SDLKit's real version, `SDLRenderer` will likely wrap an `SDL_Renderer` (obtained via `SDL_CreateRenderer`) if using the SDL render API, or handle an SDL surface for software rendering. Public API methods might include:
- `clear(color: SDLColor)` – clears the current drawing target with the given color (SDL_SetRenderDrawColor + SDL_RenderClear behind the scenes). The color could be an `SDLColor` struct or just a UInt32 ARGB as in stubs [16].
- `drawRect(x:y:width:height:color:)` – draws a filled rectangle (internally uses SDL_RenderFillRect or similar). The stub version fills an array [17]; the real one will draw to the actual GPU context or surface.
- `present()` – displays the rendered content on the window (calls SDL_RenderPresent). In stub it's a no-op [18], but in real it swaps the buffers.
- Potentially, methods for drawing lines, images, or text if needed (or those might be added later).
- The `SDLRenderer` might also handle creating textures from image files (wrapping SDL_CreateTextureFromSurface, etc.), possibly via convenience functions.

Just like `SDLWindow`, the `SDLRenderer` ensures no raw pointers or manual surface handling is exposed to the user. It may also include utility methods to load images (if SDL_image is used later) or to capture the current frame (for screenshot or testing, e.g., getting a pixel buffer). For now, core methods are simple draw calls. The API will use Swift types (Int for coordinates, maybe a custom `Rect` struct, etc.) to be idiomatic. The stub's existence proves the interface can be implemented without SDL (for tests) – it currently keeps a pixel buffer and even allows computing a hash of the image for test assertions [19]. In the real SDLKit, tests might either still rely on stub mode or we could offer a way to read pixels back via SDL (e.g., `SDL_RenderReadPixels`) to verify drawing. Either way, the public interface of `SDLRenderer` remains the same in both modes, demonstrating how **SDLKit abstracts away the difference between stub and real at the API level**.

- `SDLEvent` **and Event Handling** – SDLKit will provide an enumeration or struct to represent input events, so that higher-level code doesn't handle SDL's C `SDL_Event` union directly. The stub defines an enum with cases `.quit`, `.keyDown(keyCode:)`, `.mouseMove(x:y:)`, etc. [7]. We will mirror this in the real implementation. Likely, `SDLEvent` will be a `public enum SDLEvent` with similar cases, possibly expanded to include more event types (e.g., `.keyUp`, `.mouseUp`, window events, controller events if needed, etc.). Each case carries relevant data in Swift types (ints for key codes, or perhaps a higher-level Key enum if we map common keys). SDLKit will translate from SDL's event system to this enum. For example, internally: when SDL_PollEvent returns an `SDL_Event` of type `SDL_KEYDOWN`, we construct an `SDLEvent.keyDown(keyCode:)` with the key code.

The **public API for polling events** could be a method on `SDLWindow` (like `pollEvent()` as stub has [8]) or a global SDLKit function if we prefer. We may also provide a way to wait for events or to push synthetic events. The stub already has `pushEvent(_:)` to inject a synthetic event into the queue (used in tests to simulate a quit event, for example [20]). In real SDLKit, `pushEvent` could call SDL_PushEvent with a custom event type or simply be unavailable (tests might stick to stub mode). However, exposing a `pushEvent` could be useful for, say, programmatically closing windows (`pushEvent(.quit)`). We will likely keep it for parity with stub (marked only for testing or internal use).

Overall, event handling in the public API looks like: you call `while let ev = window.pollEvent() { ... handle ev ... }` in a loop or tick, and your logic deals with the high-level `SDLEvent` cases. This design ensures **the rest of the Fountain or Teatro codebase remains SDL-agnostic**, handling input in terms of Swift enums.

- **Additional Types**: We expect to introduce some supporting types for completeness and safety:
- `SDLColor`: a struct or typealias for color values (to avoid using raw UInt32 for RGBA everywhere). This could provide convenience initializers (e.g., from red,green,blue,alpha components) and ensure consistent byte ordering.
- `SDLPoint` or `SDLRect`: basic geometry types (though we might just use `CGRect` or similar if bridging to CoreGraphics on mac, but a simple integer rect type might be useful for cross-platform without importing heavy frameworks).
- `SDLKeyboardModifiers`, `SDLKeyCode`: we might create Swift enums or option sets for key modifiers (Shift, Ctrl) and key codes, instead of using SDL's integer constants. Initially, key events may just give an integer keycode (as stub does), but long-term a more descriptive key system (like mapping to Unicode scalar or a key enum) could be part of the API for clarity.
- `SDLAudioDevice` / `SDLAudioSpec`: if audio features are exposed, we might have types to represent an opened audio device or an audio format spec. This would abstract SDL's `SDL_AudioSpec` and callbacks into perhaps a simpler Swift closure-based playback mechanism (for example, registering a callback to provide audio samples). This is a more advanced area and would likely be in a later phase of SDLKit.

Importantly, **all these public types will be well-documented and marked as `Sendable` where appropriate**, to integrate with Swift's concurrency model. The API will be asynchronous only where needed (for example, we might not need async/await for rendering, since it's typically synchronous drawing calls, but if we ever provide an async event stream, we would consider `AsyncSequence` for events). At this stage, a straightforward imperative API is envisioned, which higher-level frameworks or the AI system can drive.

To illustrate a brief usage of the SDLKit API in practice (hypothetical):

```swift
import SDLKit

let window = SDLWindow(config: .init(title: "Demo", width: 800, height: 600))
try window.open()
let renderer = SDLRenderer(for: window)
renderer.clear(color: SDLColor.black)
renderer.drawRect(x: 100, y: 100, width: 200, height: 150, color:
SDLColor(red: 0, green: 255, blue: 128))
renderer.present()
```

```
// Event loop example (simplified)
while let event = window.pollEvent() {
    switch event {
    case .quit:
        window.close()
    case .keyDown(let code):
        print("Key down: \(code)")
    default:
        break
    }
}
```

In reality, the `SDLWindow.open()` might create the `SDLRenderer` internally, but the above pseudo-code conveys the feel: **simple, high-level calls** like `clear()` and `present()` instead of dealing with pointers or SDL context directly. This aligns with the goal that *"all types avoid leaking SDL details"* [21] in the public interface.

Integration with FountainKit and AI-Directed GUI Flows

One of the most exciting purposes of SDLKit is enabling **AI-directed GUI flows** in the FountainAI ecosystem. FountainKit (the AI/control system) can leverage SDLKit to allow AI agents or execution plans to create and manipulate GUI elements in real time, closing the loop between an AI's decisions and user-facing visualization. Here's how SDLKit will integrate:

Agents and tools for the planner: FountainKit's planner and tool invocation system will be extended to include SDLKit actions as available "tools" the AI can use. In practice, this means defining an AI Agent (or a set of functions) that the AI can call to perform GUI operations – for example, an agent that can **open a window, draw content, update the display, and even read user input events**. (We detail the agent structure in *Appendix: AGENTS.md* at the end of this document.) When the AI's plan determines a GUI step is needed – say the AI wants to show a preview of a screenplay scene or prompt the user with a visual selection – the planner can insert a call to SDLKit's functions via this agent.

Under the hood, this likely uses FountainKit's function-calling mechanism: the planner can output an intention like "call function `open_window` with parameters X" which the Fountain function-caller service will route to SDLKit's agent. Because FountainKit is modular, we can integrate SDLKit at the gateway or tool-server level so that the AI's requests result in real GUI actions. For example, an **AI plan might look like**:

- Step 1: Generate content (some text or media via other tools or LLM).
- Step 2: Call `GUI.openWindow("Title", width:..., height:...)` – this triggers SDLKit to create a window on the user's machine.
- Step 3: Call `GUI.drawText("Hello World", x:100, y:100)` – SDLKit renders the text to the window (assuming we have text rendering available, otherwise maybe draw shapes).
- Step 4: Perhaps wait for an event or user input.
- Step 5: When an event occurs (e.g., user presses Esc or closes window), SDLKit's agent captures that and can notify FountainKit, which might then plan the next step (like closing the UI or proceeding accordingly).

Realtime feedback loop: Integration isn't just one-way (AI to GUI); SDLKit will also feed events back into the Fountain system. When the user interacts with the SDLKit window (presses a key, clicks a button, etc.), those events can be propagated as triggers for the AI. For instance, SDLKit could be configured to send a signal or call a callback in FountainKit when certain events occur (like `.quit` event could signal the end of an interactive session). This could be done by publishing an event to FountainTelemetry or a similar mechanism used for MIDI events. FountainKit already handles streaming events (e.g. MIDI streams in FountainTelemetryKit for interactive music control), and SDLKit will align with that. We might treat GUI events as another stream of events the AI agent can consume. In practical terms, SDLKit's integration will likely involve an **observer or delegate** pattern: the SDLKit agent will have a way to register event handlers or push events into the planning system's queue. For example, the agent might convert an SDLEvent (like a button click) into a structured data message that the planner can interpret (perhaps akin to a function call result or an environment update).

Use cases for AI-directed GUI: Having this integration means an AI-driven application could, say, **open a preview window of a screenplay timeline automatically** when the user requests it in natural language, or an agent could orchestrate a step-by-step tutorial by drawing highlights on the screen. Another scenario is an AI tool that visualizes data: the AI could call SDLKit to draw a chart or image as part of its answer to the user. All of this happens through the defined agent interface, ensuring the AI doesn't execute arbitrary GUI code but only the controlled, safe operations exposed by SDLKit.

Consistency with FountainAI architecture: The design will ensure that using SDLKit in FountainKit flows does not block the asynchronous, multi-step nature of the system. SDLKit calls (like opening a window) will be non-blocking from the perspective of the AI's services – e.g., the function-caller can invoke the window creation and immediately return control to the planner, while the window persists independently. If the AI needs to wait for a user event (like "press any key to continue"), that can be modeled as the planner waiting for a specific callback or event from SDLKit (which could be achieved by a synchronization mechanism or a state in the conversation managed by the gateway). In summary, SDLKit will empower FountainAI to have a GUI element in its otherwise headless toolset, bringing visual interactivity to the platform. This is a key step to making FountainAI not just a backend brain, but an interactive system that can show previews, capture user input via GUI, and incorporate that into its decision loop.

Lastly, security and control are considered: since SDLKit can create windows and potentially read input, only authorized agents or certain plans should use it. This likely means the **planner will only invoke the GUI agent for appropriate objectives (like debugging, previews, user-initiated flows)**, and the `tools-factory` could enable or disable it based on environment (for example, maybe disabled in a server context or when running in a headless CI environment). But when enabled, SDLKit's integration offers a powerful visual extension to the FountainKit's capabilities.

Relationship to TeatroKit's Current Design (Stubs) and Transition Plan

SDLKit is essentially the real implementation that will **supersede the current stubbed SDL layer in TeatroKit**. In the existing TeatroKit (specifically the `TeatroGUI` module), there is a namespace `TeatroSDLBackend` which contains placeholder implementations: `SDLWindow`, `SDLRenderer`, `SDLEvent`, `SDLRunLoop`, etc., all of which are *stubs* meant to simulate a GUI without actually using SDL. These stubs were introduced to allow the codebase to compile and run deterministically in the absence of SDL3, and to enable incremental development under a feature flag [14]. For instance, the stub `SDLWindow` simply flips an internal flag on open/close and manages a queue of synthetic events [11] [8], and the stub `SDLRenderer` draws into an array for tests instead of an actual window [22]. This was crucial for testing and CI, as noted: the initial interactive GUI was delivered with "SDL-free stubs" [23] to preserve determinism and not require a graphics environment.

With SDLKit, those abstractions will be **backed by real SDL operations**. We plan the transition as follows:

- **SDLKit provides the concrete types:** We will implement `SDLWindow`, `SDLRenderer`, etc., within SDLKit (with the same or very similar APIs as the stubs). TeatroGUI will then **depend on SDLKit** and use these types. Ideally, the names and method signatures remain identical, so much of TeatroGUI's higher-level code (like `TeatroPreviewController`) doesn't need to change – it can just import SDLKit and remove or alias its own stub types. For example, Teatro's preview controller currently does `window = SDLWindow(config: ...)` and `renderer = SDLRenderer(width: ...)` [6], and calls `window.open()`, `renderer.clear()`, etc. [24]. After integrating SDLKit, those same calls will invoke the real implementations. The stub definitions can be conditionally compiled out or removed entirely once SDLKit is stable.

- **Non-breaking API:** We will take care that SDLKit's API either matches or easily replaces the Teatro stubs. The stubs were intentionally minimal and designed to mimic what a real SDL backend would do (without exposing any SDL types to the outside) [5]. This means the preview controller and any other client code see the same interface. Thus, swapping in SDLKit should not break those clients. For instance, `SDLEvent` enum will remain, just now it will be populated by actual SDL events instead of synthetic ones. The `SDLRunLoop` (which runs the frame tick loop at target FPS) can remain mostly the same logic, except where it polls events it will now call into SDLKit's event pump instead of using `pushEvent` hacks. Because we preserve the API contract, **external code using Teatro's preview API should see no difference except that things now actually display on screen.** This fulfills the promise that we avoid breaking public interfaces while changing internal modules [21].

- **TeatroPreviewController and others use SDLKit:** Internally, we will modify Teatro's `TeatroPreviewController` (and any related classes) to initialize and manage real SDL windows. For example, where it previously did `if let ev = window.pollEvent(), ev == .quit { stop() }` [25] with stub events, it will now be truly polling the SDL event queue – but from the controller's perspective it's the same call. The difference is that now, if a user clicks the window's close button or presses a key, `window.pollEvent()` will return a real event (e.g., `.quit` or `.keyDown`), and the preview will respond. This is a huge improvement: rather than only having a timed auto-quit as now [20], the preview will respond to actual user input.

- **Removal of stub code / Feature flag:** During development, we might keep the stub in place under a compile flag (e.g., `USE_STUB_SDL`) so that tests can run in a headless environment by disabling real SDL. Over time, however, the goal is to have SDLKit be robust enough that we might not need the stub at all, except perhaps in some CI configurations. Initially, we will likely **keep the stub as a fallback** – possibly by making SDLKit itself support a "no-SDL mode" (if SDL3 library is unavailable, operations either no-op or throw with a clear message), or by toggling which implementation is compiled. The Teatro documentation already mentions a feature flag controlling the SDL path [14], and that approach will continue: e.g., in CI we might compile with `-DTEATRO_NO_SDL` to use stubs. But the long-term vision is that most development and production builds include SDLKit's real implementation, and the stub is only used in purely headless automated testing scenarios.

- **Validating against stub behavior:** We will ensure that SDLKit's functionality covers everything the stubs intended. The stub was used for testing deterministically (like computing a pixel hash to verify drawing [19]). We will replicate those tests either by running the stub mode or by enhancing SDLKit to allow a "software rendering mode" for tests. For example, SDL has the ability to render to an off-screen surface which we could hash, or we use the stub itself in test builds. This way, our test suite (which likely has tests like "drawRect on renderer then snapshotHash matches expected value") will continue to pass, either by staying in stub mode or by adjusting the testing strategy. This double-check ensures that the **SDLKit-based implementation is correct and doesn't regress functionality that was implicitly tested via stubs**.

In summary, SDLKit will **take over the responsibilities of TeatroSDLBackend**. Once integrated, TeatroGUI becomes essentially a higher-level layer orchestrating the preview, but all calls dealing with windowing, rendering, or input delegate to SDLKit. The `TeatroSDLBackend` target can be deprecated. We will document this transition clearly: e.g., in Teatro's release notes we'll note that those types are now provided by SDLKit. The benefit is that TeatroKit codebase gets cleaner (no need to maintain parallel stub logic for long), and any other module (outside Teatro) that wants SDL capabilities can now import SDLKit directly rather than relying on Teatro's internal stubs.

Finally, by basing SDLKit on the stub API design, we preserve the **encapsulation of SDL**. Notice that nowhere in Teatro's public API did an `SDL_Window` or similar appear – this remains true with SDLKit. Even though under the hood we now call real SDL functions, the rest of Fountain-Coach ecosystem continues to operate on the high-level constructs. This was exactly the plan: the Teatro docs indicated "introduce real SDL3 bindings and swap-in the concrete implementations behind the same APIs" [26] – SDLKit is the fulfillment of that plan.

Roadmap: Progressive Rollout, Platform Support, and Future Extensions

SDLKit's implementation will be rolled out in phases, with careful consideration for cross-platform support and future features like fonts and audio. Below is a roadmap outlining the major milestones and plans:

Phase 1: Module Scaffolding and Basic Windowing (MVP)
Goal: Set up the SDLKit package and get a basic window showing on screen on macOS and Linux.
- **Project setup:** Create the `SDLKit` repository/package with the structure described (system library target for SDL3, Swift target for API). Verify that we can compile and link against SDL3 on a developer machine. This includes writing Package.swift to find SDL3 via pkg-config or other means.
- **Window creation and teardown:** Implement `SDLWindow.open()` and `.close()` using SDL3. For MVP, this means initializing SDL's video subsystem, creating a window (with an OpenGL or default renderer context), and destroying it properly. Also implement basic event polling (SDL_PollEvent) to get quit events. At this stage, we can likely open a window, perhaps clear it to a single color, wait a couple seconds, then close – just to prove it works.
- **Integrate with Teatro (behind flag):** Use a Swift compile-time flag or runtime check to swap TeatroPreviewDemo to use the real SDLKit when available. We should be able to manually launch the preview on macOS and see a window. This corresponds to the Teatro refactor milestone of *"Implement basic SDL window open/close in TeatroGUI, tie a render callback, Esc to close"* [27]. We will test on macOS first, and also on a Linux environment (perhaps using X11 forwarding or a dummy display) to ensure it runs there [28].

Phase 2: Rendering and Basic Drawing
Goal: Get `SDLRenderer` drawing content in the window, matching the stub's capabilities (clear screen and draw rectangle).
- **Renderer context:** Create an `SDLRenderer` when window opens (SDL_CreateRenderer with

appropriate flags, or use SDL_GetWindowSurface for a simpler approach first). Implement `clear()` to set draw color and clear the screen [29], and `present()` to update the window.
- **Shape drawing:** Implement `drawRect` and possibly `drawLine` using SDL's rendering functions (SDL_RenderFillRect, SDL_RenderDrawLine, etc.). For now, focus on solid color fills as in stub. Verify that the moving rectangle demo in `TeatroPreviewController` (which uses `drawRect` each frame [30]) now visibly animates on screen at the expected FPS.
- **Color handling:** Introduce an `SDLColor` type or use SDL's `SDL_Color` for convenience. Ensure color values (like 0x33CC99FF used in the demo [31]) are interpreted correctly (likely as RGBA).
- **Testing:** On macOS and Linux GUI, we manually test the preview. For CI, we might still run in stub mode for this phase, but we can set up a basic automated test where we open a window and immediately close it to ensure no crash (perhaps under Xvfb on Linux to simulate a display [32]). We'll also ensure that the stub's pixel-hash test still runs in stub mode.

Phase 3: Input Handling and Interactive Control
Goal: Support user input events and close the loop for interactivity.
- **Keyboard and mouse events:** Implement translation of SDL_Event to SDLEvent. Focus on window close (SDL_QUIT), key down/up (SDL_KEYDOWN/UP), and mouse events (motion, button down/up). Test that pressing Escape in the window triggers a `.quit` event that our preview loop catches to exit (matching the stub behavior of scheduling a quit after 2s [20], but now user can truly exit).
- **Event loop improvements:** Possibly introduce an `SDLRunLoop` or integrate with the existing one to use real polling. The stub `SDLRunLoop.run(tick:shouldContinue:)` currently calls our tick closure and uses `window.pollEvent()` inside [25]. In real mode, we might do similarly. Ensure that `SDLRunLoop` can run at 60 FPS without busy-waiting excessively (likely it already sleeps appropriately in stub). With real SDL, we might use SDL_Delay or just `usleep` for frame timing.
- **AI event integration:** At this phase, we can start hooking events to FountainKit if possible. For example, if we have the SDLKit agent ready, test that when a window opens and a key is pressed, we can route that info back to a log or trigger in the AI. This might require some scaffolding in FountainKit's tool-server (perhaps an RPC or callback registration in the agent). It's partly outside SDLKit's scope to implement FountainKit's handling, but we ensure SDLKit provides the necessary hooks (like a delegate or a publisher for events).
- **Cross-platform check:** By Phase 3, the core interactive pieces are in place. We should confirm that everything still works on Linux (ensuring that SDLKit finds an OpenGL context or software fallback in a headless CI). This may involve configuring Xvfb in CI to have an environment for SDL [33]. If any platform-specific issues arise (e.g., different key code values on Windows), we address them.

Phase 4: Font Rendering via SDL_ttf (Planned Extension)
Goal: Add ability to draw text in windows.
- *SDL_ttf integration: Include SDL_ttf as an optional dependency. Likely this means adding another system library target* `CSDL_ttf` *and linking against* `libSDL3_ttf`*. Provide a high-level API like* `SDLFont` *or simply extend SDLRenderer with a method* `drawText(x:y:text:font:color)`*. Implementation: load a TTF font (TTF_OpenFont) into an* `SDLFont` *object, then create an SDL_Surface for the rendered text (TTF_RenderUTF8_Blended, etc.), convert that to an SDL_Texture, and render copy it to the renderer. Manage caching of fonts and maybe glyph metrics if needed for performance.*
- *Public API changes: Introduce a way to specify fonts (by file path or system font, etc.) and perhaps text alignment. At first, keep it simple: one function call to draw a given string in a given font and color at a position. Document that this requires the SDL_ttf feature. Possibly guard it behind* `#if canImport(CSDL_ttf)` *so that if the library isn't available, we either don't compile those functions or throw at runtime.*
- *Use case: With this in place, the AI could, for example, call* `drawText` *to label something in the window. Teatro's preview might use it to display MIDI or token streams as overlay text. This aligns with plans to have*

text and overlays for device routing etc. [26] .
- *Testing:** We will test that text appears correctly on supported platforms. Also ensure fallback: if SDL_ttf isn't present (or on CI where we might not want to install it), the absence is handled gracefully (e.g., the function is no-op or error). This extension is important for future usability but is not strictly needed for initial SDLKit functionality, hence its later phase.

Phase 5: Audio Output (Exploration)
Goal: Provide basic audio playback support via SDL, if deemed useful.
- *Audio device init: Use SDL's audio subsystem to open default audio device. Possibly create an* `SDLAudio` *singleton with methods like* `openAudioDevice(format:)` *and* `playSound(buffer:)` *or even higher-level* `playAudioFile(path:)`*. This could allow simple playback of WAV files or raw PCM buffers through SDL.*
- *Integration with TeatroAudio: Evaluate if SDL audio could help in the Fountain context. TeatroAudio already handles MIDI synthesizers and likely outputs audio through CoreAudio/ALSA on its own. One idea is to route TeatroAudio's output into SDL audio if we want to unify output handling – but that might be redundant. Another simpler use: if an AI agent needs to play a sound (like a notification), SDLKit audio could do that without spinning up a full Csound engine.*
- *Scope decision: We will likely keep SDLKit's audio limited to non-MIDI use cases (playing short sound clips or providing audio feedback). For anything complex or music-related, FountainKit will use its dedicated audio pipeline. We document this clearly so users know when to use SDLKit audio vs TeatroAudio.*
- *Testing:** On each platform, ensure we can hear a short sound. Automated testing of audio is tricky (could check that the device opens and perhaps that no error is returned). This feature might remain off by default or require explicit enabling due to the complexity it adds.

Phase 6: Windows Support and CI Stability
Goal: Expand full support to Windows and ensure CI can test SDL features reliably.
- *Windows build: Modify Package.swift to find SDL3 on Windows (perhaps using environment variables or instruct users to place SDL binaries in a known location). There might be challenges with linking (SDL might be available as a static .lib and a DLL; SwiftPM on Windows might need a modulemap). We will create documentation or scripts to assist setting up SDL on Windows for development. Once configured, the SDLKit API code should largely be portable, but we'll pay attention to any differences (like window creation flags on Windows, or code page for file paths, etc.).*
- *Testing on Windows: Perform manual tests of opening windows, drawing, etc., on Windows. Ensure that key events map properly (SDL might give scancodes or virtual key codes – we'll test common keys like letters, escape, etc.). Adjust SDLEvent mapping if needed for consistency.*
- *Continuous Integration: Introduce CI jobs that run SDLKit's test suite in an environment with graphics. On Linux, use Xvfb to simulate an X display* [32] *, so we can open a window in a virtual framebuffer. Possibly run a short headless mode where we create a window, render, post a quit event, and verify it exits (similar to the TeatroPreviewDemo logic, but fully automated). On macOS CI, if available, do a basic run as well (though some CI might not allow GUI – but creating a window and immediately closing might still work). For Windows CI, we'd ensure the runner has SDL installed or include it in the repo for testing. The key is verifying that the real SDL path runs without crashing and behaves as expected even in automated scenarios.*
- *Feature flag cleanup:** Once CI proves stable, we can consider making SDL the default and removing or reducing the stub usage. Perhaps maintain an environment variable like `SDLKIT_USE_STUB` for special cases, but default to real. This will mark the completion of the core SDLKit integration.

Phase 7: Documentation, Examples, and Future Ideas
Goal: Finalize documentation and outline further enhancements.
- *Comprehensive docs: Write README and usage guides for SDLKit, similar in spirit to TeatroGUI's documentation. This will include basic how-to (like the pseudo-code earlier) and notes on how it integrates with FountainAI. We'll also update the top-level Fountain or Teatro docs to reference SDLKit where appropriate,*

explaining that SDLKit must be included for GUI features.
- Example integration: Possibly create an example in `FountainExamples` *that uses SDLKit directly. For instance, a demo that opens a window and draws something under control of an AI script. Or adapt the existing* `HelloFountainAITeatro` *to actually show the Teatro preview window when run (currently it might just print outputs). This serves as both a test and a usage example for developers.*
- Roadmap for future: Mention possibilities like integrating SDLKit with SwiftUI in the future (since TeatroGUIViews might have SwiftUI overlay [34], one could embed SDL views in SwiftUI if desired), or using advanced SDL3 features (multi-window support, game controller input, etc.) as needs arise. These are beyond initial scope but leaving the door open is good.
- SDL versioning and maintenance: Note that SDLKit is built for SDL3 (the latest major version). We should monitor SDL releases for any changes. Possibly, if needed, allow conditional support for SDL2 as fallback (but since SDL3 is the target, likely not necessary).
*- Community and Contribution:** Invite contributions, especially if others want to extend platform support or features (like adding a new drawing API or supporting a new SDL extension). Because SDLKit is modular, external contributors can focus on this package alone, which is easier to manage.

Throughout this roadmap, each stage ensures **progressive enhancement** without breaking existing functionality. Early phases ensure internal parity with stubs and basic user-visible results (window opens). Middle phases bring interactive parity (responding to input just as planned). Later phases broaden capabilities (text, audio) and environments (Windows). By the end, SDLKit will be a robust Swift SDL3 wrapper, powering Teatro's GUI and any other GUI needs in FountainAI.

One significant planning aspect is **the fallback stub mode**. We will continue to support a stub or headless mode via feature flags or dynamic checks to avoid forcing SDL in contexts where it can't run (e.g., a continuous integration server with no display, or a server deployment of FountainAI with no GUI). This could be implemented by compiling out the actual SDL calls when a certain Swift flag is set, or by making SDLKit initialize in a "dummy" mode if it detects no display (for instance, SDL can be started with `SDL_VIDEODRIVER=dummy` environment on Linux to not actually open a window). The documentation will include how to enable stub mode for testing. This approach aligns with prior mentions of using dummy devices in CI [28]. The outcome is flexibility: developers get real windows when needed, but tests can still run offscreen deterministically when required.

Finally, regarding **SDL extensions**: beyond SDL_ttf and basic audio, there's SDL_image (for loading PNG/JPG easily), SDL_mixer (more advanced audio), etc. We don't plan to incorporate those from day one, but SDLKit's structure will allow adding them if a need arises. For example, if down the line FountainAI wants to display an image in the GUI, we could add SDL_image support to load files into textures. These would follow the same pattern: a small C binding if needed and a Swift API wrapping it.

Conclusion

SDLKit will bring a crucial capability to the Fountain-Coach suite: a cross-platform, Swift-native interface to graphics and input, integrated with our AI systems. By isolating SDL3 in its own module, we adhere to the organization's modular architecture philosophy and enable broad reuse of GUI functionality. We've detailed how SDLKit's design will look – from clearly defined subsystems (windowing, rendering, input, etc.) to a Swifty API that hides C pointers and embraces Swift conventions [5]. We've also outlined how SDLKit plugs into FountainKit to allow AI-driven GUI interactions, which opens up new frontiers for user interaction in FountainAI.

This architecture and plan mirror the approach taken in modularizing Teatro and FountainKit themselves, underscoring consistency. The rationale for SDLKit is clear: **focus, reuse, and future-**

proofing. As the Fountain-Coach ecosystem grows (with complex previews, interactive editors, or tutor dashboards), having a dedicated SDL layer means we implement low-level GUI logic once and leverage it everywhere. The roadmap ensures we start simple (get windows drawing) and incrementally layer on sophistication (text, audio, multi-platform) while keeping the system stable via feature flags and tests.

Upon completion of SDLKit's rollout, TeatroGUI will seamlessly use it under the hood, and developers can forget there was ever a stub – things will "just work" with real windows popping up and real events flowing. FountainAI agents will gain a new dimension: not just text and API calls, but the ability to **create windows and graphics on the fly**, orchestrated by AI. This synergy of AI and GUI, enabled by SDLKit, aligns perfectly with Fountain-Coach's mission to blend creative AI with interactive tools.

The following appendix defines the SDLKit agent structure in FountainKit, solidifying how AI plans invoke SDLKit:

Appendix: AGENTS.md – SDLKit Interactive GUI Agent Integration

Agent Name: `SDLKitGUIAgent` (working title) – This is the root AI agent interface provided by SDLKit for the FountainAI platform. It may also be referred to as the "GUI Agent" in planning contexts.

Role and Scope: The SDLKit GUI Agent is responsible for all interactive graphics operations initiated by the AI. It acts as a bridge between high-level AI plans and the low-level SDLKit API. The agent encapsulates what the AI can do in a GUI: opening/closing windows, drawing or updating content, and controlling the event loop or responding to user input. It does *not* handle any AI logic itself; instead, it exposes functions (tools) that the AI's planner and function-caller can utilize. In essence, this agent gives the AI "eyes and hands" in the GUI world – it can display information to the user and observe basic user interactions.

Agent Capabilities (Exposed Functions): The SDLKitGUIAgent provides a set of callable actions to the AI. These actions are presented as function calls (e.g., in OpenAI function calling format or as distinct tool endpoints in the Tools API) so that the planner can include them in an execution plan. The core capabilities include:

- `openWindow(title: String, width: Int, height: Int) -> WindowID` – Open a new window with the given title and dimensions. Returns a `WindowID` handle (perhaps an integer or string token) that the AI can use to reference this window in subsequent calls. This allows multiple windows if needed (though many scenarios will use just one). Under the hood, this calls `SDLWindow.init` and `open()`. If a window cannot be opened (error or unsupported), the agent returns an error message which the planner can handle (e.g., fallback or inform the user).

- `closeWindow(window: WindowID) -> Void` – Close a previously opened window. This triggers SDLKit to destroy the window resource and remove it from tracking. The agent will also stop any rendering loops associated with that window. If the window was not open, it's a no-op (or returns an error that the planner could catch).

- `drawText(window: WindowID, text: String, x: Int, y: Int, font: String?, size: Int?) -> Void` – Render a string of text in the specified window at the given coordinates. This uses SDLKit's text rendering (requires SDL_ttf). The font and size parameters can be optional; if not provided, a default font/size is used. This function allows the AI to display dynamic textual information to the user (for example, showing a line of dialogue or a label on a visual element). The agent ensures this call results in updating the window's content (e.g., by using SDLRenderer to draw the text and present the frame). If text rendering is not yet supported (before SDL_ttf integration), the agent may respond with an "unimplemented" error or fall back to drawing a placeholder (like a colored rectangle or nothing).

- `drawRectangle(window: WindowID, x: Int, y: Int, width: Int, height: Int, color: Color) -> Void` – Draw a filled rectangle of the given color (and possibly support other primitives like lines or circles in the future). This directly leverages SDLKit's `SDLRenderer.drawRect`. The purpose is to allow highlighting regions or simple graphics. The `Color` might be a string or hex code in the API; the agent will parse it into an SDLColor.

- `present(window: WindowID) -> Void` – Update the window display. In many cases, the drawing functions (`drawText`, `drawRectangle`, etc.) could implicitly present, but it might be useful to have an explicit present function so that the AI can do a batch of drawing commands then present once (reducing flicker). This calls SDLKit's `renderer.present()`. If omitted, we might design the draw functions to auto-present, but having it gives the AI control (in case it wants to draw multiple things before showing them).

- **(Potential)** `captureEvent(window: WindowID, timeout: Int?) -> Event?` – This function would let the AI pause and wait for a user event from the specified window. For example, the AI could call `captureEvent(window)` and the agent will block (with an optional timeout) until an event occurs (like user presses a key or clicks). The returned `Event` could be a structured object (e.g., `{ type: "keyDown", key: "A" }`). This is useful in interactive flows: the AI can explicitly ask for user input. However, this capability needs careful integration because the FountainAI planner might not natively "wait" within a function call. More likely, this would be implemented by the function-caller service returning a "pending" state that the planner interprets as needing to wait, or by a callback mechanism (see below). This function might be deferred in implementation until the simpler one-way calls are solid, but it's listed here as part of the agent's envisioned capabilities for completeness.

All these functions are defined in a manner that the AI (or actually the planning system on behalf of the AI) can invoke them with JSON-like arguments and get a result. In the FountainKit system, they will likely correspond to OpenAPI endpoints on the local `tool-server` or be built-in functions that the `function-caller` service knows how to dispatch. For example, an OpenAPI spec for the GUI Agent might define a `POST /gui/window/open` taking a JSON body with title and dimensions, and the tool-server's implementation of that endpoint calls `SDLKitGUIAgent.openWindow` internally.

Agent Implementation Details: The SDLKitGUIAgent is implemented within the SDLKit module (since it directly calls SDLKit API) but registers itself with FountainKit's tool registry. Internally, it will manage a mapping of `WindowID` to actual `SDLWindow`/`SDLRenderer` instances (to handle multiple windows if needed). It likely runs as a singleton service or object. When the agent is invoked to open a window, it creates the window via SDLKit and stores it in a dictionary, then returns the handle. For draw calls, it looks up the corresponding renderer and performs the draw operations followed by a present.

Concurrency considerations: Since the agent is effectively manipulating GUI state, it may need to ensure these calls run on the main thread (depending on SDL's requirements). The implementation might dispatch to the main thread for the actual SDL calls (especially on macOS). This is hidden from the AI caller, but documented for developers. The agent can use async handlers – e.g., the function-caller can call into the agent which uses `DispatchQueue.main.async` to execute the draw and then signals completion.

Event Callback Mechanism: In addition to the AI being able to pull events via a function (captureEvent), the agent could also **push events** back to the system. For example, if the user clicks "close" on the window's title bar (triggering an SDL_QUIT), the agent can proactively inform the FountainKit planner or some observer that the window was closed. This might be done by emitting a special message or using the telemetry stream. Perhaps the agent integrates with FountainTelemetryKit to send an SSE-like event (since that system already can handle streams of events). For instance, when an SDLEvent `.quit` is received, the agent could send an event "window_closed" to a channel that the gateway or planner monitors, prompting it to remove that step from the plan or to conclude the session. Similarly, key presses could be sent if the AI is meant to react immediately. The design of this is open – initial implementation may not include it, relying on synchronous pull (captureEvent), but as interactive use grows, a push model will be more efficient.

Integration in Plans: From the perspective of writing an AI objective or a plan, the GUI agent's functions will appear as tools the AI can choose. For example, a user asks: "Show me a preview of this script." The planner could produce a plan: 1. Call some content function to get the script or render data. 2. Call `GUI.openWindow` to create a window. 3. Call `GUI.drawText` or `GUI.drawImage` to put content. 4. Maybe loop waiting for user to close, then call `GUI.closeWindow`.

The agent's actions are thus first-class steps in the execution. The *cost* or *budget* of these actions might be taken into account by the planner (for instance, opening a window might be considered a heavy operation), but since it's local, it's quite fast in practice.

Security and Permissions: The agent runs locally and can open windows on the user's machine, which is powerful. We ensure that this agent is only enabled in appropriate contexts (likely on a desktop app or dev environment, not on a locked-down server). The user might need to grant permission for the AI to open UI windows, depending on the application's policy. We will have a configuration in FountainKit to enable/disable the GUI agent. By default, developer tools or the FountainAI desktop app would enable it, whereas a server distribution might keep it off.

Example (Pseudo) – A snippet of how the agent might be defined (for clarity, not actual code):

```
// Pseudo-code outline
class SDLKitGUIAgent: ToolAgent {
    private var windows: [Int: SDLWindow] = [:]
    private var renderers: [Int: SDLRenderer] = [:]
    private var nextID: Int = 1

    func openWindow(title: String, width: Int, height: Int) throws -> Int {
        let id = nextID; nextID += 1
        let window = SDLWindow(config: .init(title: title, width: width,
height: height))
        try window.open()
        let renderer = SDLRenderer(width: width, height: height, for: window)
        windows[id] = window
        renderers[id] = renderer
        // Optionally start a background loop or integrate with SDLRunLoop
for this window if needed
        return id
    }
    func closeWindow(windowId: Int) {
        guard let window = windows[windowId] else { return }
        window.close()
        windows.removeValue(forKey: windowId)
        renderers.removeValue(forKey: windowId)
    }

    func drawText(windowId: Int, text: String, x: Int, y: Int, font:
String?, size: Int?, color: UInt32) throws {
        guard let renderer = renderers[windowId] else { throw
AgentError.windowNotFound }
        // Use a default or specified font
        let fontName = font ?? "Arial"
        let fontSize = size ?? 16
        // Possibly cache loaded font objects
        // Render text to texture (if SDL_ttf available)
        if SDLKit.isTextRenderingEnabled {
            let fontObj = try SDLFont(name: fontName, size: fontSize)
            renderer.drawText(fontObj, text: text, x: x, y: y, color: color)
        } else {
            // Fallback: draw a placeholder rectangle or do nothing
        }
        renderer.present()
    }

    // Similar for drawRectangle, etc.
}
```

And an OpenAPI fragment (for conceptual illustration) might look like:

```
paths:
  /agent/gui/window/open:
    post:
      summary: Open a GUI window
      operationId: guiOpenWindow
      requestBody:
        content:
          application/json:
            schema:
              type: object
              properties:
                title: { type: string }
                width: { type: integer }
                height: { type: integer }
      responses:
        "200":
          description: Window opened
          content:
            application/json:
              schema:
                type: object
                properties:
                  window_id: { type: integer }
```

This would allow the function-caller to call `guiOpenWindow` with parameters, and the agent code executes, returning a `window_id`. Similar definitions would exist for drawText, etc. The exact schema might differ, but the idea is the agent's capabilities are formalized so the AI can invoke them.

Agent in Action: Once registered, the SDLKitGUIAgent runs as part of FountainKit's environment. For example, the `gateway-server` might incorporate it at startup if GUI is enabled, so that when a planning result includes a call to a GUI function, the gateway knows to dispatch it to this agent. The results (success or error) flow back to the planner which then continues the plan. From the user's perspective, when the AI decides to show something, a window pops up on the screen showing what the AI intended – a direct, interactive manifestation of the AI's output.

Closing the Window (User vs AI): If the user manually closes the window (e.g., clicks the close button), SDLKit will generate a quit event. The agent should detect this (perhaps via the background event thread or when the planner next calls any GUI function). The agent can then optionally auto-invoke the `closeWindow` logic and possibly inform the planner. In practice, we might have the agent proactively call `closeWindow` internally and also send a signal to the AI (maybe as an event or simply the next time the planner tries to use that window ID it will find it gone and handle that). This ensures resource cleanup. We'll document that user-initiated closure is always handled and the AI will be made aware (so it doesn't keep drawing to a closed window).

Conclusion of Agent Appendix: The SDLKitGUIAgent is the formal definition of how SDLKit's functionality is exposed in the AI ecosystem. By defining clear, limited functions, we maintain control and safety (the AI can't draw arbitrary things beyond these functions, which prevents misuse and simplifies what we test). This agent structure will evolve as we add capabilities – e.g., if we add image drawing, we'd add `drawImage()` to the agent API. The agent serves as **the contract between FountainKit's planning layer and the SDLKit module**. With it in place, FountainAI's textual brains gain a visual toolset, enabling richer interactions and feedback for users in the loop.

---

[1] [2] [3] [9] [10] [27] [28] AGENTS.md
https://github.com/Fountain-Coach/Teatro/blob/21d080f70b2c238469bbb0133a5f14b20afdd0ab/AGENTS.md

[4] [5] [23] [26] [32] [33] TeatroInteractiveGUI.md
https://github.com/Fountain-Coach/Teatro/blob/21d080f70b2c238469bbb0133a5f14b20afdd0ab/Docs/Chapters/TeatroInteractiveGUI.md

[6] [20] [21] [24] [25] [29] [30] [31] PreviewAPI.swift
https://github.com/Fountain-Coach/Teatro/blob/21d080f70b2c238469bbb0133a5f14b20afdd0ab/Sources/TeatroRenderAPI/PreviewAPI.swift

[7] SDLEvent.swift
https://github.com/Fountain-Coach/Teatro/blob/21d080f70b2c238469bbb0133a5f14b20afdd0ab/Packages/TeatroGUI/Sources/TeatroGUI/TeatroSDLBackend/SDLEvent.swift

[8] [11] [12] [13] [14] SDLWindow.swift
https://github.com/Fountain-Coach/Teatro/blob/21d080f70b2c238469bbb0133a5f14b20afdd0ab/Packages/TeatroGUI/Sources/TeatroGUI/TeatroSDLBackend/SDLWindow.swift

15 16 17 18 19 22 SDLRenderer.swift
https://github.com/Fountain-Coach/Teatro/blob/21d080f70b2c238469bbb0133a5f14b20afdd0ab/Packages/TeatroGUI/Sources/TeatroGUI/TeatroSDLBackend/SDLRenderer.swift

34 README.md
https://github.com/Fountain-Coach/Teatro/blob/21d080f70b2c238469bbb0133a5f14b20afdd0ab/Packages/TeatroGUI/README.md