ChatGPT

Implementation Strategy for Extending SDLKit with 3D Graphics, Multi-API Shaders, and GPU Compute

1. 3D Scene Graph Architecture

To support rich 3D content, SDLKit will gain a **scene graph** system to organize renderable objects, cameras, lights, and other entities. The scene graph will introduce Swift-friendly classes/structs such as:

- `Scene` – a container for all scene elements (including a root node and possibly management of active camera, global settings, etc.).
- `SceneNode` – the basic node in the graph, holding a transform (position, rotation, scale) and references to parent and child nodes. This enables hierarchical transforms (child positions relative to parents).
- `Camera` – a subclass or specialized `SceneNode` defining view and projection (perspective or orthographic parameters). Cameras produce view-projection matrices used for rendering.
- `Light` – a scene node representing light sources (directional, point, spot), storing properties like color, intensity, direction or attenuation, etc., used in lighting calculations.
- `Mesh` – a node that contains or references geometry (vertex/index buffers) and is associated with a material/shader. It may also hold a bounding volume for culling.
- `Material` – defines surface appearance by linking to shader programs and related GPU resources (textures, uniform parameters, blend state). A material effectively encapsulates a **render pipeline** configuration (the shader code for vertex/fragment stages plus fixed-function state).
- `Renderable` – protocol or base class for nodes that can be drawn (like meshes or possibly debug primitives), allowing the renderer to collect them.

Each `SceneNode` will maintain a local transform and compute a world transform (e.g., using 4x4 matrices or SIMD float4x4) by combining with ancestors. The scene graph can support **multiple render pipelines** by tagging nodes/materials with pipeline types (e.g., an opaque pass vs. shadow map pass). The architecture will allow attaching different pipeline state objects or shader variants to different materials, enabling advanced techniques like deferred shading or custom post-processing.

Render Pipeline Concept: In this context, a "render pipeline" refers to the GPU pipeline state configuration for drawing a mesh, including its compiled shader programs and fixed-function setup. Materials will be responsible for providing the appropriate pipeline state (or reference to a shader module) for a given backend API. The scene graph doesn't hardcode any API specifics; it just links a mesh to a material (which knows what shader to use). This decoupling means the scene graph remains high-level and platform-agnostic – it describes *what* to draw, while the rendering backend (discussed next) describes *how* to draw it on a specific API.

2. Multi-Platform `RenderBackend` Abstraction

We will introduce a `RenderBackend` **protocol** in Swift that defines the interface for rendering operations, abstracting over platform-specific graphics APIs. This protocol represents a low-level rendering engine responsible for creating GPU resources and executing draws. Key responsibilities of `RenderBackend` include: initializing the GPU device/context for a given window, creating buffers/textures, compiling or loading shader pipelines, issuing draw calls, and presenting the rendered frame.

Renderer Backend Implementations: For each target graphics API, we implement a concrete class conforming to `RenderBackend`:

- `MetalRenderBackend` – for macOS/iOS using Apple's Metal API.
- `D3DRenderBackend` – for Windows using Direct3D (likely DirectX 12 for modern feature parity).
- `VulkanRenderBackend` – for Linux and optional use on macOS (via MoltenVK) using Vulkan.

Each backend class encapsulates the details of that API. For example, `MetalRenderBackend` will manage a `MTLDevice`, command queue, render pass descriptors, etc., while `D3DRenderBackend` will handle D3D12 device and swap chain creation, and `VulkanRenderBackend` will manage VkInstance, VkDevice, VkSwapchain, etc. They all implement the same Swift protocol, so higher-level code (like the scene graph or window management) can remain ignorant of the specific API and just call the abstracted methods.

Choosing the Backend: At runtime or compile-time, SDLKit will select the appropriate `RenderBackend`. A simple strategy is to choose based on platform: e.g. use `MetalRenderBackend` on Apple platforms, `D3DRenderBackend` on Windows, and `VulkanRenderBackend` on Linux. We can allow an override (perhaps via an environment variable or config) to force a particular API, which is useful for testing Vulkan on macOS with MoltenVK or toggling between DirectX/Vulkan on Windows if needed. By default, however, it will auto-pick the primary API for the OS.

Backend Initialization: We will extend `SDLWindow` to create a rendering context for the chosen backend. For example, on macOS the window will be created with the SDL window flag for Metal (`SDL_WINDOW_METAL`) and we'll use Metal-specific setup; on Linux, use `SDL_WINDOW_VULKAN`; on Windows, no special flag (just use default and prepare for D3D). The `SDLWindow` class can gain a new initializer or property to specify the desired graphics API. It will then call into the appropriate backend to initialize. Each backend needs access to the native window handles and surfaces – for instance, on macOS it needs the `CAMetalLayer`, on Windows the HWND (window handle), on Vulkan the `VkSurfaceKHR` for the window. We will obtain these via SDL's API (possibly through our C shim as described next).

Once initialized, the `RenderBackend` will provide methods like `beginFrame()` / `endFrame()` (to set up command buffers or frames), `resize(width:height:)` (to handle window resizes and recreate swapchain or buffers), `createBuffer(...)`, `createTexture(...)`, `createShaderPipeline(...)`, and drawing operations (`drawMesh(mesh: Material:)` or a more explicit sequence of setting pipeline, binding resources, drawing geometry). The scene graph layer will ultimately invoke these methods for each visible object.

Importantly, the backend abstraction means the scene graph or higher-level code doesn't directly call Metal or Vulkan APIs – it goes through the backend interface. This keeps most of the code platform-independent and centralized in the backend implementations.

3. Low-Level Graphics Handles and C Shim Modifications

Achieving the above requires SDLKit's C shim (the `CSDL3` system library target) to expose some **low-level graphics handles** from SDL, because SDL by itself will not create a Metal device or Vulkan device – it only helps with windowing and integration. We need to get the platform-specific window handles or surfaces from SDL and hand them to the native graphics API:

- **macOS/iOS (Metal):** SDL3 provides functions to create a Metal view for a window and retrieve the `CAMetalLayer`. We will use `SDL_Metal_CreateView(window)` and `SDL_Metal_GetLayer(view)` to get a pointer to the CAMetalLayer associated with the SDL window [1]. The C shim can wrap this in a function like `SDLKit_MetalLayerForWindow(SDL_Window*) -> void*` so that Swift can call it and then bridge the void pointer to an actual `CAMetalLayer` object. With the CAMetalLayer in hand, the `MetalRenderBackend` can set up the Metal device and attach it: e.g. use `MTLCreateSystemDefaultDevice()` to get a MTLDevice and assign it to the layer's device, create a command queue, etc. The CAMetalLayer also gives us the drawable textures for presenting frames.

- **Windows (DirectX):** We need the native HWND from the SDL window. In SDL2 this was done via `SDL_GetWindowWMInfo`, but in SDL3 the new way is to use window properties. SDL3 exposes an SDL property for the Win32 window handle: `SDL_PROP_WINDOW_WIN32_HWND_POINTER` [2]. We can modify the C shim to retrieve this property ID and return the HWND as a pointer or integer. For example, add a function `SDLKit_Win32HWND(SDL_Window*) -> void*` that internally calls `SDL_GetWindowProperties` and fetches the `HWND`. Once we have the HWND, the `D3DRenderBackend` can create a Direct3D12 `ID3D12Device` and a DXGI swap chain targeting that window handle (using DX12 or DX11 APIs via C++ interop). We may write a small C++ source as part of the C shim to assist in creating D3D devices or at least to translate between Swift and COM calls (since Swift can't directly call COM interfaces easily). Initially, however, just exposing the HWND is enough; device creation will be done inside the backend (perhaps by calling out to a helper C function for complex COM setup).

- **Linux (and macOS with Vulkan/MoltenVK):** SDL provides utilities for Vulkan. We will use `SDL_Vulkan_GetInstanceExtensions` to know which instance extensions are required for the window surface, then create a `VkInstance`. After that, we use `SDL_Vulkan_CreateSurface(window, instance, ...)` to get a `VkSurfaceKHR` bound to the SDL window [3]. The C shim can expose a helper `SDLKit_CreateVulkanSurface(SDL_Window*, VkInstance) -> VkSurfaceKHR` for convenience (wrapping the SDL call). The `VulkanRenderBackend` will then proceed to pick a GPU, create a VkDevice with the appropriate queue that supports presenting to that surface, and set up a swapchain. (On macOS using MoltenVK, the same path is used – MoltenVK translates the VkSurface into an underlying CAMetalLayer internally, but our code treats it like Vulkan.)

In all cases, **SDL remains responsible for window creation and event handling**, but we bypass SDL's rendering routines. The C shim additions ensure we can get the platform-specific handles SDL has prepared. We also ensure that the SDL main initialization (video subsystem) is done and that these calls happen on the main thread (as required by SDL and some platform APIs). By exposing just the needed handles (CAMetalLayer, HWND, VkSurface, etc.), the Swift code in `RenderBackend` can interface with Metal, DirectX, or Vulkan directly.

We will guard these shim functions with appropriate `#if` checks so they are only compiled on their relevant platforms (e.g., Metal functions only on Apple, Windows functions only on Windows, etc.). The `SDLKit` high-level code will call them conditionally based on which backend is being initialized.

4. Shader Abstraction and Cross-Compilation Strategy

A critical component is a **compile-time shader abstraction layer** that allows us to maintain one set of shader logic and target multiple backends (Metal, D3D, Vulkan) in their native formats. Each graphics API expects shaders in a specific language or binary format: Metal uses **MSL (Metal Shading Language)** source or precompiled metallib, DirectX uses **HLSL** (High-Level Shading Language) which compiles to DXIL or DXBC bytecode, and Vulkan consumes shaders as **SPIR-V** binary modules [4] [5]. Our goal is to avoid writing completely separate shaders for each API by using transpilation or a common source.

Shader Source Strategy: We can choose a single "source" language for authoring shaders and then compile or translate it to each platform's required format at build time. A practical choice is to use HLSL as the source language, because modern toolchains allow HLSL to be cross-compiled: Microsoft's open source DXC compiler can compile HLSL to SPIR-V (for Vulkan) and to DXIL (for DirectX12), and then we can use SPIRV-Cross (or similar) to translate that SPIR-V to MSL for Metal [6]. This approach is supported by recent SDL developments as well (SDL's upcoming GPU API expects SPIR-V, DXIL, MSL inputs and a tool called SDL_shadercross handles HLSL→SPIR-V/MSL conversion) [7] [6]. For example:

- Author shaders in HLSL (or a compatible cross-language subset of HLSL/GLSL).
- At build time, use **DirectX Shader Compiler (dxc)** to compile HLSL source to SPIR-V for Vulkan and to DXIL for DirectX.
- Then use **SPIRV-Cross** to convert the SPIR-V into MSL source, which can be compiled with Apple's metal tools into a `.metallib` library (or possibly use dxc's ability to target MSL directly or an offline metallib if available).
- Alternatively, we could author in GLSL or a high-level DSL, but HLSL->SPIR-V->MSL is a proven path.

All of this can be integrated into the build system (e.g., Swift Package Manager can run custom shell scripts or we maintain a separate build step). The shaders could reside in a `Shaders/` directory with the high-level source. For example, if we have `basic_lit.hlsl` (containing vertex and fragment shader entry points), the build process will produce `basic_lit.metallib` (Metal binary library) for Mac, `basic_lit.dxil` for Windows, and `basic_lit.spv` for Vulkan. We'll include or bundle these compiled shaders with the app (or pack them as data blobs). The `RenderBackend` on each platform will load the appropriate shader binaries at runtime: - Metal can load the `.metallib` via `MTLDevice.makeLibrary(URL)` or include it in the app bundle. - DirectX can take the DXIL (or we compile HLSL to DX bytecode if targeting DX11) and create a ID3D12PipelineState with it. - Vulkan will take the SPIR-V binary and create VkShaderModule objects.

For simplicity, initial implementation might manually maintain parallel shader code (one per API) if using cross-compilation tools is too complex. However, the recommended approach is to **automate shader translation at compile-time** so developers write one shader source per effect. This not only reduces maintenance, it ensures feature parity across platforms. If needed, we can provide a small Swift wrapper to select the correct shader variant or an enum of shader IDs that abstracts away the file names.

We will also design a **Shader module system** in code: e.g., a Swift `Shader` class that represents a compiled shader (or a pipeline state) with properties like available vertex attributes, uniform layouts, etc. The `Material` class in the scene graph could reference such a `Shader` object. Under the hood, the `Shader` will carry backend-specific compiled handles (like an `MTLRenderPipelineState` for Metal, a pair of `ID3D12ShaderBytecode` or pipeline state for D3D, or `VkPipeline`/`VkPipelineLayout` for Vulkan). Creation of these will go through the `RenderBackend` – e.g., a `createShaderPipeline(vertexSource:fragmentSource:)` that compiles/links the pipeline appropriately.

By organizing shaders by name and stage and compiling them offline, we avoid runtime shader compilation cost. The compile-time abstraction ensures that **each platform's build includes only its relevant shader binaries**, and we use Swift's conditional compilation to exclude others. (For instance, `.metal` shader files and Metal framework calls are only compiled on `#if os(macOS) || os(iOS)`). The result is a unified shading layer: developers can write a shader once and SDLKit's build process will produce the right binaries for Metal, DirectX, or Vulkan as needed.

5. GPU Compute Module Design

Beyond graphics, SDLKit will include a **GPU compute layer** enabling general-purpose computations (GPGPU) for tasks like audio DSP, physics simulation, and machine learning inference. The design will leverage the same multi-backend approach, since Metal, DirectX, and Vulkan all support compute shaders/kernels.

We will add a concept of a **ComputePipeline** (or reuse the Shader abstraction with a compute flag) and a **ComputeCommand** interface. Key components:

- **Compute shaders**: Similar to graphics shaders, but with no vertex/fragment stages – just a compute function (for example, an HLSL compute shader that we also cross-compile to SPIR-V and MSL). We'll support writing these in the same shader source files or separate ones. e.g., an `fft_compute.hlsl` that is compiled to `fft_compute.metal` and `fft_compute.spv` etc. The build pipeline we set up for graphics can be extended to cover compute shaders (DXC can compile HLSL compute to SPIR-V and DXIL as well).
- **ComputePipeline/PipelineState**: Each backend will need to create a compute pipeline state object (e.g., `MTLComputePipelineState` on Metal, `ID3D12PipelineState` with a compute shader on D3D12, `VkPipeline` with compute shader on Vulkan).
- **Dispatch interface**: The `RenderBackend` protocol might get new methods such as `createComputePipeline(shader: Shader)`, `dispatchCompute(pipeline: ComputePipeline, threads: [Int])` etc. Alternatively, we could factor out a `GPUDevice` abstraction that has both `graphicsCommandQueue` and `computeCommandQueue`. For simplicity, we can let `RenderBackend` manage compute as well (since on many APIs the graphics device is also used for compute). We may add a `ComputeBackend` protocol extension if we want to logically separate it, but likely it's easier to integrate into one `RenderBackend` that does both.
- **Resource sharing**: The compute tasks will often use data that interacts with graphics or with the CPU. We will design buffer management such that the same buffer objects can be used by both rendering and compute. For example, a physics simulation compute shader might update positions in a vertex buffer that is then used to render objects. On Metal and Vulkan, this is straightforward as long as we synchronize access (for Vulkan, using the proper pipeline barriers; for Metal, ensure command ordering). On DirectX12, using the same ID3D12Resource in compute and graphics is also standard. We will provide synchronization or scheduling tools as

needed (probably out of scope for initial design, but note that some compute tasks might run asynchronously).
- **Use cases**: We will verify that typical use cases are supported. For audio DSP, a compute shader could perform parallel signal processing (e.g. an FFT); for physics, a compute pipeline could update particle systems or rigid body physics broadphase; for ML, we could run inference by implementing neural net layers as shader code or integrate with existing libraries. Our design allows these by letting the user supply appropriate shader code and dispatch sizes.

From an API perspective, the high-level might expose something like a `GPUComputeTask` where the user can supply a compute shader and input/output buffers, then request it to run. Under the hood, that will call the backend's compute dispatch. We will ensure that results can be read back to the CPU if needed (e.g., map a buffer after the GPU finishes, or use a staging buffer).

By integrating compute into SDLKit, advanced workflows become possible entirely on the GPU – audio effects can be processed with Metal or Vulkan compute for low latency, physics can be GPU-accelerated, and ML inference can utilize frameworks like Apple's MPS or just raw shaders. Initially, we will implement the basics (shaders and dispatch), and consider adding conveniences or interoperability with platform ML libraries later.

6. Conditional Compilation and Code Organization

Implementing multi-API support will require careful use of **conditional compilation** to keep code paths separate and only include relevant code on each platform. We will use Swift's `#if` directives (and possibly separate modules) to organize platform-specific code:

- The `SDLKit` Swift Package can be structured so that common API and abstractions are defined once, but implementation details are split by OS. For instance, we can have a single `RenderBackend` protocol definition (no `#if` in the protocol itself), and then in files like `MetalRenderBackend.swift` we wrap the entire class in `#if os(macOS) || os(iOS)` so it only compiles on Apple platforms (where we can `import Metal`). Similarly, `D3DRenderBackend.swift` enclosed in `#if os(Windows)` (importing D3D12 or using our C++ helpers), and `VulkanRenderBackend.swift` in `#if os(Linux) || (os(macOS) && USE_MOLTENVK)` if we allow Vulkan on macOS under a feature flag.
- We will leverage `canImport` as well, e.g. `#if canImport(Metal)` around code that uses Metal framework types, and `#if canImport(Vulkan)` (if a Vulkan C module is available). Given Swift's package manager, we might also set up separate targets like an `SDLKitMetal` target that links Metal framework, but that complicates usage. Instead, in **Package.swift** we can specify conditional linker settings: e.g., link Metal on macOS, link D3D libraries on Windows, etc., so that the symbols resolve when needed.
- The C shim will also use conditionals: in `shim.h` (and any implementation .c/.mm files) we will include `<SDL3/SDL_vulkan.h>` or `<SDL3/SDL_video.h>` depending on platform to get the needed functions, and include `<Metal/Metal.h>` for bridging if necessary on Apple (or we might just treat CAMetalLayer as an opaque pointer in C).
- We will isolate any Objective-C bridging (for interacting with CAMetalLayer or Cocoa) in a `.mm` Obj-C++ file if needed, and similarly any C++ for DirectX. Those files will only compile on their respective platforms (controlled by Xcode build settings or Package.swift cSettings).
- **Project structure:** We propose to organize the code into clear subdirectories:
- `Sources/SDLKit/Core/` – core cross-platform classes like `SDLWindow` and basic event loop or SDL integration.

- `Sources/SDLKit/Graphics/` – contains `RenderBackend.swift` (protocol and maybe some common types), and sub-files or sections for each backend:
  - `MetalBackend.swift`, `VulkanBackend.swift`, `DirectXBackend.swift` (each with platform flags).
  - Possibly `Shader.swift` (a common abstraction for shader programs/pipelines).
  - `Compute.swift` or `ComputePipeline.swift` for compute-specific abstractions (or these could be merged with Shader).
- `Sources/SDLKit/Scene/` – classes for scene graph: `Scene.swift`, `SceneNode.swift`, `Camera.swift`, `Light.swift`, `Mesh.swift`, `Material.swift`. These are mostly platform-independent (they call into RenderBackend but don't know the details).
- `Sources/SDLKit/Support/` – any support code, utilities, math helpers (we might use SIMD from Swift `simd` library for matrices and vectors).
- `Sources/SDLKit/Agent/` – existing GUI agent remains here; if we create new agent interfaces for controlling 3D scenes via JSON, they could be added (though initially, these new "agents" might be for internal architecture rather than user-facing JSON APIs).
- Shaders might live outside Sources (since they're not Swift code). We could put them in a top-level `Shaders/` directory or inside `Sources/SDLKit/Shaders/` if we treat them as resources. SwiftPM can include them as resources if needed, or we handle them via build scripts.

This layered organization ensures a clear separation: **Core (SDL + windowing)**, **Scene Graph (engine-level constructs)**, **Graphics Backends (platform-specific rendering)**, **Shaders/Compute (cross-platform assets and their compilation)**, and **Agents (integration with AI or external control)**. The architectural layering is such that SceneGraph -> RenderBackend -> SDL/window/OS, i.e., high-level code calls into the abstract backend, which in turn uses the low-level OS handles from SDL.

Using conditional compilation means each build of SDLKit will only contain the code and libraries it needs for that platform (e.g. the Linux build won't include Metal or D3D code at all, and Windows won't include Vulkan unless we explicitly enable it, etc.), keeping binaries lean and avoiding undefined symbols on platforms where certain frameworks aren't present.

7. Integration with `SDLWindow`, `SDLRenderer`, and Event Loop

The existing SDLKit provides `SDLWindow` and `SDLRenderer` classes (with SDLRenderer likely wrapping SDL's 2D rendering). To integrate the new 3D capabilities smoothly:

- **SDLWindow Enhancements:** We will extend `SDLWindow` initialization to accept parameters for graphics context. For example, a new initializer `SDLWindow(title: String, width: Int, height: Int, graphics: GraphicsAPI = .auto)` where `GraphicsAPI` is an enum (`.none` for no GPU, `.metal`, `.direct3D`, `.vulkan`, or `.auto`). If `.auto`, we select based on platform as discussed. Internally, `SDLWindow` will call `SDL_CreateWindow` with appropriate flags (Metal or Vulkan flags) and store the created `SDL_Window*`. After creation, if a GraphicsAPI was requested, `SDLWindow` will create the corresponding `RenderBackend` instance. It might look like:

```
switch graphics {
  case .metal: backend = MetalRenderBackend(window: self)
  case .direct3D: backend = D3DRenderBackend(window: self)
  case .vulkan: backend = VulkanRenderBackend(window: self)
  case .none: backend = nil
```

```
    case .auto: // choose default for OS
}
```

The backend will likely need the `SDL_Window*` pointer (from the CSDL3 shim) to retrieve native handles as described. We ensure this initialization occurs on the main thread (SDL requirement and also UIKit/AppKit requirement for creating views).

- **SDLRenderer vs New Renderer:** SDL's own rendering system (SDL_Renderer) is geared toward 2D. For advanced 3D, we will bypass `SDLRenderer` usage. We may keep the `SDLRenderer` class in SDLKit for backward compatibility or for simple usage (it might currently wrap `SDL_CreateRenderer` which could be using OpenGL/Direct3D behind the scenes). But once a `RenderBackend` is in use for a window, we won't use `SDL_Renderer` for that window. In fact, SDL requires choosing one approach: either use SDL's rendering or an API like Vulkan/Metal on a window, but not both simultaneously.

One approach is to **augment** `SDLRenderer` **class** to handle both cases: - If user requests a "simple" renderer (perhaps via a flag or if they don't initialize a 3D backend), `SDLRenderer` can create and wrap an SDL_Renderer as it does now (for compatibility and basic 2D drawing). - If a `RenderBackend` is active, `SDLRenderer` could either be absent or act as a high-level facade that knows how to call the new backend's drawing for simple shapes. However, that might complicate things. It might be cleaner to introduce a new class, say `GraphicsRenderer` or just let the user interact with the `RenderBackend`/scene classes directly for 3D content.

We will likely **not use SDL_Renderer** when the goal is advanced rendering. Instead, the `RenderBackend` and our scene graph become the way to render. The event loop will remain the same (polling SDL events), and the user (or agent controlling SDLKit) will drive the rendering by calling something like `renderFrame()` on the scene or backend each loop iteration, after updating the scene graph.

- **Event Loop Integration:** SDL's event handling (keyboard, mouse, window events) remains essential. SDLKit will continue to provide events via SDL (e.g., `SDL_PollEvent`). The presence of a 3D scene doesn't change that; however, we will handle a few new events:
- Window resize: when the SDL `SDL_EVENT_WINDOW_RESIZED` (or SDL3 equivalent) occurs, we must inform the `RenderBackend` to resize swap chains or buffers. We can have SDLWindow detect this and call `backend.resize(newWidth, newHeight)`.
- Perhaps expose new events for graphics (if needed, e.g., device lost events for D3D12, or handling going to fullscreen, etc., but SDL already covers those to some degree).
- Input events like mouse or touch can be used to control the camera or objects – that is up to the user, but our system should allow e.g. feeding mouse motion to the scene (for camera rotation etc.). We might later integrate with the agent system to allow programmatic camera control.

The **overall integration plan** is to make the addition of 3D as seamless as possible: if a developer doesn't need 3D, they can continue to use SDLKit as before with the simple 2D renderer. If they do need it, they opt-in by creating a window with a 3D backend, then use the new scene graph and backend API. SDLKit will manage the low-level details, so the developer (or AI agent) can focus on populating the scene and writing shaders.

We will also ensure that shutting down or closing a window properly releases the graphics resources. SDLWindow's deinit or an explicit close method will destroy the `RenderBackend` (which in turn

releases the GPU device, swapchain, etc.) and then destroy the SDL window. This prevents leaks across window recreations.

8. SDL 2D Renderer vs Advanced Rendering Trade-offs

SDL's built-in 2D renderer is convenient for simple drawings and is cross-platform, but it is limited in capability (only 2D operations) and abstracts away the details of GPU state. For advanced rendering (3D, or even complex 2D with custom shaders), we need to bypass it. Here are the considerations:

- **Performance & Flexibility:** By bypassing SDL_Renderer and using Metal/Vulkan/DirectX directly, we gain full control over GPU features (shaders, multiple render targets, depth buffering, etc.). This is essential for 3D. SDL_Renderer cannot do 3D geometry or user-defined shaders, so it's not viable for our goals beyond a certain point. Moreover, our multi-API design can tap into modern graphics API performance benefits (e.g., parallel command recording in Vulkan/D3D12, explicit GPU memory control), which SDL_Renderer (often implemented on older APIs or simplified abstractions) might not leverage fully.
- **Complexity:** The downside is more code complexity on our side. We'll be re-implementing many things that SDL's renderer would have handled (like creating a context, basic draw calls). However, SDL3 is evolving with a GPU API of its own; in the future we might align with it, but for now we proceed with our custom abstraction to meet our specific needs.
- **Coexistence:** We should decide if it's possible or necessary to allow both rendering paths in one application. A scenario might be an app that mostly uses 3D but wants to quickly draw some 2D UI with SDL_Renderer. Mixing is tricky: for example, on Windows if SDL_Renderer is created with D3D and we also use D3D12 ourselves, there could be conflicts; on Metal, SDL_Renderer could create its own CAMetalLayer or Metal device separate from ours, which is inefficient or impossible to coordinate. The *simplest policy* is: **if using the advanced 3D renderer, do not use SDL_Renderer on the same window**. For 2D overlays in a 3D scene, one can either:
- Draw UI using our system (e.g., as textured quads in the 3D world or in screen space via an orthographic camera).
- Or maintain a separate SDL window purely for GUI if needed (not ideal).

We will likely deprecate or discourage `SDLRenderer` usage in combination with the new scene graph on the same window. The new architecture can encompass 2D drawing anyway (since one can write a simple 2D shader and treat sprites as textured rectangles in the scene). - **SDL_ttf and text rendering:** One feature SDL_Renderer via SDLKit currently has (according to the agent spec) is text drawing using SDL_ttf. If we remove SDL_Renderer, we lose easy text drawing. To address this, we have options: integrate SDL_ttf to render text to an SDL surface or raw bitmap and then upload that as a texture in our system, or use signed-distance field fonts or native platform text for better quality. This could be an area of future enhancement. Initially, we might keep a minimal SDL_Renderer alive in the background only for text if absolutely necessary (render text to an SDL surface then convert to texture for Metal/Vulkan). However, that's an edge case – the primary direction is to migrate fully off SDL_Renderer for advanced use. The **agent-based approach** (where AI tools call `drawText`) could be reworked to use a similar mechanism (render text to texture, etc.) behind the scenes when the advanced pipeline is in use.

In summary, keeping SDL's 2D renderer is useful for backward compatibility and quick-and-dirty GUI, but for "real" 3D rendering we will bypass it in favor of our more powerful pipeline. We will document this clearly for users. SDLKit can continue to support both modes, but they will be separate code paths.

9. Proposed File/Module Layout

Below is a proposed project structure incorporating the new components (new files **in italics**):

```
SDLKit/
├─ Package.swift
├─ Sources/
│   ├─ CSDL3/                  # C shim for SDL3
│   │   ├─ module.modulemap
│   │   └─ shim.h (and platform C/C++ source files if needed)
│   └─ SDLKit/
│       ├─ SDLKit.swift        # Top-level config/state as before
│       ├─ Agent/
│       │   ├─ SDLKitGUIAgent.swift    # Existing 2D GUI agent
│       │   └─ **(Future: perhaps new agent interfaces for 3D?)**
│       ├─ Core/
│       │   ├─ SDLWindow.swift   # Extended for 3D context support
│       │   └─ SDLRenderer.swift # May remain for legacy 2D
│       ├─ Graphics/
│       │   ├─ **RenderBackend.swift**    # Protocol & common structs (e.g.
color, vertex formats)
│       │   ├─ **MetalBackend.swift**     # Metal-specific implementation
(`#if os(macOS) || os(iOS)`)
│       │   ├─ **DirectXBackend.swift**   # D3D-specific implementation (`#if
os(Windows)`)
│       │   ├─ **VulkanBackend.swift**    # Vulkan-specific implementation
(`#if os(Linux) || …`)
│       │   ├─ **Shader.swift**          # Shader abstraction (could include
compile pipeline logic or references to compiled blobs)
│       │   └─ **Compute.swift**         # Compute pipeline abstraction (or
merged with Shader)
│       ├─ Scene/
│       │   ├─ **Scene.swift**         # Scene container (could manage root
node, cameras)
│       │   ├─ **SceneNode.swift**     # Base class for scene graph nodes
│       │   ├─ **Camera.swift**        # Camera node
│       │   ├─ **Light.swift**         # Light node
│       │   ├─ **Mesh.swift**          # Mesh node (geometry & material
reference)
│       │   └─ **Material.swift**      # Material (ties to shader pipeline,
textures, etc.)
│       ├─ Support/
│       │   ├─ Errors.swift           # Error enums (extend with Graphics/
Shader errors)
│       │   └─ Utilities.swift (optional helper functions, e.g., matrix math
if needed)
│       └─ Shaders/ (if included as resources or source for shadercross)
│           ├─ **basic.vert.hlsl** (example shader source)
│           ├─ **basic.frag.hlsl**
```

```
│           └─ **...**
└─ Tests/
    ├─ SDLKitTests.swift        # Extended to test new functionality (if
feasible, though graphics tests might need integration tests)
    └─ ...
```

In this layout, the **Graphics** directory encapsulates all the low-level rendering logic per API, **Scene** encapsulates the high-level 3D scene structure, and **Shaders** (optionally) holds shader source files. The CSDL3 shim might also have sub-files like `shim+metal.mm`, `shim+d3d.cpp` for platform-specific parts.

This modular structure will help assign development tasks – for example, one engineer/agent can work on the scene graph in isolation from the Vulkan backend, etc. It also allows future expansion (e.g., adding OpenGL as another backend if desired, or adding more utility modules like a physics integrator or integration with audio).

Finally, we maintain the separation of concerns: SDLKit's public API remains Swift-centric and hides the C interop and platform details. We continue to not leak raw pointers in the public API [8] – for instance, the user will never see a `CAMetalLayer` or `VkInstance` directly from SDLKit, they just get a high-level `SDLWindow` or `RenderBackend` object. This adheres to the existing design philosophy of SDLKit, extended now for 3D and GPU compute.

10. Draft AGENTS.md: New Agent Roles for 3D Graphics Extension

(The following is a draft `AGENTS.md` section outlining the roles of new specialized coding agents within the SDLKit project. These agents are conceptual divisions used to plan and implement the 3D extension, assisting an autonomous coding system in dividing responsibilities.)

Agent Roles Overview

With the introduction of 3D graphics and multi-API support in SDLKit, we define several collaborative agent roles. Each agent focuses on a specific aspect of the implementation, with clear inputs/outputs and communication channels between them:

- **GraphicsAgent** – Oversees core rendering integration and platform backends.
- **ShaderAgent** – Manages shader language translation and pipeline creation.
- **ComputeAgent** – Handles GPU compute functionality across platforms.
- **SceneGraphAgent** – Builds the 3D scene graph structure and logic.

These agents work together under the guidance of the project planner to autonomously implement and maintain the new features.

GraphicsAgent

- **Responsibilities:** The GraphicsAgent is in charge of the overall rendering infrastructure. This includes modifying the SDLKit core to initialize graphics contexts, implementing the `RenderBackend` protocol, and writing the platform-specific backend classes (Metal, Vulkan, DirectX). GraphicsAgent ensures that low-level handles from SDL (window, surfaces) are correctly obtained and passed to the graphics API. It also manages frame lifecycle (initialize devices,

handle resize, present frames) and enforces that rendering calls occur on the correct thread and order.

- **Inputs:** It receives the required design of the rendering abstraction (the protocol definition and expected functions) and platform details such as SDL window handles (from SDL through the C shim). It also takes in **shader binaries or pipeline definitions** from the ShaderAgent and the rendered scene data (a list of draw commands or scene graph from SceneGraphAgent). For example, ShaderAgent provides compiled shader objects that GraphicsAgent must load into pipeline states.
- **Outputs:** It produces the implemented `RenderBackend` classes in Swift and any necessary C/C++ support code for initialization (e.g., DirectX device creation routines). The GraphicsAgent's code outputs include a working rendering loop that other parts of SDLKit can call (e.g., a function to render a Scene each frame). It also outputs callbacks or interfaces that SceneGraphAgent can use to submit draw calls. In terms of runtime, GraphicsAgent yields the final rendered image presented to the window.
- **Intercommunication:**
- *With ShaderAgent:* GraphicsAgent relies on ShaderAgent to provide shader programs in the correct format. There will be a defined interface whereby GraphicsAgent asks ShaderAgent for a shader pipeline given a material or effect name. They must agree on identifiers for shaders and on when compilation is done (e.g., ShaderAgent might run an offline compilation step that GraphicsAgent then uses).
- *With ComputeAgent:* GraphicsAgent and ComputeAgent coordinate on shared GPU context – there may be a single GPU device used for both rendering and compute. GraphicsAgent exposes an API for ComputeAgent to get access to the device/command-queue or to schedule compute work. They must also synchronize if compute jobs need to run before or after rendering.
- *With SceneGraphAgent:* GraphicsAgent provides SceneGraphAgent with an interface to submit drawable objects. For example, SceneGraphAgent, after culling or preparing the scene, will call something like `renderBackend.draw(mesh, with: material, transform: matrix)` for each object. GraphicsAgent (via the backend) executes those calls. Conversely, GraphicsAgent might communicate back any rendering-specific events (like device lost, or if certain resources are low) so SceneGraphAgent can adjust (though such cases are rare).
- The GraphicsAgent essentially acts as the **executor** that takes data from SceneGraph and Shader agents and uses it to produce frames.

ShaderAgent

- **Responsibilities:** The ShaderAgent handles everything related to shader code and GPU pipeline states. This includes organizing shader source files, implementing the compile-time translation to multiple shading languages (MSL, HLSL, SPIR-V), and providing a Swift interface to create and use shaders in the engine. ShaderAgent ensures that each graphics backend has the shader binaries it needs. It may also define a system for associating shader code with materials or pipeline descriptors.
- **Inputs:** It takes high-level shader specifications or source files (e.g., HLSL code provided by developers or by the project requirements). It also takes **target platform info** from GraphicsAgent – for instance, knowing which backends are in play to produce the appropriate outputs (if we skip compiling some format not needed). It may also get requests from SceneGraphAgent for certain standard shader features (like "create a Phong shader" or "a texture shader") which it will fulfill by either selecting an existing shader or compiling one.
- **Outputs:** ShaderAgent outputs compiled shader artifacts: e.g., Metal library files, SPIR-V binaries, HLSL bytecode. In the codebase, it produces the `Shader` Swift structures/classes that encapsulate these artifacts for use at runtime. It also could output helper functions for binding shader uniforms or validating shader inputs. Essentially, it yields a **shader module library** for

the engine. Additionally, ShaderAgent will update build scripts or Package.swift settings to perform shader compilation steps as needed (this might be in collaboration with build system, but conceptually it's part of its output to configure that).

- **Intercommunication:**
- *With GraphicsAgent:* ShaderAgent provides GraphicsAgent with ready-to-use shader identifiers or objects. For example, after compilation, it might register shader programs with GraphicsAgent or the RenderBackend. GraphicsAgent might call an API like `ShaderAgent.getPipeline(for: materialX, backend: .metal)` to retrieve a compiled MTLRenderPipelineState or similar object.
- *With SceneGraphAgent:* ShaderAgent works indirectly with SceneGraphAgent by way of materials. SceneGraphAgent defines materials and what they need (e.g., a material might indicate it needs a shader with certain features or it attaches a shader by name). ShaderAgent ensures those shader programs exist and meet the requirements. They might co-design a Material descriptor format that includes shader references which ShaderAgent resolves.
- *With ComputeAgent:* If there are compute shaders, ShaderAgent similarly compiles those. ComputeAgent will request from ShaderAgent the compiled compute pipeline objects for a given compute task (for instance, "FFTComputeShader" compiled to each backend). So ShaderAgent's domain covers both graphics and compute shaders, unifying them.
- Internally, ShaderAgent may use external tools (DXC, SPIRV-Cross) – while not other "agents," this means it might orchestrate command-line calls or use libraries. It should handle errors in shader code and present them meaningfully (for a human developer or for logging).

ComputeAgent

- **Responsibilities:** The ComputeAgent focuses on the GPU Compute layer. It implements the API for scheduling and running compute workloads on the GPU. This involves creating compute pipeline states from shader code (with ShaderAgent's help), managing command dispatch (thread groups, etc.), and coordinating memory transfers for input/output of compute tasks. It also needs to handle multi-platform differences in how compute is launched (e.g., Metal's `dispatchThreads` vs Vulkan's `vkCmdDispatch` details, though the GraphicsAgent's backend abstraction will cover the API specifics).
- **Inputs:** ComputeAgent receives high-level descriptions of compute tasks. For instance, a request like "perform convolution on audio buffer with GPU" or "run physics update kernel on these data buffers" – likely coming from the user code or higher-level logic. More concretely, it takes:
- References to **compute shader programs** (from ShaderAgent, since those must be compiled already for each backend).
- References to data buffers (which may be created via GraphicsAgent's resource APIs).
- Execution parameters (like number of threadgroups, or the dimensions of work).
- **Outputs:** It produces execution of the compute tasks and any results in memory accessible to the requester. In the codebase, ComputeAgent will output new classes or methods such as `GPUComputeTask` or `dispatchCompute` functions. It may also produce utility code for common patterns (like reducing results, copying data back to CPU). Another output is synchronization primitives or ensuring that the compute tasks complete (maybe providing a callback or future/promise when done, or blocking until completion if called synchronously).
- **Intercommunication:**
- *With GraphicsAgent:* ComputeAgent works closely with GraphicsAgent because they share the GPU context. Likely, the actual command buffer submission for compute will be done through the `RenderBackend` (GraphicsAgent's domain). For example, ComputeAgent might call `renderBackend.encodeCompute(command: someComputeCommand)`. This means ComputeAgent might actually be implemented partly within each backend or call into backend-specific hooks. The two agents must agree on resource states: if a buffer is used by compute

then by graphics, GraphicsAgent (or the backend) must insert the correct synchronization (barriers or wait for completion). They communicate about who has ownership of a resource at a given time.

- *With ShaderAgent:* ComputeAgent requests the compute shader pipelines from ShaderAgent. This is similar to GraphicsAgent's interaction but specifically for compute stages. They must ensure that compute shaders are compiled and available. For example, if ComputeAgent wants a "physics.step" shader, it asks ShaderAgent for the compiled pipeline for each platform.
- *With SceneGraphAgent:* There might be less direct interaction, but consider physics: SceneGraphAgent could delegate physics updates to ComputeAgent. For instance, SceneGraphAgent could maintain a physics simulation and call ComputeAgent to advance it on GPU, then get back results (like updated transforms) and apply them to SceneNode positions. In such a case, SceneGraphAgent defines what data goes into the compute shader (positions, velocities) and after ComputeAgent runs the simulation step, SceneGraphAgent reads the output and updates the scene. So they communicate via data (buffers) rather than direct calls. Another example is using ComputeAgent for GPU skinning of animated models – SceneGraphAgent might prepare a compute task for skinning vertices and then use the result for rendering.
- In summary, ComputeAgent ensures that non-graphics computations can be offloaded to GPU and works with the others to make the GPU a shared resource for both rendering and computation tasks.

SceneGraphAgent

- **Responsibilities:** The SceneGraphAgent handles the high-level scene management. It implements the new classes for scene graph nodes, cameras, lights, meshes, and materials. It is responsible for update logic (e.g., updating world transforms each frame, culling objects outside the camera view, perhaps managing scene-level animations). It also integrates user input or higher-level commands to manipulate the scene (like moving a camera with keyboard input, or adding/removing objects).
- **Inputs:** It takes in design specifications for the scene graph (the class hierarchy and relationships defined in the strategy). It also listens to **user or AI commands** that want to create or modify the scene (for instance, an AI planner might instruct "add a light at position X" or "load model Y as a mesh"). It uses the public SDLKit API (or internal knowledge) to construct the scene accordingly. It also receives frame updates (e.g., a tick signal each iteration of the event loop, possibly with a delta time) to animate or move objects.
- **Outputs:** SceneGraphAgent produces the Swift implementation of the scene graph classes (`SceneNode`, `Camera`, etc.) and ensures they are well-integrated. At runtime, it outputs a list of renderable commands each frame: effectively, after culling and processing, it will output to the GraphicsAgent what needs to be drawn. This could be done by calling drawing methods on the `RenderBackend` for each visible object. Additionally, SceneGraphAgent might output events or state information (like "window resized – update projection matrices" or "camera switched"). In code, its outputs are also user-facing APIs – e.g., functions to add a child node, set a material's properties, etc., which will be part of SDLKit's public API for 3D.
- **Intercommunication:**
- *With GraphicsAgent:* SceneGraphAgent sends drawing requests to the GraphicsAgent (or rather, uses the RenderBackend provided by GraphicsAgent). For example, in the main loop SceneGraphAgent might do: `for object in scene.visibleObjects { renderBackend.draw(object.mesh, with: object.material, transform: object.worldTransform) }` wrapped between `renderBackend.beginFrame()` and `endFrame()`. They also communicate for resource creation: when a new Mesh is created, SceneGraphAgent might call something like `renderBackend.createMesh(vertexData)` which invokes the backend to create GPU buffers. SceneGraphAgent doesn't manage GPU

memory itself; it asks GraphicsAgent to do it and stores references (handles) in the Mesh or Material objects. Thus, GraphicsAgent and SceneGraphAgent collaborate on resource lifetime (SceneGraphAgent will inform when a mesh is no longer needed so GraphicsAgent can free the buffer, etc.).

- *With ShaderAgent:* SceneGraphAgent interacts via the Material system. A Material may reference a shader by name or ID; SceneGraphAgent will ensure that request gets fulfilled by ShaderAgent. For example, if a Material is set to use "BlinnPhong" shader, SceneGraphAgent either calls ShaderAgent to compile it (if not already) or assume ShaderAgent did so during initialization. They coordinate on ensuring the needed shaders are ready before rendering. SceneGraphAgent might also pass uniform data (like light positions, colors) to ShaderAgent's structures or directly to GraphicsAgent's uniform buffer updates – depending on who manages the constant data. (Possibly the backend will have an API to update uniform buffers, which SceneGraphAgent will call each frame with camera matrices, etc.)
- *With ComputeAgent:* As mentioned, if any aspect of scene updating is offloaded to GPU (like physics or particle updates), SceneGraphAgent will prepare and trigger those via ComputeAgent, then incorporate the results. For example, each frame SceneGraphAgent might do: `computeAgent.dispatch(task: physicsComputeTask)` and then later retrieve updated positions to apply to the SceneNode hierarchy. They ensure the compute task is complete before rendering that frame (or use the results from the last frame if doing pipelined simulation).
- *With GUI/other Agents:* (Not in the list, but worth noting) SceneGraphAgent may still coexist with the SDLKitGUIAgent (the 2D GUI agent). If the system uses an AI planner that can call both GUIAgent and SceneGraphAgent, we need to ensure they have distinct roles: GUIAgent for 2D windows and basic drawing, SceneGraphAgent for 3D content. They might share SDLWindow resources – e.g., an AI could open a window via GUIAgent and then use SceneGraphAgent to populate 3D content in it. We should ensure the protocols don't conflict. Possibly, we'd unify them in the future, but for now we treat them separately.

Communication and Autonomy

All agents will follow the repository's conventions (coding style, thread-safety rules, etc.). They communicate primarily through well-defined Swift interfaces and some shared data structures:

- **Interface contracts:** e.g., RenderBackend protocol (GraphicsAgent implements, SceneGraphAgent calls), Shader compilation interface (ShaderAgent provides, GraphicsAgent/ComputeAgent invoke).
- **Data exchange:** common data like mesh vertex data, textures, etc., is shared via references. For instance, SceneGraphAgent might load a model from disk (if that's in scope) and then call GraphicsAgent to create GPU buffers – passing the raw data as input and receiving a handle.
- **Autonomous implementation:** Each agent can be thought of as an AI coder responsible for a section. For example, if an autonomous system (Codex) is orchestrating, it could assign the ShaderAgent to write the shader compilation script, while GraphicsAgent writes `MetalBackend.swift`, etc., concurrently. AGENTS.md serves to prevent overlap and clarify responsibilities to those AI agents. They should respect each other's domains (e.g., ShaderAgent shouldn't directly fiddle with window code, that's GraphicsAgent's job).

In conclusion, these four agents collaborate to deliver the new SDLKit 3D features: **GraphicsAgent** provides the foundation and ties to hardware, **ShaderAgent** provides the cross-platform GPU code, **ComputeAgent** broadens capabilities to GPGPU tasks, and **SceneGraphAgent** offers a user-friendly 3D scene API. By dividing the work this way, we enable parallel development and clear focus areas, whether by human contributors or autonomous coding agents. Each agent's outputs feed into the next, creating an efficient pipeline for implementing this complex feature set.

1 3 SDL3/QuickReference - SDL Wiki
https://wiki.libsdl.org/SDL3/QuickReference

2 SDL3/SDL_GetWindowProperties - SDL Wiki
https://wiki.libsdl.org/SDL3/SDL_GetWindowProperties

4 5 glsl - How to write shaders that can be compiled for DirectX, OpenGL, and Vulkan - Game Development Stack Exchange
https://gamedev.stackexchange.com/questions/189515/how-to-write-shaders-that-can-be-compiled-for-directx-opengl-and-vulkan

6 7 Introducing: SDL_shadercross – Moonside Games
https://moonside.games/posts/introducing-sdl-shadercross/

8 AGENTS.md
https://github.com/Fountain-Coach/SDLKit/blob/16f3e8d36a254e359f3971c5be5e0edb6324c793/AGENTS.md